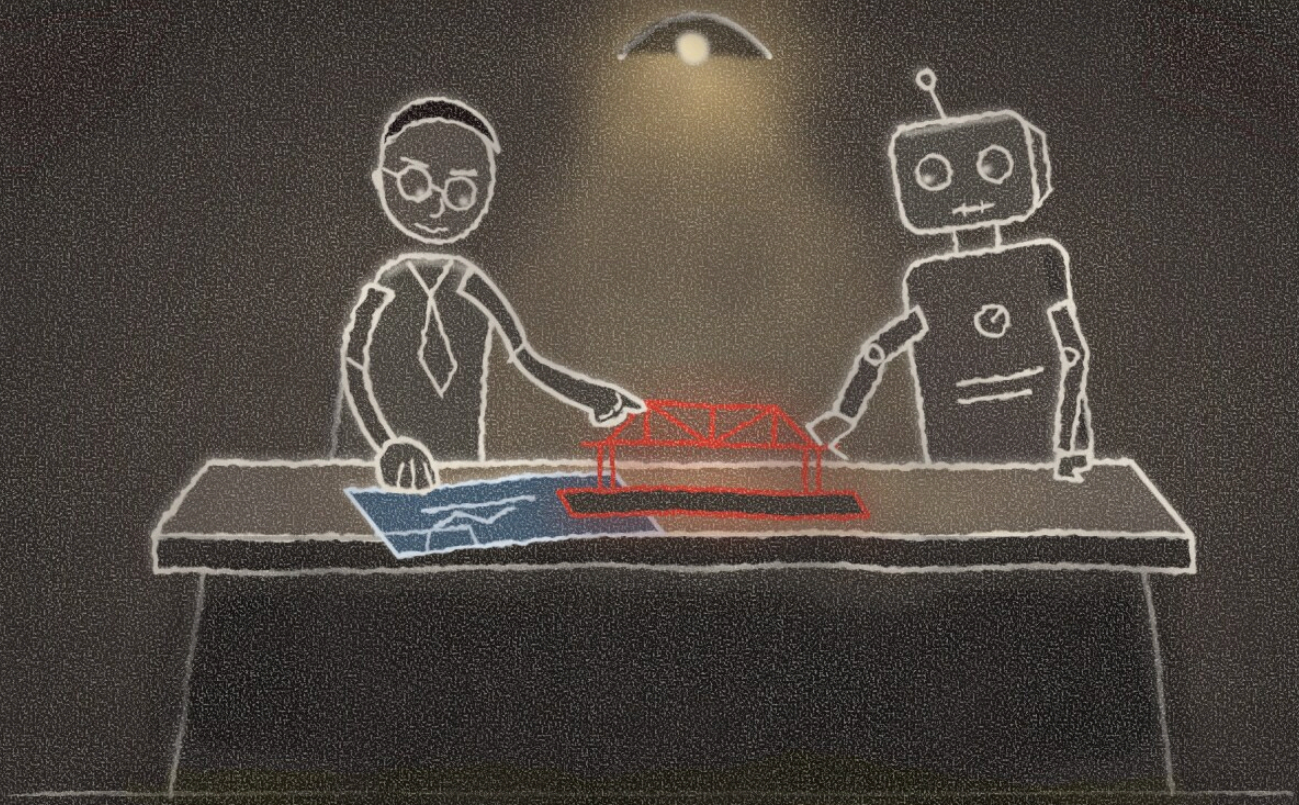
THE MAGE METHOD

Model-Based Agentic Software Engineering

JAMES C. DAVIS



Edition — last modified 2026-08-05

Acknowledgments

I thank K. Davis, H. Davis, W. Davis, N. Pond, N. Davis, R. Davis, A. Kazerouni, S. France, N. Eliopoulos, P. Jajal, P. Amusuo, B. Çakar, H. Peng, and u/donk8r for their involvement during the creation of this book.

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

The MAGE Method at a Glance

MAGE is one method with two theses and a conversion that grows an environment around them. To help you understand the full argument, Figure 0.1-1 shows the whole of it on a single page and how the parts relate to each other — the vocabulary this book builds, laid out in the order the method runs.

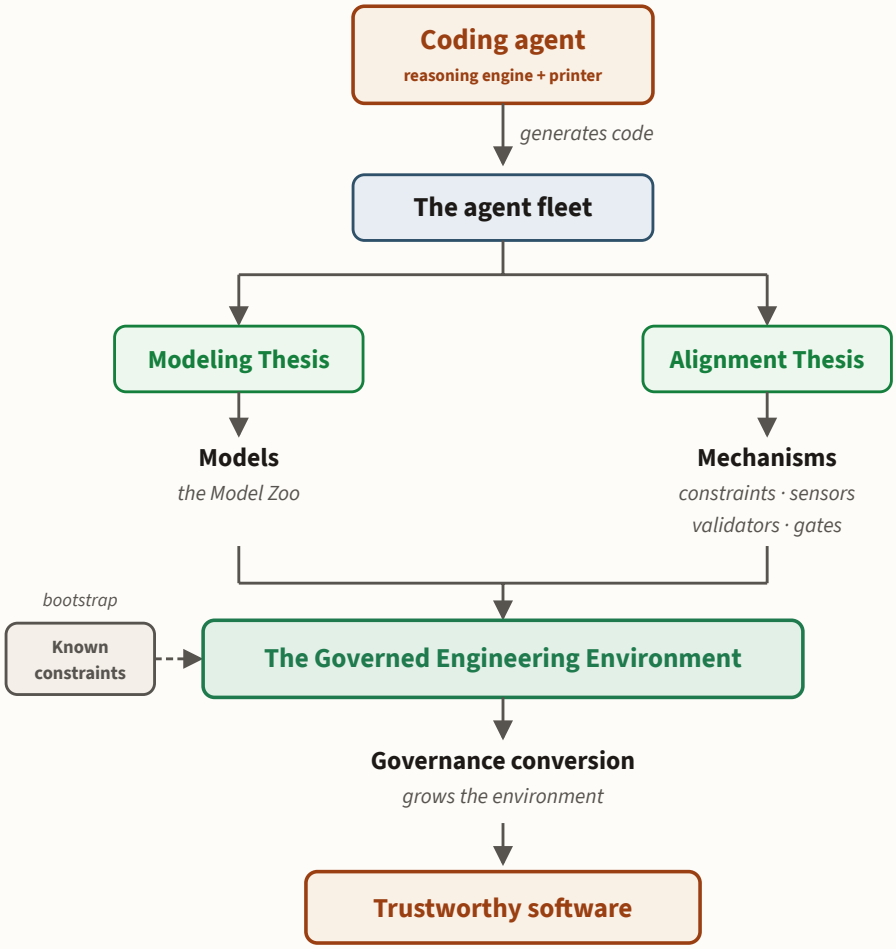


Figure 0.1-1: The whole method on one page: a coding agent — a reasoning engine paired with a printer — generates code through a fleet, and the two theses fill a governed environment. That environment is bootstrapped up front from known constraints, then grown by governance conversion into trustworthy software.

The Book's Language

Glossary

Agentic software engineering is a fast-evolving topic, and it seems that many concepts have been given overlapping or contradictory names. Part of the purpose of this book is to articulate coherent vocabulary for the field. The glossary is divided into four parts: the core ideas of the book, the vocabulary for coding agents and their accompanying phenomena, the terms for the governed engineering environment that stabilizes them, and a few terms from the case study that grounds the book.

The core ideas

The ideas the whole argument turns on — the two theses, the conversion that grows the environment, the governing metaphor, and the vocabulary of models underneath them.

Model. A cheaper, structured approximation of a system that an agent can reason through and an engineer can specify, analyze, and predict on. The base term the Modeling Thesis turns on.

Map and territory. A model is a map, not the territory: it earns its use by dropping detail and never stands in for the running system itself.

Modeling Thesis. Bind intent and system structure into an explicit, structured model, and the fleet gains a compact representation to reason through while the engineer gains a surface to specify, analyze, and predict on.

Alignment Thesis. A governance mechanism the environment enforces holds implementation to intent. A policy decided once then stands against every later change, so confidently-wrong work is made visible instead of shipped.

Governance Conversion. Converting a recurring failure into a durable engineering mechanism that permanently retires that failure class.

The Printer. The book's governing metaphor: treat a coding agent as a 3D printer, not a coder — it builds almost anything you describe, and only what you describe. A bad build is evidence about the instructions, not a limit in the machine.

Judgment is the scarce resource. The book in one sentence: agents make implementation cheap, so the expensive part becomes judgment — what to build, what "correct" means here, which failures deserve a wall and which a note. Those calls stay human. The discipline that pays converts each one into infrastructure, encoding a decision once so no agent has to make it again.

Middle-range theory. After Merton: not a universal law, not raw description, but a mechanism bounded by stated scope conditions. It names the register of this book's claims — a theory drawn from one deep case, transferable wherever its conditions hold, never a coefficient asserted over all software.

How coding agents work (and fail)

What a fleet of coding agents is, and the ways it comes apart.

Churn. The agent-fleet form of velocity decay: an increasing share of effort spent rediscovering context, undoing recent changes, repairing regressions, or reconciling inconsistencies rather than advancing the system.

Context Window. The bounded amount of prior conversation and code an agent can hold at once. Fixed the moment you choose the model, it is the sharpest driver of churn when the work outgrows it.

Foundation model. The trained model at the bottom of the agent stack. Choosing it fixes the context window and the reasoning ceiling every layer above works 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 hosts the hook, skill, and tool mechanism points.

Skill. A soft mechanism the agent *can choose* to enter: trigger criteria plus a body of guidance — prose, or prose mixed with code — for a recurring kind of work.

Tool. The code half of a skill: a library or command-line entry point that lets the agent take a deterministic action — rewrite a run of text, add a page — instead of guessing at it.

Fleet. The set of coding agents working the codebase — the agentic-era workforce this book governs, in place of a team of human engineers.

One-shot Scripting. The task mode for a small, unsubtle job: describe the process once, get it back in a single pass, no supervision loop.

Supervised Autonomy. The task mode for hard, reasoning-heavy work: you supply correctness conditions, tools that determinize the recurring steps, and monitoring to shape the process, rather than taking a single unsupervised pass.

Loop engineering. Shaping the agent's generate-check-correct loop — the correctness conditions, the tools that determinize recurring steps, the monitoring — instead of steering a single pass. Nearly everything in this book is an instance of it.

The Governed Engineering Environment

The environment that holds the fleet to a standard, and the mechanisms it is built from.

Governed Engineering Environment. What you get when engineering policy is encoded into the environment itself — models, constraints, sensors — rather than held in a person's memory.

Support ratio. Support-apparatus lines of code divided by production lines of code — a measure of how much governed environment a fleet has built around the product it governs.

Governance Mechanism. A repeatable environmental structure that encodes an engineering policy and does one of four things: constrains an action, detects a violation, requires evidence, or controls admission.

Constraint. A governance mechanism that prevents drift by scoping the action space, so the mistake is impossible to make in the first place.

Sensor. A governance mechanism that catches a violation after the fact instead of preventing it, failing the iteration so the mistake cannot ship unnoticed.

Validator. A governance mechanism of the evidence or admission class: it checks a candidate result against a standard before the result may advance, and refuses or flags what fails.

Gate. A check placed across a pipeline step, such as a commit or a deploy, that refuses to let the step through until its condition is met.

Lint. An automated check that scans code for a banned pattern and fails the commit when it finds one, so a rule holds without a human remembering it.

Hook. A hard mechanism the harness fires deterministically at an injection point — before or after the agent acts. The enforcement primitive that lints and gates are built on.

Invariant. A predicate over a model that must always hold — the testable claim a gate or a model-checker enforces.

Model drift. A model and the code it describes stop agreeing — the divergence a drift gate exists to catch.

Drift Gate. A build-time check that fails when a model and the code it describes disagree, keeping the model from decaying into stale documentation.

Structured (model). Said of a model written in an explicit, declared shape a machine can read and validate — a schema, not prose — so a build can check the system against it for drift.

Executable source-of-truth. An architecture that binds a model and the code that must agree with it tightly enough that a drift gate can block the build the moment they diverge.

Traceability. The round-trip join between a model and the code it governs: every model node maps to the code it constrains, and back.

Pattern. Following the Gang of Four's *Design Patterns* form — name, recurring problem, solution shape, consequences, known uses — the book writes each governance mechanism as a pattern, so an engineer can reach for a vetted answer instead of re-deriving one.

The Model Zoo. Part 3′s tour of the 4+1 architectural views — logical, process, development, physical, scenarios — as executable, drift-gated models, each walked on the real DocAble codebase.

DocAble specifics

Two mechanisms from the book's running example, DocAble.

Fidelity Validator. The mechanism that asserts a remediated document's meaning survived the remediation, catching the case where a file became more accessible and less true to the original.

Provenance Layer. The mechanism that stamps every artifact the system inserts, so a document's remediation history can be reconstructed and any single change explained or reversed. (Distinct from model provenance, which tracks what a derived model was generated from.)

Preface

This is a book about engineering — specifically, about what engineering work becomes when writing the code stops being the hard part. For most of the history of software, the scarce resource was implementation: someone had to sit down and type the thing into existence, and that typing was the bottleneck everything else queued behind. Coding agents have moved that bottleneck. Code is now cheap to produce and fast to change. What is not cheap is judging, constraining, validating, and maintaining it — deciding what to build, catching where fast code went wrong, and keeping a growing system trustworthy while it changes under you every day. That shift is the subject of this book.

My name is James Davis. I am a professor of software engineering in the Elmore Family School of Electrical and Computer Engineering at Purdue University. My job is to find out things other people do not know and write them down so they do. This book records one of those findings. Over a stretch of a few months I built a real production system — a document-accessibility service, DocAble — almost entirely by dispatching AI coding agents rather than writing code myself. I watched what that mode of work demanded, and this book is what I learned about doing it well. DocAble is the running example throughout; the claims here rest on that real production system. A first concrete look at it comes early, in MAGE by Example the full description lives at the back, in Part 5 (A MAGE Case Study).

I want to be plain about what this book is and is not. It is not a memoir of the experience, though the experience was strange enough to warrant one. It is not a survey of every AI coding tool, which would be stale before it printed. It is a field report and a method — *M*odel-Based *Ag*entic Software *E*ngineering, or MAGE: the concrete engineering practices that turn a fast code fabricator into a production line you can trust. Where a claim rests on a number, I give the number. Where a practice was learned by getting it wrong first, I say so.

One caution belongs up front, because it colors everything after it. This is a single-case field report: one production system, built by one engineer, watched over a few months. Where I give a number, read it as an observation from that case rather than a measured law. But this case joins a conversation already underway. Practitioner reports agree, in volume, that these practices work with agents: write far more tests than you would for a human team, specify before you build, make the environment hold the line. Those reports supply the *what*. What the conversation lacks is the *how* and the *why*: the mechanism behind the practices, worked out in enough detail to transfer. That is the contribution here. A deep case you control end to end is the right instrument for mechanism — I have built the machine, watched it run, and seen when it worked and how it failed.

A word about where the evidence comes from. Everything in this book rests on one system: DocAble, authored by me inside a single vendor's agent ecosystem, deployed and in daily use by me and my colleagues at Purdue. It is one deep case rather than a broad sample, and every claim stays scoped to it; the closing limitations set out exactly how far that reaches. The book also holds itself to the method it teaches. Its empirical claims are not loose lines in the prose; each is registered in a structured model that binds it to the datum or the citation behind it, and a check reports any claim left with neither. So where a sentence here rests on a measurement, the measurement is tracked, and a missing one shows rather than hides.

A footnote on the evidence. The claims and their backing sit in a small ledger beside the manuscript: each empirical claim tied to a supporting datum or a positioned citation, nested under the book's argument, with a query that surfaces any claim carrying neither. It is the same discipline

the book turns on code, turned here on the book's own argument. The colophon lays out the full set of models that govern the manuscript.

Why Build a System to Study a Method?

Building DocAble was not only a way to ship a product. It was a way to ask a question that is hard to ask from the outside. Once agents write almost all of the code, what is the engineering that remains? The question is no longer whether they can produce useful code. They can. It is what is left for a human to do when they produce nearly all of it.

That question resists the usual instruments. A benchmark isolates one capability and scores it. A build report tells you a fleet reproduced a framework, compiled a kernel, or migrated a codebase in a week. A set of interviews tells you how developers supervise the tools they were handed. Each is a real piece, and none of them shows how the pieces move together over the life of a new production system: how an architectural mistake hardens into a lint, how a failure seen twice becomes a gate, how one authored model changes what an agent can do, how the controls pile up until the environment can be trusted to run itself. You cannot watch that from a scorecard or a single interview. You have to build the system, hold it to a standard someone would actually rely on, and watch it over months.

So that is the design of the study. I built something whose quality mattered, let the fleet do nearly all of the implementation, and treated every failure as evidence about the method rather than noise to clean up. The book's vocabulary — governance conversion, the Modeling and Alignment theses, constraints and sensors, the governed engineering environment — names structures that recurred often enough to earn names. I offer them as working concepts, precise enough to compare, argue with, and replace, not as the field's settled terms. Where my investigation extends the existing literature, I have tried to cite the work it stands on, so a reader can see where this case joins the conversation and where it departs.

Read the book in that spirit. One deeply worked case cannot prove that every project will behave like this one. What it can do is open a single machine far enough to say what the rest of us should look for.

Where this book sits on the shelf

Three books frame what this one attempts.

The premise: mechanize discipline

The first is **Software Engineering at Google** (Winters, Manshreck, and Wright, 2020)[1] — three Google engineers' account of how the company sustains software engineering across tens of thousands of people and one vast codebase over decades — whose premise this book shares: engineering discipline should be *mechanized, not remembered* — pushed into review, testing, tooling, and culture so that correctness does not depend on any one engineer keeping a rule in their head. This book carries that premise into a different era and a different workforce. Google governs thousands of human engineers. Here the workforce is a fleet of coding agents, governed by the lints, structured models, and gates they run. The failure mode is not a tired reviewer; it is a confident machine that produces plausible, subtly wrong code across every file at once, at a rate no human can read. The mechanization has to be sharper because the thing being governed is faster and less accountable.

[1] Titus Winters et al., *Software Engineering at Google: Lessons Learned from Programming over Time* (O'Reilly Media, 2020).

The deeper parallel is what each book is fighting. SE@Google exists to keep productivity *linear* as the team grows. It holds off Brooks's Law: the point where adding an engineer buys less than the last one did, then buys nothing, because the communication paths between N people grow as N-squared and the coordination overhead eats the gain. This book shares the same goal, sustaining linear productivity as you scale up, but scales agents instead of people, and the wall you hit is a different one. An agent does not drown in an N-squared communication circle. It hits a different bound: a fleet is limited by how well its models reason and how much they can hold in view, and when it reaches that bound, ungoverned, it decays into churn. Context pressure is the sharpest driver — when the work outgrows what the fleet can hold in view, the agent loses the thread, re-derives what it already built, and confidently undoes yesterday's fix. That reframing is what the rest of the book turns on: the wall is churn, not the coordination overhead that bounds a team of people.

Churn's causes are three things the fleet does not know. It does not know *what* to build: the requirements are ambiguous. It does not know *how* to realize the change: the architecture has drifted into a soup no context window can hold. And it does not know how to change the system *without breaking it*: every edit lands blind on invariants nobody wrote down. Every technique here attacks a cause, not the symptom. Models attack the first two not-knowings — they bind intent to structure and hold the architecture in a compact form, so the agent knows what to build and the work fits in-window. Governance attacks the third: it keeps a change from introducing errors in the first place, or makes them visible the moment it does.

> **Churn.** *The condition in which an increasing share of agent effort is spent rediscovering context, undoing recent changes, repairing regressions, or reconciling inconsistencies rather than advancing the system. It is the agent-fleet form of velocity decay: fixing crowds out building.*

The stakes here are **tokenomics**, not just tidiness. Keeping a task inside the window is far cheaper than letting it spill, and the gap compounds: I once ran four Claude Max subscriptions flat out to keep the fleet fed, and the same velocity now barely fills two — because a model lets each agent find what it needs instead of re-deriving it, and keeps the work in-window, off the cliff where the token bill jumps.

The grammar: patterns, and two theses

The second is the **Gang of Four's** *Design Patterns* (Gamma, Helm, Johnson, and Vlissides, 1994)[2], the catalogue that gave object-oriented design its shared vocabulary of reusable solutions, whose lasting contribution was less any single pattern than the *form* — name, recurring problem, solution shape, consequences, known uses — written so an engineer could reach for a vetted answer instead of re-deriving one. This book borrows that form for a new subject. The governance mechanisms it teaches are written as patterns: each names a failure class, the shape that prevents it, why it is not merely the cheaper thing everyone already does, and where it has been used. The companion catalogue restates the full set in that layout, and the appendix here is a patterns reference in the classic style. If SE@Google supplies the premise, the Gang of Four supplies the grammar.

It also supplies a useful contrast. Behind the Gang of Four's twenty-three patterns sits essentially one idea — *add a level of indirection* — dressed a dozen ways. Two theses drive this book, and they run through every chapter that follows.

[2] Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994).

The Modeling Thesis. Binding intent and system structure into an explicit, structured model gives agents a compact, coherent representation to reason through, and gives engineers a surface on which to specify, analyze, and predict the system.

The Alignment Thesis. A governance mechanism the environment enforces keeps implementation aligned with intent — a policy decided once holds against every later change — so confidently-wrong work is prevented, or made visible, instead of shipped.

The two theses divide the three not-knowings named above. The Modeling Thesis treats the first two — knowing what to build and how to realize it — by binding intent to a model compact enough to hold in-window; the Alignment Thesis treats the third, changing the system without breaking it, by catching the confidently-wrong edit before it compounds. A model chapter is the Modeling Thesis worked out; a lint, gate, or validator chapter is the Alignment Thesis. Hold those two in mind and the book stops being a list of tricks and becomes two ideas applied over and over. Figure 0.3-1 draws the whole method in one picture.

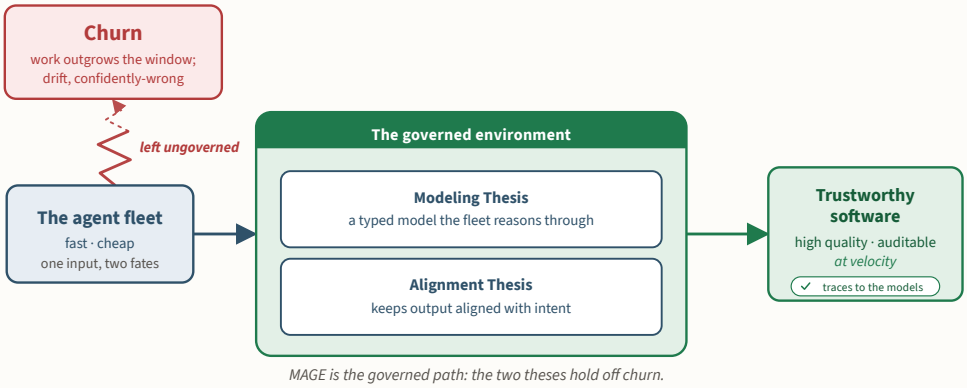


Figure 0.3-1: The MAGE method in one picture. A cheap agent fleet, left ungoverned, drifts into churn as its work outgrows the context window. Governed through the Modeling Thesis and the Alignment Thesis, it converges on trustworthy software at velocity.

Hold both theses at once and a larger claim comes into view — the one this whole book argues toward. **Models are the universal language of engineering.** Every mature engineering discipline reasons about the system it builds through models, not through the raw artifact: a load diagram, a circuit, a control loop. Software could rarely afford to, because keeping the model equal to fast-moving code cost a person's standing labor. Agents cut that recurring cost sharply. They make the code cheap and run the drift checks that keep the map in sync for almost nothing — someone still authors the model and the gate — and that shift relocates the engineer's job upward — to the essential part it could never fully reach before: deciding what the system is *for*, and authoring the models whose properties drive the tradeoffs. In the system that underpins this book, the code stopped being the bottleneck; the model became the work. For software engineering, models are an idea whose time has come.

And a model cheap enough to keep true stops being a picture of the system and starts running it. The fleet reads its models at commit time to grade its own gates, and at measurement time to shape its own metrics. The map no longer merely stays equal to the territory; it steers the machinery that keeps it equal.

The lineage: the modeling tradition

The third is **Model-Driven Software Engineering in Practice** (Brambilla, Cabot, and Wimmer, 2017)[3], the standard concise statement of the modeling tradition: treat models as first-class engineering artifacts, structured representations with defined semantics a tool can check, not pictures pinned beside the code. That tradition is the lineage this book descends from — the *M* in MAGE — and it inherits the tradition whole, minus its two prescriptions. Classical model-driven engineering prescribes an authoring direction (write the model, generate the code from it) and a heavyweight tool stack to make the generation work. MAGE needs neither. Its claim is about maintenance, not authorship: model and code must stay equal, and agents now keep them equal under drift gates, whichever artifact came first — model authored and code derived, model induced from existing code, or the two co-evolved. The tradition's moment arrives now because one model does two jobs at once: it solves the agent's context problem, since the fleet reasons over a compact picture instead of the whole subsystem, and it holds quality, since the gates keep the map equal to the territory. The executable zoo makes the full inheritance argument.

The three together leave a gap this book means to fill: agentic-era *methodology*, not how to build an agent but how to engineer with a fleet of them and trust what ships. The titles that share this book's era mostly teach how to build agentic systems, where the agent is the product. This one runs the other way — the agents are the workforce, and the subject is the engineering that governs them.

[3] Marco Brambilla et al., *Model-Driven Software Engineering in Practice*, 2nd ed. (Morgan & Claypool, 2017).

Three ways to run a fleet

The hard problem of building software with agents is simple to state: the tools are fast and the tools are unreliable. A coding agent can produce more change in an afternoon than a team used to produce in a week, and some fraction of that change is confidently, plausibly wrong. Every account of how to work this way is really an answer to one question — where does the reliability come from? Three answers dominate. Figure 0.3-2 sets the three side by side.

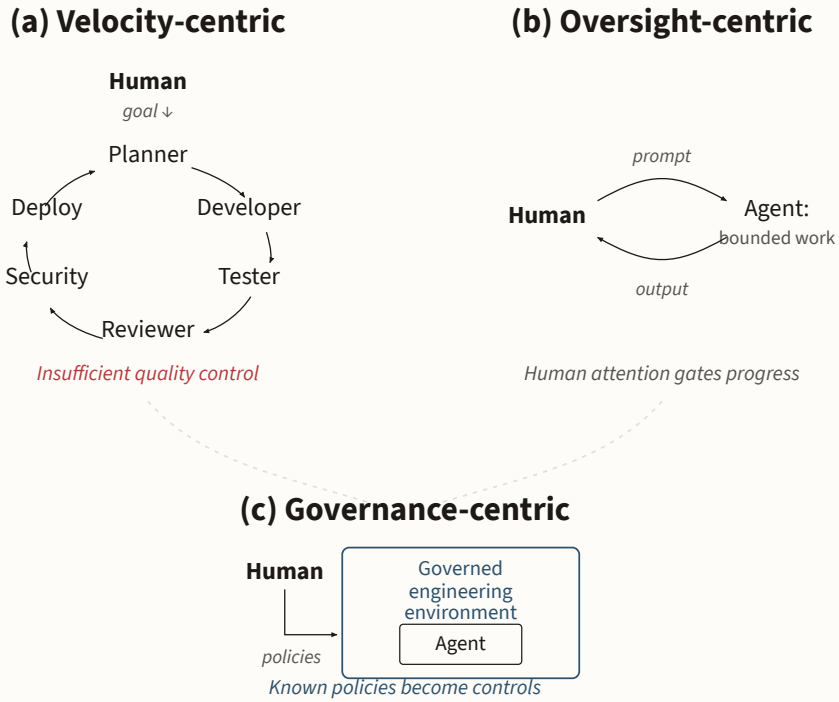


Figure 0.3-2: Three process models for agentic software engineering. Velocity-centric rings job titles around an implicit quality mechanism; oversight-centric straddles a human over each bounded piece, whose attention does not scale. Governance-centric is the synthesis — agents work inside enforced mechanisms set up once, so reliability scales.

The first answer is velocity. Give the agents autonomy, and organize them the way you would organize people: a planner hands work to a developer, the developer to a tester, the tester to a reviewer, and around the loop it goes. Assigning agents job titles makes the work look like engineering, and the throughput is real. But process resemblance is not control. A ring of role-labeled agents can imitate the *form* of software work — the handoffs, the review step, the sign-off — without any of it actually catching a bad change. The quality mechanism is left implicit, and implicit quality does not survive contact with volume.

The second answer is oversight. Treat the agent as a useful but untrustworthy assistant, and keep a human next to the implementation loop. You prompt a bounded piece of work, the agent does it, you read the result, you prompt the next piece. This is honest about the failure mode, since agent output does need inspection, and for a while it works well. Its limit is arithmetic. The control is human attention, and attention does not scale with the agents. Turn the volume up and the human becomes the bottleneck: the fleet produces changes faster than any person can read them, and the reading is now the slow step in the whole line.

Field work fills in what that oversight actually involves. Interviews with seventeen experienced developers working alongside coding agents identify four recurring forms — a priori control, co-planning, real-time monitoring, and post hoc review — and find that agent-written code is often hard to inspect directly, so developers lean on tests and other indirect evidence to judge it[4]. But naming the work does not repeal its arithmetic: every one of those forms still spends human attention per change.

The third answer is governance. Build the environment before the agents work in it. Decide the obligations up front — what must never happen, what must always hold — and encode each one as a mechanism the environment enforces: a type the compiler checks, a lint that blocks a commit, a gate that refuses a bad deploy. The human hands the environment its policies; the environment holds the agents to them. Quality stops depending on a tired reviewer noticing, and starts being a property of the ground the agents stand on.

These three stances are the landmarks of a live debate, and it helps to place the names on them. At the velocity end sits *vibe coding*, the term Andrej Karpathy coined for prompting an agent and accepting what looks right — Steve Yegge's Gas Town is the same spirit written large. At the far end sits the rigor of formal verification, argued by Bertrand Meyer in *From Probable to Provable*[5] and pursued by the emerging vibe-OS and vibe-tools work on formal tooling for AI-written code. This book stands between them — closer in temperament to the rigorous end, but reaching its assurance by a different route than a proof.

Notice what the first two answers have in common: they are the two ends of one axis. Slide the dial toward velocity and you buy speed by spending reliability; slide it toward oversight and you buy reliability by spending speed. Every point on that line charges the same toll: per-change human attention, whether the human is coaxing the output at the fast end or inspecting it at the slow end. Call it the *pet tax* — the price of caring for each change by hand. It grows with the size of the fleet until the human becomes the bottleneck no matter which way the dial is set.

Governance is not a point on that line. It is a different move. Instead of choosing where to sit between speed and safety, you change what the reliability is made of: you build the fences and chutes once, up front, and let the guardrails ride every change on their own. The old distinction between *pets* and *cattle* names the shift exactly. A pet server is nursed by hand, one at a time; a herd of cattle is run by the fences and the routine, not by attention paid to each animal. Governed agents are cattle. You stop paying attention per-change and start paying it per-*class-of-failure* — once, when you build the mechanism that retires the class. The dial-trading stops, because you are no longer on the dial.

This book takes the third stance, and then pushes past where the current accounts stop. The governance literature starts from obligations you know in advance (a regulation, an access policy, a rule the organization already wrote down) and asks what mechanism would enforce it. That is the easy half. The hard half, the half I lived, is the obligations you *cannot* know in advance: the ones that stay invisible until an agent, moving fast, trips over them and breaks something. The engineering that matters is what you do next — reading that failure as evidence of a missing mechanism, and encoding it so the whole class of failure cannot happen again. Velocity is what exposes the missing mechanism. Judgment is what converts it into a guardrail. That conversion, done over and over until the environment can be trusted to run itself, is the engineering this book is about.

---

[4] Shipi Dhanorkar et al., "Human Oversight of Agentic Systems in Practice: Examining the Oversight Work, Challenges, And Heuristics of Developers Using Software Agents," 2026, https://arxiv.org/abs/2606.05391.

[5] Bertrand Meyer, "From Probable to Provable," *Communications of the ACM*, ahead of print, 2025, https://doi.org/10.1145/3773295.

The through-line, and who it is for

The through-line is a single idea; hold every chapter against it. **When you build with an agent, you are not managing a coder; you are running a printer.** A 3D printer builds almost anything you can describe, but only what you describe — hand it a bad model and you get a bad part, so read a bad build first as evidence about the instructions, the representation, and the task boundary, not by reflex as a limit in the machine. An agent is the same. It builds most of what you can explain to it, and the entire discipline of this book lives in that "so long as you can explain it." Explaining a system to a probabilistic machine well enough that it builds the right thing, at speed, without a human reading every line — that is a real engineering craft, not a knack, and it can be taught.

This book is for the software engineers who suspect the ground has moved and want to know where to stand. On the surface it is about a tool I built to help people with disabilities access documents. Underneath, it is about a working method for building anything with agents: how to frame the problem, model the world for the machine, put every change on rails from dispatch to production, and convert each failure you hit into a guardrail that stops the next one. You will not need my domain to use any of it. You will need a codebase, an agent, and the willingness to treat the instructions as the job.

Everything that follows is one move, repeated: convert judgment into infrastructure.

How to read this book

Five parts, and they compose in a particular order. **Part 1** sets the frame — why code got cheap, and the two theses everything rests on. The next two parts work one thesis each: **Part 2** develops the Alignment Thesis as the controls that govern a fleet — lints, gates, the agent stack, the graph that catches guardrails colliding; **Part 3** develops the Modeling Thesis as the executable, drift-gated models the fleet reasons through. **Part 4** puts both to work as daily practice, and **Part 5** is the grounding case: DocAble end to end, the real system the claims rest on. The Theory of MAGE, the closing implications and conclusion, and the appendices follow. Figure 0.4-1 shows the whole path and the routes through it.

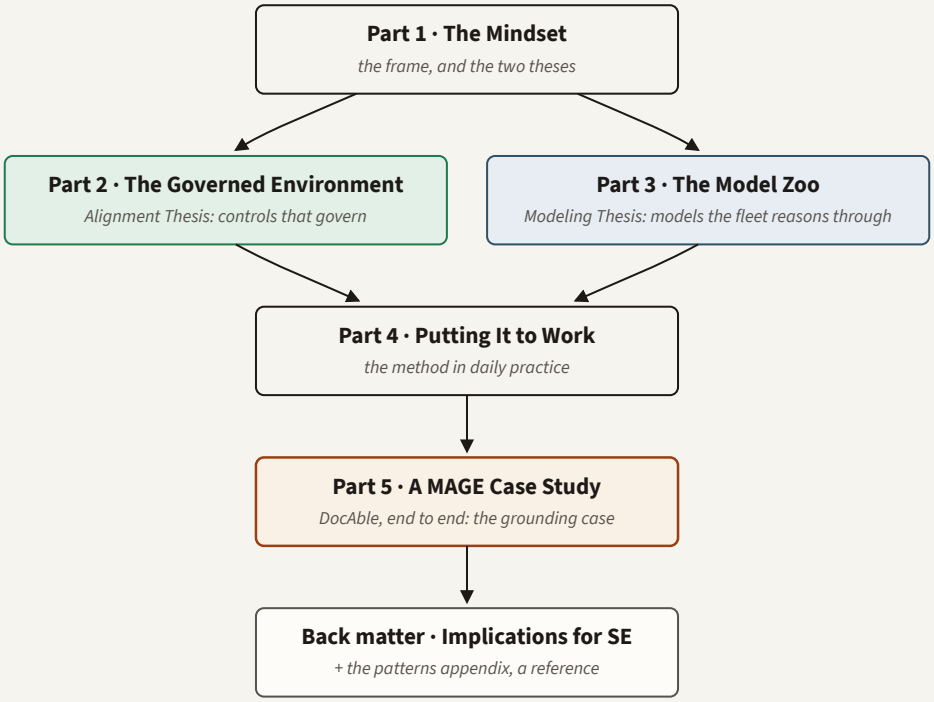


Figure 0.4-1: The book in one map. Part 1 sets the frame and the two theses; Part 2 works the Alignment Thesis into fleet controls, Part 3 the Modeling Thesis into models. Both feed Part 4 (the method) and Part 5 (the DocAble case study), then the closing implications and appendix.

Read it straight through and the argument builds in order. But there are faster ways in. If you want to *do* this on Monday, read Part 1, then jump to Part 4 and dip back into Parts 2–3 when a technique needs its foundation. If you want to be *convinced* first, read Part 1, then Part 5 (watch the method survive a real production system), and come back for the how. And the appendices are a reference, not chapters: consult a pattern when you meet its failure, the way you would the Gang of Four.

Two ways to read this book

Read the book once, front to back, and you learn the method. Return to it while you build, and it works a second way.

- **The first pass is sequential.** Each part rests on the one before it, from the two theses through the method and into the DocAble case, then out through the back matter — the Theory of MAGE, its implications, and the conclusion — that closes the argument. Read in order and it arrives assembled.
- **Every pass after that is not.** Once you are standing up a governed environment of your own, the book turns into a reference you open at the page you need. The appendices carry that second mode. They reward repeated consultation, not a straight read, and most readers will reach for them long after they have set the chapters down.

The appendices are the reference interface to MAGE once you have the method. Treat them as part of the book's working surface, not as back matter tacked on behind it.

What each appendix is for

Five appendices, each with its own job. The first three form a reference triad you work in sequence; you reach for a different one depending on what you are asking.

- **Appendix A — MAGE Engineering Stacks.** Reusable engineering architectures: the clusters of mechanisms that travel together to stand up one capability, with the mandatory members marked off from the ones you can leave out. Reach here when the question is *"How should I build this capability?"*
- **Appendix B — Flagship Mechanisms.** The engineering judgment behind the mechanisms that carry the most weight — the failure each one kills, the shape that kills it, the alternatives it replaces, and the seam where you wire it in. Reach here when the question is *"Why is this the right decision?"*
- **Appendix C — Mechanism Catalog.** The whole vocabulary at a glance: every mechanism as a compact brick with its figure, a short summary, and its metadata, each linking on to its fuller treatment. Reach here when the question is *"What is this mechanism again?"*

The last two appendices stand apart from that triad, each answering a question the build sequence does not raise.

- **Appendix D — Operator's Reference.** The operator's dashboard: the metrics that steer the work while it is in flight and the verdicts that certify the system at maturity, each with what it counts, when to read it, and the direction that reads as healthy. Reach here when the question is *"What should I watch while running the system?"*
- **Appendix E — How to Write a Skill.** A build recipe, not a reference layer: the step-by-step for packaging a body of judgment into a skill an agent can reach for.

Working the appendices together

The three reference appendices are strongest read in sequence, from the capability you want down to the detail you need to ship it.

- **Start with the capability.** Appendix A names the stack that delivers it and the mechanisms that compose it.
- **Study the judgment.** Appendix B explains why each load-carrying mechanism takes the shape it does.
- **Keep the map open while you implement.** Appendix C gives the quick recognition and the jump to a deeper entry.
- **Go to the online catalogue for the complete documentation** when print runs out of room.

Read in that order, the appendices reinforce one another rather than repeat. Each answers a question the one before it raised.

The book and the online catalogue

The book and the online catalogue divide the work.

- **The book teaches the method and the engineering judgment behind MAGE** — why the mechanisms take the shapes they do, and how they compose.
- **The online catalogue is the complete reference for the mechanism library** — every mechanism as a full Gang-of-Four entry, with implementation guidance, examples, and the detail that does not fit in print.

Come to the book to understand the ideas. Go to the online catalogue when you need the exhaustive record of one mechanism.

Where to look

Table 0.4-1 lays the surfaces side by side — each with the job it does and the question that sends you to it, from the main text through the five appendices to the online catalogue.

Table 0.4-1: Where to look — each surface of MAGE beside the job it does and the question that sends you to it, from the main text through the five appendices to the online catalogue.

Resource	Primary purpose	Typical question
Main text	Learn the method	*How does MAGE work?*
Appendix A — MAGE Engineering Stacks	Compose capabilities	*How should I build this capability?*
Appendix B — Flagship Mechanisms	Engineering judgment	*Why is this the right decision?*
Appendix C — Mechanism Catalog	Recall the vocabulary	*What is this mechanism again?*
Appendix D — Operator's Reference	Operate the running system	*What should I watch while running the system?*
Appendix E — How to Write a Skill	Package judgment into a skill	*How do I build a skill?*
Online catalogue	Complete reference	*How do I implement this mechanism?*

The Theory of MAGE — the causal account of why the method follows from cheap code — lives in the **main text**, near the end just before the implications, not in the appendices. It belongs to the argument, so it reads in sequence with the chapters that earn it.

Part 1

The Mindset

1.1 The Printer

Coding agents have made writing software cheap. Describe what you want and a capable model builds it, fast, at a scale no team could match. Implementation is abundant now. What has not come with it is reliability: the same machine builds the wrong thing as readily as the right one, and just as fast.

So the chapter opens on the question the whole book turns on: when an agent builds the wrong thing, whose fault is it? Treat the agent as a *printer*, not a coder. A printer builds exactly what its instructions describe, so read a bad build first as evidence about the instructions, the representation, the task boundary, and the environment that produced them — not, by reflex, as a fixed ceiling in the machine. That shift in posture is the premise everything after this chapter rests on. If implementation is cheap and unreliable, the engineering moves to authoring what the printer reads.

1.1.1 What this book is about (and what it is not)

Of course, not every programming task needs the machinery this book describes. Some tasks a capable model can simply do. You describe the input and the output, you push the button, and a working program comes back on the first pass. No iteration, no governance, no loop management. Knowing which tasks are like that, and which are not, is the first judgment an agentic engineer has to make. Figure 1.1-1 lays out that judgment as a flowchart.

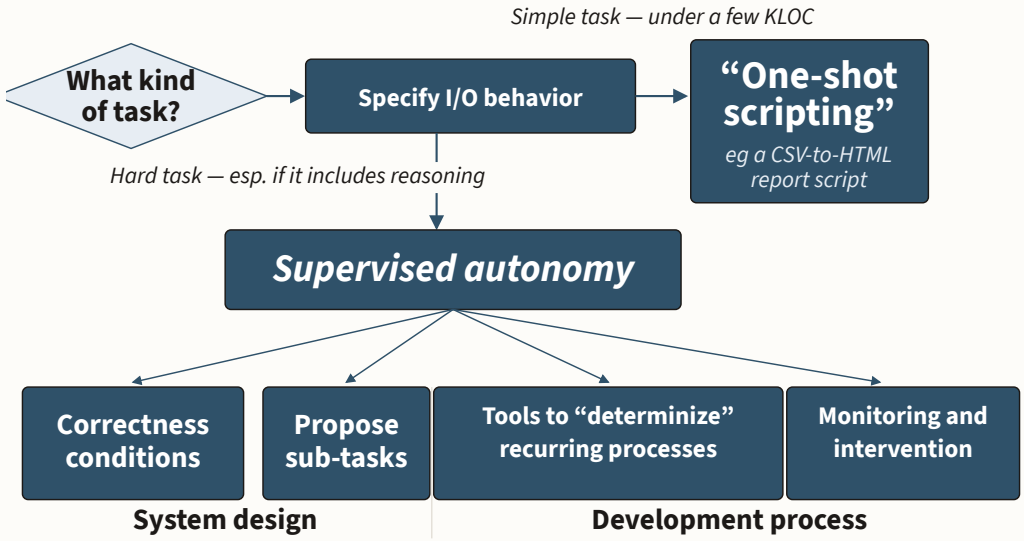


Figure 1.1-1: The Task-Mode Heuristic. *Specify a task's input/output behavior, then split on size and difficulty: a simple no-reasoning task goes to one-shot scripting, a hard or reasoning-heavy one to supervised autonomy. If unsure, try to one-shot it, but switch paths rather than throw good tokens after bad.*

The two modes

Start at the decision on the left. You have a computational task and you can say what it takes in and what it must produce. From there the path forks on how big and how hard the work is.

One-shot scripting is the righthand branch. The task is small, a few thousand lines at most, and it does not ask the agent to reason its way through anything subtle. Think of a CSV-to-HTML report script. You specify the behavior once, the agent writes it in a single pass, and you read the result. The whole transaction fits in one context window and one sitting. When a task really is this size, the full apparatus of governed autonomy is wasted effort. Send it one-shot and move on.

Supervised autonomy is the downward branch, and it is where most of this book lives. The task is hard, especially if it includes reasoning, and it is too large to hold in a single context window or to trust to a single pass. So you do not hand the agent the whole job and hope. You supply four things, in two groups. The system side shapes what gets built:

- **Correctness conditions** — what a right answer looks like.
- **Sub-tasks** — the breakdown you have proposed to divide the work.

The development side shapes how the building runs:

- **Tools that determinize recurring processes**, so the same operation runs the same way every time.
- **Monitoring and intervention**, so you can watch the work and step in when it drifts.

Those four are the skeleton of supervised autonomy, and each later part of the book fills one of them out.

Where the emphasis falls

This book covers both modes. It has to, because the first skill is telling them apart, and getting that wrong in either direction is expensive. Treat a one-shot task as if it needed supervised autonomy and you drown a five-minute script in ceremony. Treat a supervised-autonomy task as if it were one-shot and you get confident, plausible, subtly wrong output at a scale no human can read.

So the book spends real pages on the sizing question. How long a leash a task can run on comes down to two things: how densely the model's training data covers the kind of work you are asking for, and how capable the model is at that work in the first place. The denser the coverage and the stronger the model, the more briefly a task can be described and the farther it can be trusted; the nearer the task sits to the edge of what the model has seen, the more the reasoning must be broken into smaller, checked steps. Judging that distance tells you whether a job can go one-shot or must be supervised, and the book gives you the means to judge it.

But the weight of the book is on the downward branch. The interesting engineering, the part that did not exist a few years ago and is not yet written down, is **supervised autonomy**: building larger-scale products whose quality needs exceed what a model can be trusted to do alone, and whose size exceeds current and foreseeable context windows. That is the regime where you stop pressing go on a script and start running a printer, framing the goal, modeling the world for the machine, and putting every change on rails from dispatch to production. Everything after this chapter is about how to do that well.

Writing code used to be the expensive part. It no longer is. So the premise of the book is short: we've got a fast code-fabricator and we need to figure out how to turn that into a production line.

1.1.2 The printer

Think of a 3D printer. It will build almost anything you can imagine — a replacement gear, a prosthetic hand, a scale model of a building — out of nothing but spooled filament. Its only hard limits are the material and your own creativity. That machine is the metaphor that drives this entire book, because building software with a coding agent is far more like running a 3D printer than most people expect — a difference the rest of this chapter unpacks.

1.1.3 An open-ended technology

A 3D printer belongs to a rare class of technology — the *open-ended* kind — alongside the invention of writing and the printing press. What those three share is that none of them arrives with a list of what to make. People had to invent that. Writing began as marks for counting cattle; only later did someone find you could write a poem, and later still that a poem could rhyme, could take a stanza, could hold a shape — and then the textbook, the treaty, the contract. Each was a *genre* somebody had to work out. The printing press inherited every genre writing had found, and then, by making production cheap, minted ones no scribe could ever have justified: the poster, the pamphlet, the newspaper.

Generative AI belongs to this class. It is open-ended in the same way — it will make almost anything, and it hands you no list. So running the machine is the smaller half of the work; the larger half is working out which genres are worth producing, and the method for producing each one well. This

book takes one genre and gives it a method: **production software** — code whose quality metrics you actually care about, and whose every change you can trace back to the metric it moves.

> **The quality floor, made cheap.** *The apparatus in this book — the models, the gates, the traceability — is the price of* any *software you can trust: name the properties that must hold, check them, and keep every change tied to the property it moves. That price was always the same. What changed is who can afford it. For decades this level of assurance was reachable only by the shops that could justify the cost — the Rolls-Royces and the Boeings, where a mistake is catastrophic and a modeling program pays for itself. Agents drop that cost to almost nothing: they write the checks, maintain the models, and hold the line, so the quality once affordable only to a regulated few is now within reach of almost anyone. The method teaches how to get very high quality nearly for free. For the data, see The ADA Context →*

1.1.4 A picture is not enough

Everyone can sketch a piece of software. We have all used Google Maps, spreadsheets, a document editor — programs that take in data and do something useful with it. The concept is easy to picture. But a picture is not a thing you can print.

Hand a 3D printer a photograph of the object you want and nothing comes out. The printer needs instructions: a CAD model that shows not just the outer shape but the internal structure — how the pieces fit, where the load is carried — and then a set of print notes on top of that. You slice the model into layers, and you think about how those layers compose as the machine lays them down. Different materials have different minimum thicknesses, different strengths, different joints that hold and joints that fail. There are a hundred ways to misprint a part that looked fine as a picture.

Software agents are the same. You cannot hand an agent only the picture of what you want. You must also give it — or work out how to articulate — the detailed instructions for how to build it. Good instructions, and it works beautifully. Bad instructions, and you have not got a hope.

1.1.5 Whose fault is it?

Here is the shift in posture the whole book turns on. When you use a stapler and the staple jams, that is the stapler's fault, not yours. A stapler is supposed to just work — it is so foolproof my three-year-old runs one, insert the paper, ka-chunk, no problem. That is exactly what an agent is not. An agent is not a stapler — it is a printer, and a printer only ever builds what its instructions describe. So when the agent produces the wrong thing, suspect the instructions first.

In my own experience, when an agent has failed to produce what I wanted, the fault has almost always been mine — the instructions, the way I represented the problem, where I drew the task boundary. Look there first, before you conclude the model simply cannot do the work. The strong models build most of what we can envision in software — *so long as you can explain it to them*. That "so long as" is the entire job, and it is an engineering discipline this book teaches. And the discipline is older than it looks: an engineer has always been accountable for the instructions handed to a machine, even the ones not hand-crafted — the back matter returns to what that responsibility becomes when the thing you author is the model, not the code.

The printer builds anything you can describe. That, of course, is the promise, and it is real. But it is also the danger, because a printer with no judgment behind it builds a wrong thing as fast and as willingly as a right one, and carries you at full speed exactly where your instructions point.

> Claude is the fastest road to hell.

The danger was already in the instructions; the speed just gets you there first. Hand the machine a doomed idea and it does not slow down to warn you; it makes the doomed idea better and better, an afternoon and ten thousand plausible lines deep into a dead end you picked. The printer that builds anything will build your mistake with the same tireless fidelity it builds your success. Which one you get is down to your instructions.

A word on the example this book builds on. The running example is a real shipped system: DocAble, the production accessibility-remediation tool — its own pipeline, its own agent fleet, its own scars — described in full at the back, in The Built System. Every later chapter returns to it, so what you watch getting printed correctly, chapter by chapter, from model through mechanism and architecture to migration, is a system that shipped. The next chapter gives you a first concrete look.

1.2 MAGE by Example

The last chapter closed on a promise: the running example of this book is a real, shipped system. Before the abstractions arrive — the loops, the models, the governed environment — you should hold enough of that system in your head to attach them to. That is this chapter's whole job. What kind of object needed all this machinery? How does it fit together? And why did I build it in the first place?

1.2.1 Two birds, one stone

The explicit problem is a mountain of inaccessible documents. In 2024 the Department of Justice gave public universities a deadline: the slide decks, PDFs, and spreadsheets they distribute must meet a real accessibility standard, so that a student using a screen reader gets the same course as everyone else. A typical deck fails that standard dozens of ways — images with no descriptions, missing titles, a reading order no machine can follow. Remediating one course's materials by hand costs roughly 7.5 full work weeks, and a university offers thousands of courses. The mandate is just; the arithmetic is crushing. The ADA Context walks the full bind.

The implicit problem was mine. In early 2026, coding agents got good — good enough that I, a professor of software engineering, no longer knew what my discipline's daily craft was about to become, or what I should be teaching in the fall. Nobody did. The only way to find out was to build something real with the agents and watch what the work demanded. And I had a second, smaller motive sitting in my own course folders: hundreds of my own slides needed remediation, and I had no intention of doing it by hand.

The two problems met in a committee meeting about that very deadline. Bored, I opened a chat model, fed it screenshots from one of my own decks, and asked it to transcribe what it saw. It could. That five-minute experiment said the technical core was suddenly within reach — and one project could answer both problems at once. Build the remediation system, and let the building teach the method: two birds, one stone. The system is DocAble, live in beta. The method is MAGE, and this book is the field report from watching it emerge.

1.2.2 What the user sees

From the outside the system is almost boring, which is the point. A professor drops a slide deck onto a web page. A few minutes later a remediated version comes back: figures carry descriptions, slides carry titles, the reading order is set, the contrast fixed. An editor lets her review and correct any change before she downloads.

What she gets is more than the file. The result carries evidence: a record of every change the system made, and the clause of the standard each remaining finding answers to. A university does not just need its documents fixed. It needs to be able to show an auditor — or a courtroom — what was fixed, and to reverse any change that got the meaning wrong. The evidence is as much the product as the file is.

Concretely, the evidence is two lists. One is a changelog of every edit the system applied — each a typed operation, stamped with what made it and why. The other maps each finding the checker still reports to the clause of the standard it violates. Neither list is a paragraph of reassurance; both are records a machine produced and a machine can re-verify.

1.2.3 The shape of the machine

Inside, six parts carry that upload, drawn in Figure 1.2-1. A **front door** takes the file, checks who is asking, meters the quota, and queues the job. A **dispatcher** splits the document into chunks — slides, pages, sheets — so a hundred-slide deck finishes in about a minute instead of an hour. A **remediation core** repairs each chunk, walking the document through a structured model of its format. Then the trust half: a **checker** maps every remaining finding to a specific clause of the accessibility standard; a **fidelity validator** asserts the input's meaning survived into the output; a **provenance layer** stamps every inserted artifact, so any change can be explained and reversed.

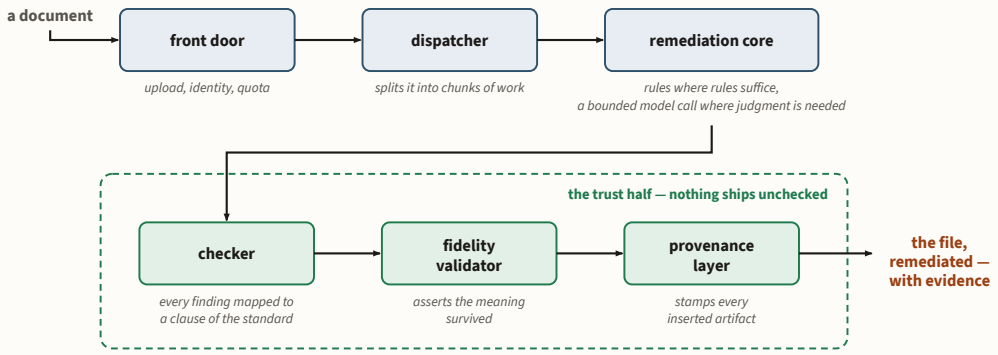


Figure 1.2-1: DocAble in one picture. The working half takes the document in — a front door, a dispatcher that splits it into chunks, a remediation core that repairs each. The trust half is why the result can be believed: a checker, a fidelity validator, a provenance stamp, and nothing ships unchecked.

The interesting joint is inside the core. Most repairs are a matter of rule, and ordinary deterministic code makes them. But some are a matter of judgment — describing a figure, reading an equation — and only a vision-capable model can make those. The system never hands the model the document. It hands the model one bounded task, takes back a typed candidate, and lets a deterministic validator pass or refuse it before anything touches the file. Deterministic code owns the workflow; the model is a subroutine inside it, never trusted on its word. The Built System develops this pattern in full.

Two commitments run underneath all six parts. Every document format is touched through one structured model, never the raw format library, so a fix applied once holds everywhere. And the pipeline fails loud: when a dependency is unreachable, the job stops rather than quietly shipping a degraded file. Both will matter shortly.

1.2.4 One human, a fleet

Now the part that makes this a book about engineering rather than accessibility: I did not write this system. A fleet of coding agents did, over about 20 weeks — 501,094 lines of production code, with six to eight agents running in parallel on a routine day and landing around 200 commits. I inspected almost none of it. My work was the other seats: deciding what to build, judging what came back, and building the environment that held the fleet to a standard I could no longer enforce by reading. In practice that judging had a rhythm — an agent proposes a phased design, surfaces the handful of decisions that genuinely need a human, I make those calls, and it executes.

That environment is why the repository looks upside down. The support apparatus — tests, lints, models, load-bearing documentation — runs about 3.0 times the size of the production code. At a hundred commits a day, no human reads the diffs; the apparatus is the review. That inversion is not a one-time snapshot but a curve that led the fleet from below parity to a threefold peak. For the data, see The Timeline and the Work → How it happened, week by week, is the story of the timeline.

1.2.5 Two scars

None of that machinery was designed in advance. It accreted, failure by failure. Two of those failures are worth meeting now — one architectural, one procedural — because they are the reader's first scars, and the book's whole method is visible in miniature in what happened next.

The library that lied

Early on, the codebase held two libraries that could touch a PDF: the sanctioned one, and a convenient second one an agent had adopted for quick page operations. The convenient one had a habit nobody had ordered: on write, it silently dropped the document's structure tree — the hidden hierarchy of headings, figures, and reading order that a screen reader walks. That tree is the product. A release shipped whose output opened fine, looked fine, and had been quietly made *less* accessible than its input. Nothing crashed, and no error was logged; the failure was invisible precisely where the product's promise lived. The fault was architectural: nothing in the system said which code was allowed to touch the format, so every code path could, and one did.

The lever that lost work

The second failure came from the process that built the product. The orchestrating agent — the one that dispatches the others and lands their work — occasionally hit a stuck merge. Git offers a fast way out: a destructive reset that abandons the mess by silently discarding the committed work already applied. Twice the orchestrator reached for that lever, and twice finished work vanished. A bare prohibition would not have fixed it: an agent forbidden the lever, with no named alternative, is simply stuck. The failure was in the environment, which offered a catastrophic escape and no sanctioned one.

1.2.6 The move that recurs

Neither fix stopped at the patch. The stripped structure tree became a **ban-lint**: all mutation of a document format now routes through one structured model, and a commit that reaches for the raw library anywhere else reddens on the spot. That is where the one-structured-model commitment from the machine section came from — a scar, healed into a wall. Not "we removed that call" — no agent, ever again, can make that class of mistake and get it into the tree. The lost commits became a paired rule, enforced at the hook level: the destructive lever is refused, and the rule names the sanctioned escape beside it, so a stuck agent has somewhere to go.

> Learn more about this governance mechanism: one structured model per format.

That conversion — a failure read as evidence of a missing mechanism, then encoded so the class cannot recur — is the single move this book teaches, made over and over until the environment could be trusted to run itself. Notice what it buys. My attention stopped being spent per change, which cannot scale, and started being spent once per class of failure, which can.

1.2.7 Where the map leads

You now hold the object: a real system, built by a fleet, governed by machinery that grew scar by scar. The rest of the book explains that machinery — the environment in Part 2, the models in Part 3, both as daily practice in Part 4 — and Part 5 reconstructs the full case, from the committee meeting where it started to the system that shipped. Part 5 is not offered as proof — one system, one engineer, one model vendor is a case, not a law. It is the grounding case: the system this method was derived from, and the place every claim in the coming chapters can be traced back to.

1.3 Loops and Models

Two ideas carry the rest of this book, and everything else is a consequence of them. The first is that all agent work is a *loop*. The second is that an agent steers that loop well only when you hand it a *model* of the world it is working in. This is what makes the MAGE method work: Model-Based Agentic Software Engineering means engineering that loop and handing the agent that model, deliberately, every time. Get these two right and the agent finds your answer fast. Get them wrong, it wanders.

1.3.1 Everything is a loop

An agent runs in a loop, and the loop has a fixed shape. It takes an input: the current problem, plus some notion of what success looks like. It reasons, and it acts in its operating environment. It produces a result. And that result folds back into the input for the next turn. Input, reasoning, output, fed around again. Shaping that loop is **commonly called loop engineering**, and nearly everything in this book is an instance of it. The close of this chapter draws the loop with the engineer's control points marked (Figure 1.3-2).

The loop that *produces* has this shape (Figure 1.3-1): a goal becomes a model, the agent builds, evidence meets judgment, and the model sharpens each pass.

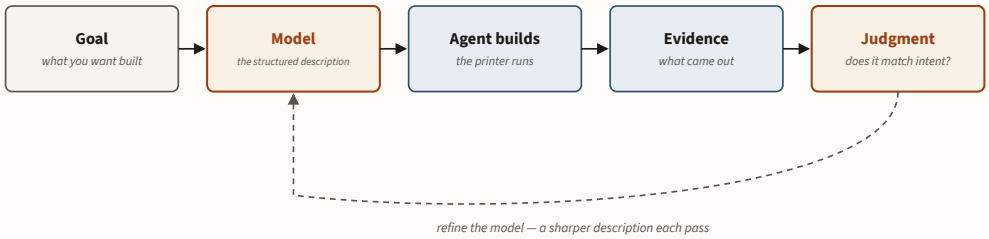


Figure 1.3-1: The Printer Loop. *The loop that produces: a goal becomes a model, the agent builds, evidence meets judgment, and the model sharpens each pass.*

Engineering a loop well starts with the metric, because the metric is what the agent steers by. The agent does not magically know your goals or your environment. You have to make its progress *measurable* — deterministically, through a quantitative check, or through a rubric that scores a qualitative result. Skip this, and you have told the agent nothing about what "done" means, so it searches, and searches, and searches, and never knows when to stop.

The orchestration layer of this book's companion repository is itself a worked example of a loop engineered this way. The catalogue models the orchestrator — the process that dispatches agents, waits for them to land work, and cleans up after them — as an explicit reactive loop with a typed lifecycle: *dispatch* → *work* → *land* → *tombstone*, then refill and go again. The loop is written down as a model the agent can read — which is the second idea.

Learn more about this governance mechanism: orchestrator model.

1.3.2 Give it a model of the world

> Vibe coding treats the program as an I/O device; engineering treats it as a system.

The difference is not aesthetic. It is what counts as proof that a change is good. Treat the program as an I/O device and the only question is whether the right thing came out this time. Treat it as a system and you ask what must stay true across every input, and you build the structure that holds the answer. Agents make both modes cheap, which is why you have to choose deliberately. The fast, seductive default is the I/O device, and the I/O device is what turns a hundred thousand lines into a mess nobody can reason over.

The second piece is the input the agent reasons *over*. You have to give it a sense of the environment: when it pulls this lever, what happens, and what is it even allowed to do? That sense of the world is itself an input, and the more of it you articulate, the better the agent knows what kind of reasoning the situation calls for.

Why does this work at all? It is easy to feel you are writing documentation into a void. But a foundation model is, loosely, a probabilistic reasoning machine. *Probabilistic* means it does not always do exactly what you want. *Reasoning machine* means it genuinely reasons — the interpretability work probing these networks keeps turning up real internal structure, and even lets us steer it. The Agent Stack makes that case in full. So write the model down: the agent reads it, and it reasons over it.

Think of the agent's behavior as a search. It is searching an enormous space: every place the answer might live, every configuration it might try, every function that might implement what you asked. That space is far too large to sweep.

So when you structure the loop and hand over a model, what you are really doing is **conditioning the probability distribution of the agent's behavior** — biasing it to search where the answer actually is. Done well, this takes the agent from failing your task to completing it. And when it was already completing the task, it now does so in fewer tokens and less time — materially fewer, in my experience — so you get back to the interesting parts of your job.

A model earns its keep twice over here. It aims the search, and it shrinks what the agent has to hold — which is what keeps the work from spilling the window and churning. The case study measures that churn directly: it peaks as the environment is built, then falls as the built environment holds. For the data, see The Timeline and the Work → (preliminary)

> **The metaphor's seam.** *Lakoff and Johnson, in* Metaphors We Live By *(1980), show that a structural metaphor illuminates where its source and target align and misleads where they break, so naming the break is part of honest use. Here is ours: a real 3D printer is deterministic — the same G-code yields the same part, every run. This printer is probabilistic — the same prompt can yield different output. That is exactly the gap the chapter turns on. Because the agent is a probabilistic search and not a deterministic extruder, we condition its distribution rather than instruct it. The metaphor claims "you get only what you describe"; it does not claim "you get the identical thing twice."*

1.3.3 The four levers of an engineered loop

Now put the two ideas back together, because they are one machine. A loop alone is generic; a thermostat runs one. What makes agent work engineerable is that the environment owns the loop's control points, and you set every one of them.

> **Engineered agent loop.** *A feedback process in which the environment supplies a modeled state, constrains available actions, measures progress against an explicit criterion, and determines whether the result may advance.*

Read the definition as a list of levers, because that is what it is. Figure 1.3-2 draws them at their stations.

- **The state exposed.** What the agent sees is not the world; it is the state you supply — and this chapter just argued it should be a *model*.
- **The action surface.** What the agent may do is not everything; it is the tools and sanctioned seams the environment offers, so the wrong move is unavailable.
- **The success criterion.** What counts as progress is not a feeling of doneness; it is the metric or rubric you made measurable above.
- **The admission rule.** Whether the result advances is not the agent's call; it is the gate its output must pass before the work feeds the next turn.

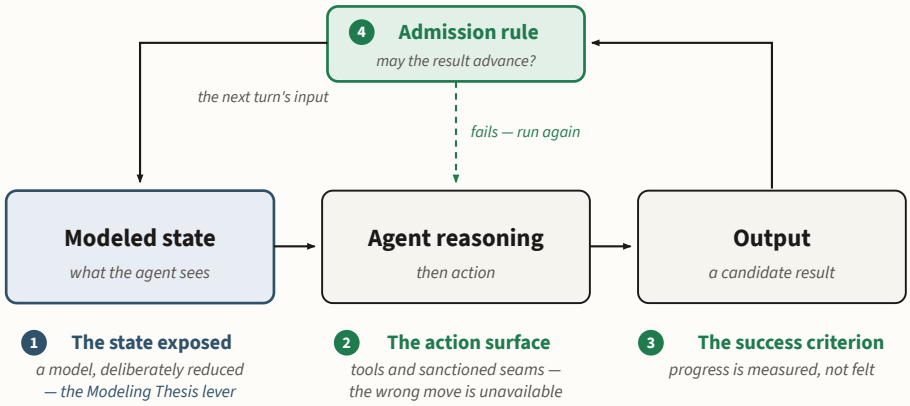


Figure 1.3-2: The Four Levers. *The engineered agent loop — modeled state in, reasoning and action, a candidate output, fed around again. The four numbered levers are the engineering: the state exposed, the action surface, the success criterion, and the admission rule that decides whether the result advances.*

The four levers are the book's two theses in miniature. The first — a modeled state — is the Modeling Thesis: bind intent and structure into an explicit model, and the agent reasons through it while the engineer specifies, analyzes, and predicts on it. The other three — constrained actions, an explicit criterion, an admission rule — are the Alignment Thesis: mechanisms the environment enforces, holding the output to intent. Part 2 builds the second set; Part 3 builds the first.

The preface named churn as the wall a fleet hits and traced it to three not-knowings. The levers are where those causes get treated. Not knowing *what to build* and not knowing *how to realize it* are failures of the exposed state; the modeled state answers both. Not knowing *how to change the system*

safely is a failure of the other three levers — an unconstrained action, an unmeasured criterion, an ungated admission. That is the division of labor the book runs on: modeling treats the first two not-knowings, alignment the third.

1.4 Why MAGE Follows from the Machine

Reinforced concrete carries a building because two materials split the work: concrete takes the compression, steel takes the tension, and the method — where the bars run, how the loads route — follows from those properties. Every engineering discipline is read off its material this way. MAGE is not classical software engineering with agents substituted for programmers; its shape follows from the machinery.

The machinery is itself a pairing. A foundation model is a broad but probabilistic reasoner: it can interpret abstractions and search open-ended spaces, yet it works through a bounded context, rebuilds its picture of your system afresh each task, and cannot certify its own output. An agentic harness gives that reasoner tools, a lifecycle, and points where action can be observed or blocked. Together they form a new **engineering substrate** — capable enough to reason through explicit models, cheap enough to maintain those models continuously, and controllable enough that policy can live in the environment rather than in the agent's promises.

The last chapter ended with the two theses stated. This chapter shows they are not chosen; they are derived. First the reasoner's properties, then the harness's, then the engineering each one forces.

1.4.1 The reasoner's properties

Loops and Models called the foundation model a probabilistic reasoning machine. Take that machine apart. Four of its properties do the deriving.

- **Broad semantic reasoning.** It interprets, synthesizes, and searches across open-ended domains. Hand it an abstraction — a schema, a state table, a policy — and it operates over the abstraction, not just a fixed script.
- **Probabilistic execution.** The same input does not yield the same behavior twice, and the model's confidence is not evidence. It cannot certify its own output.
- **Bounded, reconstructed working state.** The context window is fixed, and no durable understanding of your system persists from task to task. Every task starts cold; the model rebuilds its picture of the system, every time.
- **Cheap repeated cognition.** Search, generation, repair, reconciliation — cognitive work that once cost an engineer's week now runs at a scale that was formerly prohibitive.

Notice the tension. The first and fourth properties are the promise: a reasoner that works at any level of abstraction, for cents. The second and third are the catch: it cannot be trusted to certify itself, and it cannot remember your system. An engineering method for this material has to spend the promise against the catch.

1.4.2 What the harness adds

A raw model only recommends. The harness — the runtime around it — turns recommendation into action, and it brings four properties of its own.

- **Tool-mediated action.** The agent acts on the world only through identifiable interfaces: a file edit, a command, an API call. There are no bare hands.

- **Interposition.** Because every action crosses an interface, a hook or wrapper can inspect it, constrain it, redirect it, or deny it at the moment it happens.
- **Lifecycle visibility.** Dispatch, tool calls, compaction, completion, merge, deploy — the work's transitions are machine-observable events, not private states in a worker's head.
- **Parallel execution.** Many independent reasoning loops run at once, on one codebase, around the clock.

Read these as control points. Tool mediation and interposition are where authority lives; lifecycle visibility is where observation lives; parallelism is what raises the stakes on both. Part 2 walks the stack these properties live in, layer by layer.

MAGE does not derive this substrate in a vacuum. A 2026 literature on *harness engineering* has reached the same reading. Autonomous capability is a property of a model–harness–environment system, not of the model alone[6]. The harness is the runtime an engineer designs and tunes, enough of a variable that recent work optimizes it automatically[7]. Its craft is as much removal as construction: every harness component encodes an assumption about what the model cannot yet do, and that assumption expires as models improve[8]. What lasts is the durable substrate under the deletable scaffolding, a rule one team states as thin harness, fat skills[9]. Independent evidence sharpens what in a harness lasts. Lin and colleagues evolve harness components automatically from observed agent trajectories, and their ablations localize the gain to tools, middleware, and long-term memory rather than the system prompt. Factual harness structure transfers; prose-level strategy does not[10]. MAGE stands on this literature and supplies what it leaves open: the engineering method the substrate implies, once the environment the agent acts in is itself governed.

1.4.3 The engineering that follows

Cross the two lists and the method starts writing itself. Broad reasoning means an agent can work from a model instead of a rigid script. A bounded, reconstructed context means that model must be compact and structured, or it will not fit the window that has to hold it. Probabilistic execution means action and admission need deterministic envelopes — checks that do not care how confident the model was. And tool-mediated action means those envelopes can be enforced at real authority boundaries, not requested in a prompt.

The economics finish the job. Cheap repeated cognition means the models, tests, projections, and drift checks can be maintained continuously; the upkeep that killed the modeling tradition becomes a background task. Parallel velocity means human per-change oversight saturates; attention has to move from changes to classes of change. And cold task starts mean policy cannot live in remembered training or managerial expectation. It has to live in the environment, where every fresh context finds it.

---

[6] Hailin Zhong and Shengxin Zhu, "AI Harness Engineering: A Runtime Substrate for Foundation-Model Software Agents," 2026, https://arxiv.org/abs/2605.13357.

[7] Yoonho Lee et al., "Meta-Harness: End-to-End Optimization of Model Harnesses," 2026, https://arxiv.org/abs/2603.28052.

[8] Justin Young, "Effective Harnesses for Long-Running Agents," Anthropic, November 26, 2025, https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents.

[9] Garry Tan, "Thin Harness, Fat Skills," April 9, 2026, https://github.com/garrytan/gbrain/blob/master/docs/ethos/THIN_HARNESS_FAT_SKILLS.md.

[10] Jiahang Lin et al., "Agentic Harness Engineering: Observability-Driven Automatic Evolution of Coding-Agent Harnesses," 2026, https://arxiv.org/abs/2604.25850.

Table 1.4-1 lays out the whole mapping: each property of the substrate, the problem or opportunity it creates, and the response the method gives it. The last row names a hazard the properties compose to produce: a broad reasoner working probabilistically can make a change that looks right while violating a global property it never saw.

Table 1.4-1: Foundation models make model-based reasoning useful; harnesses make environmental enforcement possible; their economics make both scalable.

Property of the new substrate	Resulting problem / opportunity	MAGE response
Broad semantic reasoning	Can operate over abstractions, not just fixed scripts	Structured models as working representations
Probabilistic generation	Instructions + self-reports can't establish correctness	External validation + admission gates
Bounded working context	Raw systems exceed what can be held coherently	Semantic compression through models
Cold task starts	Organizational memory doesn't naturally persist	Authoritative, machine-readable environment
Tool-mediated action	Behavior crosses identifiable control points	Sanctioned surfaces, wrappers, hooks, permissions
Cheap repeated labor	Maintenance formerly too costly becomes feasible	Continuous generation, reconciliation, testing, repair
High throughput + parallelism	Human review + tacit conventions saturate	Judgment amortized into durable mechanisms
Observable lifecycle	Work has machine-visible transitions	Lifecycle controls, provenance, orchestration, recovery
Locally plausible failure	A change can look right while violating global properties	Invariants, composed models, end-to-end + preservation checks

1.4.4 The theses follow

Now collect the derivation. Because the reasoner is broad, it can work through an explicit model. Because it is context-bounded, the model is how the system fits. Because reconciliation is cheap, the model can be kept true. That is the Modeling Thesis, read off three properties of the machine.

Because its behavior is probabilistic, correctness needs external mechanisms. Because its actions cross controllable boundaries, those mechanisms have somewhere real to stand. And because the resulting workforce moves faster than human attention, each recurring failure must become a durable property of the environment rather than one more thing to watch for. That is the Alignment Thesis.

The two derivations land in one place: an environment that carries the models the fleet reasons through and enforces the mechanisms that hold its work to intent — the governed engineering environment. Figure 1.4-1 draws the derivation end to end.

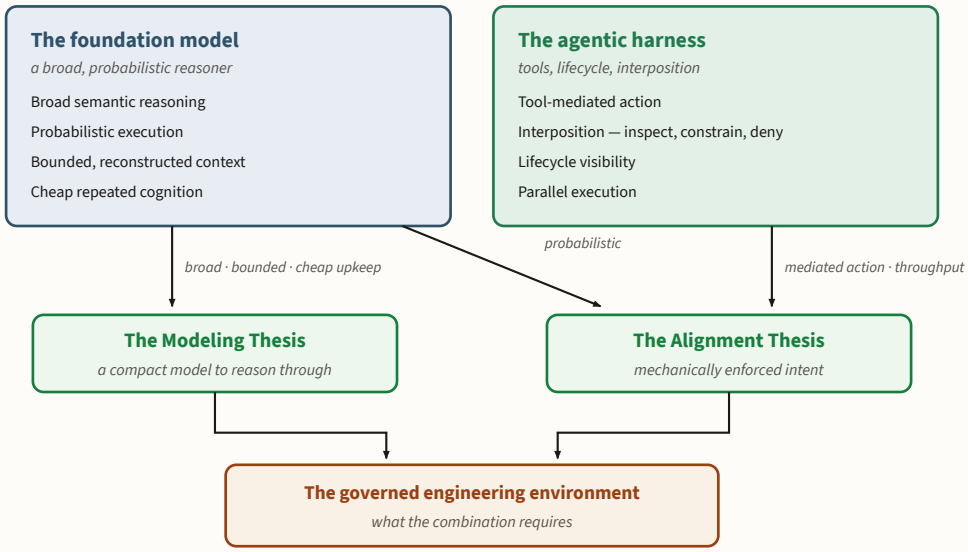


Figure 1.4-1: The Derivation. *Four foundation-model properties and four harness properties feed the two theses — broad reasoning, bounded context, and cheap upkeep yield the Modeling Thesis; probabilistic execution, tool mediation, and parallel throughput yield the Alignment Thesis. Read the arrows as "therefore": their combination requires the governed environment.*

Nothing above leaned on an analogy to a human organization. The derivation runs from properties of a machine to the method they force, the way reinforced concrete's method runs from compression and tension. When the substrate's properties shift, with longer windows or new interposition points, the derivation is what you re-run. And what it yields is an implication, not a proof: whether the implied method holds up under a real system is a question for evidence, and the case study at the back of the book supplies one system's worth — a grounding case.

1.5 The Engineer's Seat

Older engineering fields draw a clean line between the engineer who designs and the technician who builds. Software spent decades trying to draw the same line — an engineer class over a developer class — and the complexity of system-building defeated it every time. Knowing a system well enough to design it kept requiring you to build it, and building it kept requiring the judgment that was supposed to sit a rank away. The two seats kept collapsing back into one person. What is new is not a new phase or a new artifact. It is that **agents finally hold the two roles apart**: the fleet takes the developer's seat, and the human keeps the seats that were always the engineering. That is the whole of the change — one seat reassigned, the lifecycle otherwise intact. This chapter draws that lifecycle for the reader who has never worked it, and states the claim the rest of the book stands on: MAGE does not remove the engineering lifecycle; it changes the role of the engineer within it.

1.5.1 The lifecycle it keeps

Engineering ran on the same handful of activities long before software, and software inherits them whole: **requirements** (find out what the system must do, and for whom — usually the hard part, since the people who need it cannot state it), **specification** (say what it will do, precisely enough to check later), **design** (decide the parts, the seams, and how the whole meets its constraints), **implementation** (build it — for software, write the code), **validation** (show the built thing meets the spec), and **maintenance** (change it over the long life without breaking what already works). The reader knows this list; it is named here only so the seat that moves has a chair to move from.

What makes *software's* version its own is worth one line. A program answers to far fewer physical constraints than a bridge, copies for nothing once written, and stays changeable long after release — where most of its life is spent. Looser physics, free copies, changeability under maintenance: the same six activities, played out differently. Each has a literature of its own — Sommerville's *Software Engineering*, Pressman and Maxim's *A Practitioner's Approach* — and this book assumes only the sketch, not the depth.

The process models differ only in how they order these activities — waterfall once through, front to back; iterative processes in loops days or hours wide. The activities are the same either way, and this book takes no position on the ordering. What it takes a position on is who sits where.

> *A footnote on "waterfall."* The pure, once-through waterfall is usually credited to Winston Royce's 1970 paper, "Managing the Development of Large Software Systems." The credit is backwards. Royce drew the strict sequential model in order to argue *against* it — he called it flawed, said no sane organization would build that way, and called for cycles, for iteration between the steps. His cycles were simply years long, sized to the Department-of-Defense projects of the day, whose scope still runs to years now. The strawman outlived the argument that built it.

One thing the six activities never name: improving the process itself. Continuous improvement is also part of engineering — you optimize the product, and you optimize the process that creates it. Manufacturing folded it into the line, and Toyota's production system gave the moves their names. *Poka-yoke* is mistake-proofing: shape the work so the defect cannot happen, the way a fixture refuses a part mounted backwards. *Jidoka* stops the line the instant a defect appears, so no downstream work builds on it. And *kaizen* is the habit behind both: every recurring problem becomes a permanent

countermeasure. The lifecycle says how the work runs; kaizen is what you do when the same failure surfaces twice. All three return in Part 2 as the book's own machinery, and kaizen is MAGE's actual method — the improvement loop, run daily on the lifecycle itself.

1.5.2 The two seats

Two words for two jobs, and this book keeps them apart. The **engineer** authors the intent, chooses the structure, and judges the result — requirements, specification, design, validation. The **developer** implements it: writes the code that realizes the design. Of the six activities, one dominated the calendar — implementation. Writing the code was the expensive part, so the discipline arranged itself around the writing, and the seat that ate the calendar defined the job. How a shop divides that labor is its own affair — a lone founder may hold both; a large organization splits them across teams, titles, and contractors. The book prescribes none of it; it needs the two words only to name the seats.

MAGE changes exactly one thing in that picture. The developer's seat goes to the agents. The fleet writes the code, and the human keeps the seats that were always the engineering: stating what to build — requirements, specification, design, authored as models the fleet can reason over — judging what comes back, and holding the whole to a standard the fleet cannot lower.

Do not read this as the engineer doing less. The preface said code got cheap and judgment did not; here is where that lands. Every hour the fleet saves on implementation is throughput pointed wherever your design points it, and a wrong design now gets built out at full speed. The judgment seats were always the consequential ones. Now they are the job.

Early evidence is starting to describe this shift, not just predict it. A six-month longitudinal study of engineers adopting AI assistants reports most of them writing less code and spending more time directing, checking, and correcting what the model produced — a mode the authors name *supervisory engineering work*. It does not come free: flow and cognitive load both eroded even as output held[11]. Software-design education is testing the same move, teaching students to treat a model's design output as a candidate to critique before adopting, not an answer to accept[12]. MAGE gives that supervisory category a concrete technical shape. The engineer authors the models the fleet reasons through, chooses the properties the environment must enforce, designs the evidence that a change is done, and decides whether a failure is a local bug or a missing mechanism. None of this makes coding a lost skill, and it does not turn every engineer into an architect. Implementation moves toward the fleet; accountability does not.

Authoring intent as models is not a new duty. Engineers always had models; they did not always write them down. When agile embraced the uncertainty of requirements, a written model went stale faster than it paid, and the discipline chose delivery over discernibility. The model never left; only the writing-down stopped paying — and a fleet that executes from the model makes it pay again.

1.5.3 SDLC to SELC

The field's own name for the lifecycle gives the old arrangement away: SDLC, the software *development* life cycle. The name centers development because development was the bottleneck; you name

[11] Annie Vella and Kelly Blincoe, "The Impact of AI Coding Assistants on Software Engineering: A Longitudinal Study," 2026, https://arxiv.org/abs/2605.23135.

[12] Victoria Jackson et al., "Using Generative AI in Software Design Education: An Experience Report," 2025, https://arxiv.org/abs/2506.21703.

the cycle after the seat where the calendar goes. Move that seat to the fleet and the name is pointing at a chair the human has left.

So the rename follows on its own: **SELC**, the software *engineering* life cycle. The D is delegated — development belongs to the fleet now, and a name should not center a seat the human has left. The E is what remains, and it was always the point: stating intent precisely, structuring the system, deciding what proof a change owes, and holding the line. SELC names the job that is left, which is the job this book teaches. Figure 1.5-1 draws the two arrangements.

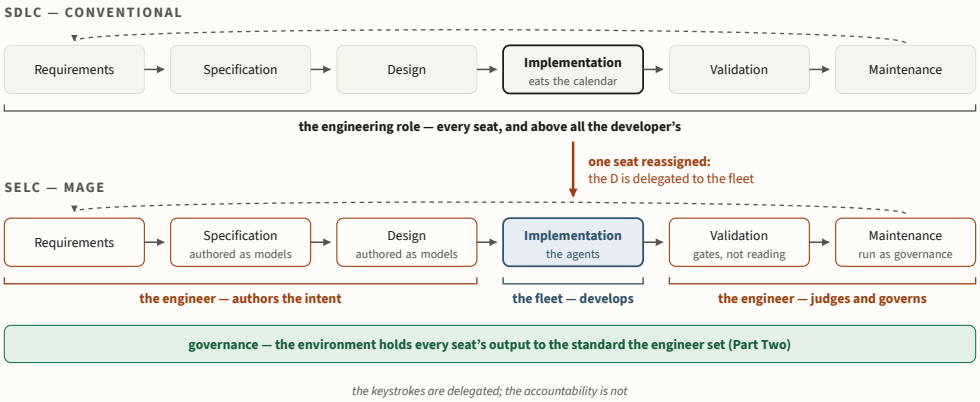


Figure 1.5-1: The Reassigned Seat. *The conventional SDLC gives every seat to an engineering role; the SELC reassigns the developer's seat to the agent fleet. The engineer flanks the fleet — intent flows in as models, judgment and governance meet the output, and a governance band holds every seat's output.*

Delegating the keystrokes delegates nothing else. The engineer who no longer types the code is still answerable for it, the way an engineer was always answerable for what a chosen tool produced. The back matter returns to what that responsibility becomes (Implications for Software Engineering); for now it is enough that the seat moved and the accountability did not.

1.5.4 A different seat is a different discipline

Same lifecycle, then; no new phases. But the reassignment still remakes the discipline, because the intent now has to be legible to a probabilistic machine. A specification a human developer could interpret charitably must now be precise enough for that machine to execute — which is why this book authors spec and design as structured models (The Model Zoo). Validation at fleet velocity outruns any human reader, so it moves into the environment as gates the fleet cannot skip. And maintenance under a workforce with no memory is, I will argue, best run as governance: rules that hold because the environment enforces them, not because anyone remembers them. Whether large organizations keep their teams, their titles, and their handoffs through this shift, the book does not say — nobody knows yet.

Reaching for models is also an old move, made once before. The field left assembly for high-level languages the moment a machine could execute the higher description. We always had models; the open question was how far into development they had to be carried — how much of the blueprint you still had to turn into code by hand — before a machine could run it. Compilers moved that line once. MAGE moves it again: the blueprint is all the machine needs. But it does need the blueprint. A fleet can execute from a structured model; it cannot execute from a vibe. Intent left unwritten, or written loosely, gives the machine nothing to build from and the engineer nothing to hold it to.

That is the map of the rest of the book: it equips the engineer's seat, phase by phase. And it starts where the loss of the developer's seat bites first. The engineer's grip on implementation is now indirect — you govern the machine that writes the code, not the code itself. So where do you reach in to do it? That is the question the next part opens on.

The Printer Commandments

Part 1's method, in five rules to carry into the rest of the book:

1. **Treat the agent as a printer, not a coder.** It builds exactly what your instructions describe. Read a bad build as evidence about the instructions first, before you blame the machine.
2. **Engineer the loop.** All agent work runs in a loop. Set its four levers: the state you expose, the actions you allow, the metric it steers by, and the gate that admits its output.
3. **Hand the fleet a model of the world.** Give the agent a compact model to reason through, not the raw codebase. It aims the search and keeps the work inside the context window.
4. **Govern with mechanisms, not attention.** Encode each obligation as a constraint or a sensor the environment enforces, so a policy decided once holds against every later change.
5. **Govern; do not type.** The developer's seat goes to the fleet. Keep the engineer's — intent, design, judgment, and the line. SDLC becomes SELC: the keystrokes move, the accountability does not.

Part 2

The Governed Engineering Environment

2.1 The Agent Stack

A language model alone gives probabilistic recommendations. A harness turns some of them into actions. A governed engineering environment interposes between capability and consequence: it decides which recommendations become actions, and which actions become the system. Part 1 derived why the substrate demands that interposition; this Part builds it.

The building material comes in three kinds. **Soft conditioning** aims the reasoner — prompts, briefs, models, examples, playbooks. **Hard authority** binds it — tool interfaces, permissions, schemas, lints, gates, validators. **Feedback** closes the loop — results made machine-legible, so the agent can repair its own miss. In one line: soft guidance goes in, deterministic guardrails constrain the actions, and a deterministic envelope governs what gets admitted out.

If governing an agent means conditioning where it searches, the practical question follows at once: *where do you reach in to do it?* There are four places, stacked one atop another, and each gives you a different kind of grip. Once you understand the stack, every mechanism in this book resolves into the same move: pick a layer, inject a constraint. Figure 2.1-1 draws the stack; this chapter walks it from the bottom up.

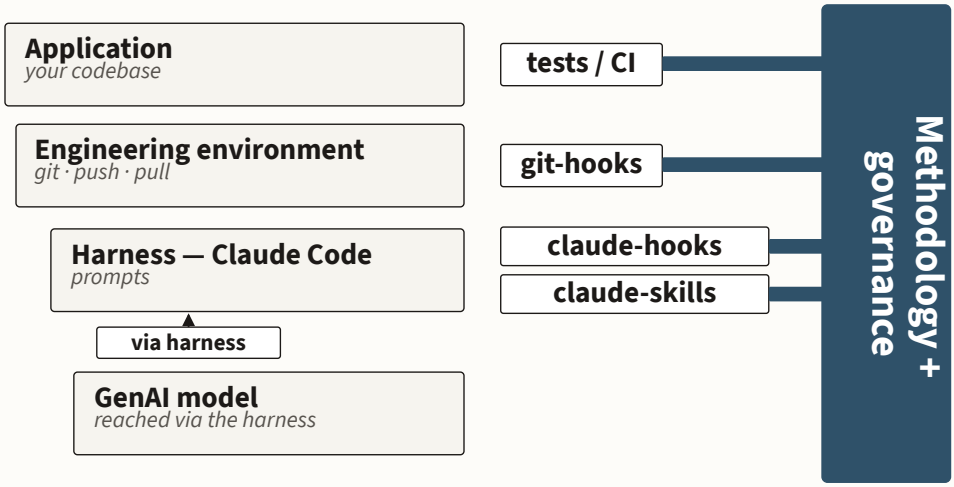


Figure 2.1-1: The MAGE Stack. *From the top: the Application (codebase, tests, CI), the Engineering environment (git and its hooks), the Harness (the agent's runtime), and the GenAI model at the foundation. The governance bar notches into every layer — a posture at all four levels, not one end gate.*

2.1.1 First, four definitions

Before the walk, I owe you four definitions — *engineering*, *software engineering*, *agent*, and *model* — and, before we leave them, one adjective the last of them will come to lean on: *structured*. Ordinary words, all of them, and every reader arrives with a working sense of each. But this book derives its argument from these definitions and the relations between them, so a working sense will not carry the weight. Let us fix them now. The stack will keep.

> **Engineering.** *The discipline of developing operable technological artifacts to solve problems, and of analytically identifying and assessing the competing means to those solutions, each with distinct trade-offs.*

Take the definition in halves. The first half says engineering produces something that *works*: a technological artifact, with human operators, addressing a real problem. The second half separates the engineer from the technician: if you cannot articulate the trade-offs, you are not engineering — you are a technician. An engineer weighs competing architectures and designs against the properties the solution must have.

A third property hides inside "analytically assessing": an engineer can model and predict the outcomes of the decisions they are about to make, from the architecture and design they propose, *prior to building*. Prediction-before-construction is the engineering move. Hold onto it, because the definition of *model* below exists to serve it.

Software engineering. *Software development produces a functioning artifact. Software engineering* predictively selects *a software artifact according to its desired properties.*

Development asks: does it function, and does it meet the requirements? Engineering asks: of the artifacts that would function, which one ought we build, given the properties we want? — and answers *before* building it. That is why this is a book about software *engineering* and not software development. And agents sharpen the distinction rather than blunting it: when implementation is cheap, you can afford to build more than one — in Fred Brooks's words, to build one to throw away — and to select among them. Selection needs a basis for prediction, and the basis is a model.

Agent. *A controllable intelligence capable of independent reasoning over a knowledge base.*

Intelligence I will not attempt to define in full; a working sense will do. Given a set of constraints and a goal, an intelligence can devise a *means* of satisfying the goal without violating the constraints. Think of a raven dropping pebbles into a jar to raise the water high enough to drink. Goal: a drink. Constraint: the beak cannot reach the bottom. Means: pebbles. A raven Archimedes. *Eureka.*

Controllable is the adjective everything else in this book hangs on, and it does not say *controlled*. A controllable intelligence responds to its handler: change its environment and the knowledge available to it and it does different things, and it will generally follow orders. But *controlled* means perfect guarantees of behavior, where *controllable* means probabilistic ones. The stronger your mechanisms, the tighter the distribution — never a guarantee. That one adjective is why governance takes up so much of this book. If agents were controlled, we would not need the governance at all.

Independent reasoning rules out the parlor trick. The agent genuinely reasons in response to a query; it is not retrieving stored answers, and it is not pattern-matching its way to something that merely looks like one. (The next section takes up the evidence for this claim, and the evidence is stronger than you might expect.) Finally, the *knowledge base* names the thing it reasons over. For this book, that clause is the hinge: the models are the knowledge base.

> *A footnote for the reader with a textbook nearby.* This definition departs from Russell and Norvig deliberately. For them, an agent is anything that *perceives* its environment through sensors and *acts* on it through actuators, and a rational agent acts to maximize an expected performance measure. That perceive-and-act framing is the right one for *building* an agent. This book is about *governing* one, so it foregrounds a different property: controllability, in the cybernetic sense of behavior you can steer but not perfectly determine. The gap between controllable and controlled is no hedge. It is the reason this book exists.

> **Model, first pass.** *A useful approximation of a phenomenon — simplified, but good enough to predict — and, as this book uses the word, a* blueprint*: a prescriptive description of a system to be built, which the built system is bound to realize. The formal definition arrives once its last ingredient,* structured*, is on the table.*

The reader likely knows the first sense already: a model as a simplified description, prized because it *predicts*, not because it is exact. Think $f = ma$, which neglects friction and is useful anyway; George Box's aphorism — all models are wrong, some are useful — is the canonical statement. Models in this

sense come in kinds: mathematical, qualitative, relational, data-flow. And their fidelity trades against cost: the coarser the model, the cheaper it is to reason about, and yet the wider the error bars on what the reasoning tells you. How much fidelity you need is in the eye of the application. "It processes information serially" may do to sketch a system, while reasoning about security or privacy wants a model much closer to the real thing.

The second sense is the one I embed here. A blueprint does not *approximate* an existing building; it *specifies* one that does not yet exist, and the builder is bound to it. That is the sense this book leans on: a model as a *prescriptive* description of the system you intend to build. A system built to a model may *elaborate* on the model, but it may not *diverge* from it. (The approximation sense is Box's. The blueprint sense — a description that specifies rather than describes — is what semiotics calls signification, a sign standing for its object; *blueprint* is the engineer's word for it, and the one I use.)

The blueprint sense is what lets a model serve as an agent's marching orders. Handed to an agent, a model is three things at once: a *constraint*, bounding what may be built; a *blueprint*, saying what to build; and, because the agent reasons over it, the *knowledge base* for the artifact under construction.

One caution before moving on. In these pages the word does double duty: the *foundation model* at the bottom of the stack is itself a model in the first sense, a learned approximation of language and the world, while the models this book teaches you to write are blueprints in the second. Context will say which is meant.

And one adjective before we go, because every blueprint in this book carries it: *structured*.

> **Structured.** *Said of a model: written in an explicit, declared shape a machine can read and validate — a schema, not prose. The declared structure is what lets a machine hold the system up to the model: read the shape, check it for drift, query it, reason over it, enforce it. Prose can be a model; only a structured model can be checked.*

The word is broader than a programming manual's *typed*. A variable has a type; that is the narrow sense, and type-checking is its sharpest case. The sense this book strains on is wider: the model has a shape, the shape is written down — fields, states, relations, all explicit — and a checker can validate any instance against it. *Structured* is the word I use, over *typed*, because it names that whole span. A type is one way to declare a shape; a schema, a state table, or a registry will do as well. Free prose fails the test not because it is vague but because a machine cannot act on it: it can pattern-match the words, but it cannot hold the code up to them and prove the two agree.

Watch what the adjective carries. Prose can be a model — simplified, predictive, even prescriptive; the first half of the model definition asks no more. But every promise this book makes about models — that they can be read, drift-checked, queried, reasoned over, enforced, that a gate stays red until the map tells the truth again — is a promise about *structured* models. The structure is what admits the machine. Drop the adjective and the drift gate has nothing to read; keep it and the model stops being documentation and becomes something the build can act on.

The pieces are now on the table — a model approximates, a blueprint prescribes, structure admits the machine — and they assemble into the definition the rest of this book holds a candidate model to. The book will call many things models: registries, graphs, manifests, state machines, typed records, topology descriptions. The word needs a boundary, or everything structured gets the name and the claim dissolves. Here is the boundary.

> **Model.** *A deliberately reduced, machine-readable representation of system intent, structure, behavior, policy, or evidence that supports reasoning, derivation, or checking, and whose relationship to the implemented or observed system is mechanically maintained.*

Every clause earns its place. *Deliberately reduced* excludes the codebase itself; the territory is not its own map. *Machine-readable* excludes ordinary prose. *Of intent, structure, behavior, policy, or evidence* names what is represented, so a data structure with no referent does not qualify. *Supports reasoning, derivation, or checking* demands the model do work. And *mechanically maintained* is the sharpest gate: a diagram nobody checks fails it the day the code moves. The next chapter builds that last clause into a discipline, and Part 3 admits a whole zoo of representations because each clears all five.

Now hold the four together, because the relations between them tell you the whole book in a nutshell. Engineers love models: prediction without excess cost is their bread and butter. An agent is a controllable reasoner in want of a knowledge base, and a model is a knowledge base that constrains and specifies. Hand one to the other and you have the method. The book's job, then, is fourfold: to describe the kinds of models useful to agents; to show how agents interact with them; to keep the models consistent with the artifact as it is built; and to help engineers construct models, assess them, and use them to predict prior to construction. So much for definitions. Now, the stack.

2.1.2 The four layers

At the bottom sits the foundation model. It comes from a commercial vendor or an open-weight release, and it is a general language reasoner, able to *induce* an understanding of your environment — though that induction is expensive, and it happens fresh every time. The game at this layer, then, is to spare the model that cost: tell it, up front, where to look. Sometimes you tell it deterministically, with an executable program; sometimes softly, with written-down pseudocode — the algorithm to follow, the playbook for a root-cause analysis; often it is a hybrid of the two. The more reasoning you hand the model up front, the shorter the chain it must construct on its own — and the shorter the chain, the less likely it is to go wrong along the way. If each step goes right with probability $1 - p$, an n-step chain reaches the end intact only with probability $(1 - p)^n$, so the chance of a wrong answer climbs with every step you did not spare it:

$$P(\text{wrong}) = 1 - (1 - p)^n$$

Every stretch of reasoning you hand the model, rather than making it re-derive, is one fewer step in n the error probability falls with it.

What you control in the model. The foundation model has three parts, and you steer each one differently.

- **Its reasoning is a grade you buy.** Pick the tier, and size each task to a model that can reach the end of its chain — or let the orchestrator route the task to the model that fits.
- **Its training data you do not touch.** This book assumes you run the model as it ships. Bending the weights to your domain is a real lever, but a resource-intensive one, and in the main it is much more cost-effective to optimize the context window and revisit at the next generation of foundation models. (A later chapter treats training data as the novelty axis: how far your system sits from what the model has seen decides how hard you must condition it.)
- **Its context window is fixed the moment you choose the model.** You cannot buy more of it. What you can change is how much meaning each token carries.

So the window bounds the fleet — but not the way it first looks. You cannot widen it; you can pack it denser. A model is a higher-abstraction representation, more meaning per token, so the same window holds more of the system. That is the Modeling Thesis (derived in *Why MAGE Follows from the Machine*) aimed straight at the window bound: the fleet scales on two capacities, its reasoning grade and its window.

To the extent you control model choice, make it; if you have the resources to tune, it may help. But this book's position is that controlling the context window is the key, and modeling is the path we take.

This whole layer rests on one claim: the model reasons. It is not retrieving stored answers or matching surface patterns — it builds an internal model of the world it is working in and reasons over it. Think of a model trained only on the move sequences of the board game Othello. Nowhere in its training does it see a board; it sees strings of moves and nothing else. Yet interpretability work found a recoverable representation of the board state living inside the network. More telling still, the model does not merely *read out* that representation; it *causally intervenes* on it. Flip the encoded state of a square and the model's predictions change to match the altered board, exactly as they would if you had made a real move. The representation drives the behavior. This is the Othello-GPT "emergent world representations" work[13], and it is one example, not a lone toy: follow-up work generalized the finding — probing *and* steering learned world-models — across setting after setting, and it agrees with what anyone who works with these models day to day already sees. So we treat the model as a reasoner, because it is one, and the job at this layer is to equip it to reason *well* — to supply the context it needs so its reasoning starts from your world instead of a guessed one. Equipping it is a concrete act: you package that context as a skill, a body of knowledge and judgment the model reaches for. Appendix E gives the recipe for building one.

On top of the model sits the agentic harness — Claude Code, OpenCode, and their kin. The harness takes your prompts, feeds them to the model, and manages the context window: the fixed budget of tokens the model can hold. Exceed the budget and the harness must compact — a lossy compression of everything that has happened so far. The harness is also where two of the most useful mechanism points live, and the line between them is sharp: a hook is a thing the agent *has* to do; a skill is a thing it *can choose* to do.

The field has begun to name this layer with some care. A working taxonomy separates the *framework* that composes agents and tools from the *harness*, the full runtime around an agent — context, memory, permissions, failure recovery, verification — and from a broader *agent OS* that schedules and allocates across the whole stack[14]. The analogy that keeps returning is exactly that: the harness is the operating system, supplying the interrupts and interfaces to the outside world and managing the agent's processes and memory[15]. Meta-harnesses now sit above several such runtimes and orchestrate across them[16]. This book says *harness* for the runtime layer throughout, and treats the agent OS as the same idea drawn wider.

---

[13] Kenneth Li et al., "Emergent World Representations: Exploring a Sequence Model Trained on a Synthetic Task," 2022, https://arxiv.org/abs/2210.13382.

[14] Gurbinder Gill, "The Rise of the Agent OS: Comparing Harnesses, Runtimes, And Orchestration Layers for LLM Agents," LinkedIn, July 15, 2026, https://www.linkedin.com/pulse/rise-agent-os-comparing-harnesses-runtimes-layers-llm-gurbinder-gill-72c9c/; Aishwarya Naresh Reganti, "The AI Agent Stack in 2026," The Nuanced Perspective, April 29, 2026, https://thenuancedperspective.substack.com/p/the-ai-agent-stack-in-2026.

[15] Han Lee, "Hidden Technical Debt of AI Systems: Agent Harness," May 8, 2026, https://leehanchung.github.io/blogs/2026/05/08/hidden-technical-debt-agent-harness/.

[16] Matei Zaharia et al., "Introducing Omnigent: A Meta-Harness to Combine, Control and Share Your Agents," Databricks, June 13, 2026, https://www.databricks.com/blog/introducing-omnigent-meta-harness-combine-control-and-share-your-agents.

- **Skills are soft mechanisms.** A skill has trigger criteria and a body of guidance: "about to touch a PDF? enter this skill." Inside, it holds instructions — pure prose ("be careful editing PDFs"), or a mix of prose and code. The code half is a **tool**: a library or command-line entry point that lets the model take a *deterministic* action — add a page, delete a page, rewrite a run of text — instead of guessing.
- **Hooks are hard mechanisms.** Before or after the harness does something — passing a prompt, dispatching a sub-agent, calling a tool — you can run a deterministic program that inspects the action and, if it likes, *forbids* it. A hook can look at an outbound network call and refuse it; catch an email about to go out with a typo, or with sensitive information in it, and block the send. Run after the fact, the same hook logs what happened. Either way, you hold authoritative control over the agent's behavior at the moment it acts.

Above the harness sits the engineering environment — the tools and conventions your team already uses. Jira holds the tickets. Git holds the code. A named server is where you deploy; a named node is the head of your compute cluster. Engineers carry all of this in their heads, of course; an agent arrives with none of it, so for agents it *must* be written down. This layer also holds your processes: the standard operating procedures for handling a bug report, running a design review, triaging an incident. And it, too, offers a concrete injection point. Git has hooks on every command — before and after commit, pull, push. The classic use is a pre-commit hook that runs a fast quality check before a change is allowed into the tree, moving feedback as far left as it will go. Above that, your CI pipeline is another set of stages where mechanisms attach.

Learn more about this governance mechanism: pre-commit hook.

At the top sits your application — the product code itself, which can carry governance of its own: models describing how it is supposed to work, and static analyses that check the code against them. A static analysis at this layer is where a whole class of mistakes can be made structurally impossible. The companion catalogue's sharpest examples live here: a **ban-lint that routes all mutation of a document format through a single structured model**, so that no agent can reach the raw library and silently corrupt the file. The rule is enforced once, at the one gateway, and it holds for every agent that ever touches that code.

Learn more about this governance mechanism: one structured model per format.

2.1.3 Every layer is a place to steer

Read the stack as a list of opportunities, because that is what it is. The harness noticing it is near its context limit and writing out a reliable summary of state and plan, instead of waiting for a lossy compaction, is an opportunity to help the model reconstruct less after the window resets. The pre-commit hook is an opportunity to catch a defect while everything that produced it is still fresh in the model's window, rather than surfacing it from a failed test three compactions later. The CI pipeline turning a failure into a message the agent can read is an opportunity to point it back toward the fix.

The single goal underneath all of it never changes. At every layer, you are trying to prevent the agent from making a bad choice, and to narrow its search toward the choices you want. This matters more than it first appears, because the context window is too small to hold any serious system — think of the US tax code, or a real web application — all at once. Give the model no structure and it must induce the structure as it goes, burning most of its window just working out the rules of the road; the

more structure you hand it, the more of that window is left for the work itself. Every mechanism you place on the stack is a rule of the road you handed it for free.

One thing the stack does not show: the mechanisms you install at these layers do not act in isolation. They run together, and the interactions *between* them have to be governed too — two that touch the same thing at the same moment pull against each other and confuse the very agent they were meant to steer. Governing each layer is not enough; the interplay is its own problem, and a later chapter takes it up.

The stack is where the two theses get their footing. A model handed to a layer enacts the Modeling Thesis; a hook or lint installed at a layer enacts the Alignment Thesis (derived alongside the Modeling Thesis in Part 1). The two chapters that follow take one thesis each.

2.2 Models and the Semantic Gap

The last chapter gave you four places to inject a mechanism. This one is about the most valuable thing you can put there — a *model* — and about a trap that catches people the moment they try to enforce one: pushing on the model at the wrong level, and watching it slip. A model is the sweet spot between prose too vague to bind and code too verbose to reason over. But a model only helps if you check it where checking makes sense.

2.2.1 Words are soft, code is verbose, a model is the sweet spot

Your first governance mechanism is the prompt, because the agent is a language model and words are how you talk to it. But words are soft. They are ambiguous, and the more ambiguity you leave in, the less happy you will be. You can push the ambiguity down by writing documentation — describing the environment, the product, its goals, its policies — and if you were briefing a *human* team, that would be enough. Humans are not dumb; they remember, they know what matters, they know where to look things up, and they know they will be fired if they misbehave. Agents are different: brilliant reasoners who must be told exactly what to reason over, and for whom a loose document is simply not precise enough.

Code is the opposite failure. Code is perfectly precise — it is exactly what will happen — but it is far too verbose to reason over in bulk. What you want sits between the two: something accurate and unambiguous, yet compact. The word for that thing is a **model**.

You already know models from physics. Think of $F = ma$, or $y = mx + b$. Neither is reality — F does not quite equal ma, and real data scatters around the line — but each is a cheap, honest approximation you can actually reason over, instead of wrestling the full empirical dataset or a nightmare of fluid equations. A circle approximated by tangents is not a circle, but it is good enough to compute with. Software has its own such models. UML was the famous attempt: a whole language of them, out of the 1990s, expected to be the future. It never took off, and the reason it never took off is the reason this whole chapter exists.

2.2.2 Why the models drifted — and why that changes now

The models are the map; the code is the territory. You draw the map, you build the territory from it — and then the territory *drifts*. Code changes constantly, and the more it changes, the more it costs to keep the map true. Worse, no single model captures everything, so teams either piled every concern into one unreadable mega-model, or split their concerns across so many views that a single code change meant updating ten diagrams. So people gave up and drew box diagrams instead.

Keeping the map equal to the territory is an old research program, not a new worry. The field spent decades formalizing how a change on one side should cross to the other: model-transformation theory mapped out the design space of how one artifact is derived from another[17] bidirectional-transformation work pushed past the two-artifact case to whole networks of models that must

[17] Krzysztof Czarnecki and Simon Helsen, "Feature-Based Survey of Model Transformation Approaches," *IBM Systems Journal* 45, no. 3 (2006): 621–45, https://doi.org/10.1147/sj.453.0621.

stay mutually consistent as any of them is edited[18] and architecture-conformance techniques let an engineer measure the gap directly, computing where the code converges with the intended design and where it has drifted away[19]. The correspondence was always achievable. What never got cheap was holding it reliably and economically while both sides moved — the same daily-labor cost that pushed everyone outside the regulated shops back onto the code.

That cost is what agents change, and this book measures the change rather than asserting it: the model-synchronization measurement in Part 3 asks whether an agent fleet can carry the recurring reconciliation labor a person used to carry by hand. The claim is economic, not historical — agents did not invent synchronization; they made a maintenance regime feasible that the Agile era could not afford. Synchronization cost is reduced and redistributed, not zero: what does not move to the fleet is deciding what the correspondence should mean. The rest of this section is that regime made operational — agents reasoning over the models, and a drift gate that turns a disagreement into a failed build.

Anyone who has worked in a shop that tried to keep models by hand knows the failure in its bones. A new hire opens a diagram and cannot tell whether it is describing the system or lying about it — whether the box that is missing means the code lost a component or the diagram lost a maintainer. The model was drawn years ago by someone who has since left, and nobody updated it, because updating it was always the task that could wait until after the release. The map and the territory drifted apart quietly, and the first person to trust the map paid for the gap. That lived failure — a hand-maintained model that rots the moment the person maintaining it stops — is why the practice fell out of favor. Only the industries where a mistake is catastrophic — Rolls-Royce, Boeing — kept their models honest, because they could justify the cost. Everyone else chose agility and leaned on the code.

Agents flip the calculation, because an agent cannot reason over the code — there is too much of it — but it *can* reason over models. So if you can find the models that usefully approximate your system and keep them in sync, your agents will always know where to go. They walk the docs, hit a pointer to the model — a data-flow diagram, a performance model, an inheritance hierarchy, a user journey — and if that model is linked to the relevant code, they can make the change with the whole picture held in a single window.

And which model the agent reaches for shifts with the move it is making. Planning a feature, it climbs to the architecture to see how the pieces fit. Writing the code, it drops to the module the model points at. Asking whether the change still serves the user, it crosses to the journey that says what the product must do. The models form a hierarchy, and the craft is meeting each step at the altitude where it is legible — high enough to see the shape, low enough to act. One sweet-spot model cannot give you that; a *set* of honest, linked models can, because each is the right map for a different moment, and keeping them cheap to maintain is what lets the agent move among them instead of getting lost in the code between.

The model also lets the agent *check its work*: a test says the suite passed, but an invariant over a model says what must be true beyond the tests. This is how you stop architecture from decaying — the agent finds the architectural model, follows it, verifies against it, and when the territory genuinely

[18] Perdita Stevens, "Maintaining Consistency in Networks of Models: Bidirectional Transformations in the Large," *Software and Systems Modeling*, ahead of print, 2020, https://doi.org/10.1007/s10270-019-00736-x.

[19] Gail C. Murphy et al., "Software Reflexion Models: Bridging the Gap between Source and High-Level Models," in "Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-3)," special issue, *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-3)*, 1995, 18–28.

moves, retrofits the map so the two stay equal. The companion catalogue's **drift-and-parity gates** are the mechanism that makes "the map equals the territory" a checked property rather than a hope: they fail the build the moment a model and the code it describes disagree.

Learn more about this governance mechanism: drift-and-parity gates.

Why agents don't reach for models on their own — a conjecture

Here a skeptic has a fair question. If working at the model level is such a win, why does the agent never reach for it on its own? Ask a coding agent to make a change and it dives straight for the code. It does not propose a model, does not ask where the map is, does not think to draw one. If models were the sweet spot, you would expect the model trained on all our software to have noticed.

I have a conjecture — and I flag it as a conjecture, not a finding. Two things may be compounding. The first is a gap in what the agent learned from. Serious model-based engineering happens inside regulated industry — aerospace, rail, power — where the stakes justify the cost, and that work does not leak into the open corpus a model trains on. The public code the agent read is overwhelmingly the agile kind that gave up on models and leaned on the code, so the agent saw very few real examples of the practice at all. The second is worse than absence. The corpus is full of *text about* models, and most of it is negative: the blog posts saying UML is dead, the war stories about diagrams that rotted, the received wisdom that model-based work is heavyweight and painful. So the agent is conditioned away twice over — starved of positive examples, and fed explicit priors that the whole approach is a mistake. There are, of course, other possible explanations; I think this one likely. And if it is right, the agent's reluctance is not evidence that models do not help. It is an echo of the era that abandoned them — and the thing that abandoned them, the cost of a human keeping the map in sync, is the thing agents just made cheap.

2.2.3 The apex of the documentation hierarchy is a structured model

Step back from drift for a moment and look at what you actually hand an agent to work from. It is never one thing. It is a hierarchy of descriptions, each aimed at the same system from a different angle. Prose docs at the top say what the thing is for; code comments say why a line reads the way it does; tests assert what must stay true; and schemas and typed records pin the shape of the data. Every one of these is a *description of intent* — and every one is partial, and every one drifts. The prose goes stale, the comment outlives the code it explained, the test pins yesterday's behavior, the schema lags the migration. This is the hierarchy the up-front half of governance climbs.

The usual agentic recipe gives you docs, code, and tests — and *nothing binding them together*. They sit as three separate islands. A change to the code does not touch the doc; a stale test does not know the schema moved; the prose swears the system does one thing while the code quietly does another. There is no glue, so there are no guardrails: nothing in the setup can tell you the three have fallen out of agreement, because nothing in the setup relates them at all. Drift, in this arrangement, is simply the default, and you find out only when it bites. Figure 2.2-1 draws the two arrangements side by side — the unbound islands, and the structured model that binds them.

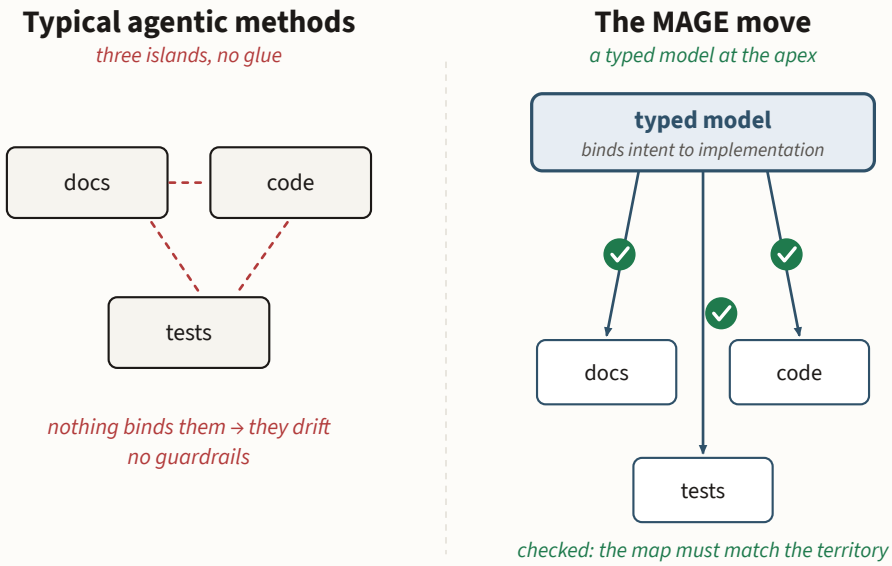


Figure 2.2-1: The Model at the Apex. *Left: the typical setup leaves docs, code, and tests as unconnected islands that drift. Right: the MAGE move puts a model at the apex, the one description the machine checks the others against — so a drift check fails the build when any disagrees.*

The MAGE move is to take the hierarchy to its limit. Climb past the most careful prose and what sits at the apex is a **structured model** — the description that binds the islands into one checkable whole. A model is a description precise enough that a machine can read it, and *because* the machine can read it, the machine can check the other descriptions against it. That is the property no amount of good prose can give you. The model becomes the single description you hold the docs, the code, and the tests up to: does the code still match the state machine the model declares? Does the schema still match the records? When the model and the territory disagree, a build-time check fails, and the disagreement surfaces the instant it appears instead of the instant it bites. This is the same **drift-and-parity gate** from the last section, seen from the other side: there it kept the map equal to the territory; here it is the glue the three-island picture was missing. It is the one edge that makes docs, code, and tests a bound system instead of three things that happen to describe the same repo.

That is the Modeling Thesis at full strength. Documentation matters, and agents have made thorough documentation cheap for the first time, so climb the hierarchy all the way up. The rung at the top is a structured model that binds intent to implementation and that a checker can prove things about. Everything below it — the prose, the comments, the tests — is honest and useful and still drifts; the model is the one description that cannot silently lie, because the gate stays red until it tells the truth again. This is why the companion catalogue gives the **models-bridge** its own role. The whole aim of

the up-front half is to build the one description the machine can check the rest against — and building it well is a craft the Model Zoo in Part 3 teaches by example.

One school of thought comes most of the way here and then stops one rung short. It agrees that loose prose is too soft, and it pushes intent into a fixed shape — but it keeps that shape in *constrained English*: requirements written to a strict grammar, structured to read like acceptance criteria, with a real model reserved for the cases whose stakes justify the trouble. This is the old cost calculation wearing new clothes. Structured English is still prose. A machine can pattern-match it, but it cannot hold the code up to it and prove the two agree, because the words carry no semantics a checker can act on. And reserving the model for where the stakes justify it is the same *where-warranted* hedge that once confined formal modeling to the high-stakes industries — a judgment that modeling is expensive, formed in a world where keeping the map current cost a person's standing labor. Agents cut that recurring cost sharply. When the map stays in sync almost for free, the structured model stops being the rung you climb to for the critical case and becomes the one you start from.

Modeling the surface turned up real bugs

The argument so far is that a structured model earns its keep by binding intent to implementation and letting a checker prove things about it. There is a plainer payoff the practice kept producing: modeling a subsystem surfaced real defects in it, not just a cleaner picture of it. Across a run of efforts that each built a structured model for a previously un-modeled part of the system, the act of writing the model — naming the states, the invariants, the seams — forced the kind of reading that finds bugs. It found a command-line seam whose exit codes were mislabeled, a resource leak on a cleanup path, an editing session that could lose an update because two writers were never fenced apart, and a queue-depth counter that drifted because it only ever moved one way. These were not style notes. They were live defects the tests had not caught, made visible because someone had to state precisely what the subsystem was supposed to do before they could model it.

Two honesties keep the claim the right size. Half of the defects were low-severity — the count is four real bugs, but two of them were minor — so read this as "modeling reliably surfaces something actionable," not "modeling finds a catastrophe every time." And the efforts that turned up no bug were not failures: each still produced an architectural strengthening, because modeling a clean subsystem still forces you to write down what "clean" meant. The mechanism at work is not luck. It is that you cannot author a spec — as opposed to mirroring the code — without deciding what the code *ought* to do, and that decision is exactly the reading under which a wrong "is" finally shows.

2.2.4 A drift check proves agreement, not correctness

The claim I just made hides a trap, and the book's enthusiasm for drift gates can be misread into a dangerous half-truth. A model wired to a build-time drift check cannot *rot* — that much is real. But it can still be *wrong*. The gate proves one thing: the model and the code **agree**. It says nothing about whether either of them matches what you meant. Both can move together, in the same wrong direction, and the gate stays a contented green the whole way.

Picture a model that says an endpoint is protected, and code that leaves it open. A drift check catches that — but only because the model already carries the concept that the endpoint is *meant* to be protected. Now derive the model from the implementation instead. The derived model reads the open endpoint and records it, faithfully, as unprotected. The code is unprotected; the model says unprotected; the drift check agrees. The bug is invisible, the gate is green, and everything is "in sync."

The power, then, was never in modeling as such; it was in the model being **authored independently of the code**. A model derived from the code is a **mirror**: it reflects what *is*, and a mirror cannot tell you the thing it reflects is wrong. A model authored from intent is a **spec**: it says what *ought to be*, and only a spec catches a wrong "is." People will generate a model from their codebase and believe they hold the same thing as someone who wrote one from intent. They do not. The drift check looks identical in both cases. What differs is whether an independent claim exists for the code to be measured against, or only a reflection agreeing with itself.

This does not weaken the drift gate; it tells you what the gate is *for*. Checking that the model and the code agree is mechanical, and the book automates it. Deciding **what belongs in the model** — which concepts it must carry, which "ought"s it must assert — does not automate. That is the essential residual Brooks warned would not yield, and under this method it moves one level up: from "write correct code" to "author the model that says what correct means." The gate is necessary. The authored model is what makes the gate mean anything.

We met a corner of this already, without naming it. A done-check can confirm the model was *updated* but not that it is *right* — the same asymmetry surfaces below, in the semantic-gap passage on the definition-of-done audit. And it recurs in the test dimension: a journey test that asserts only that the flow *ran* is a mirror of execution, while a typed task-closure post-condition is the authored spec that says what "done" means — the Scenarios view draws that line for oracles.

A model is a convenience, not a second copy of the truth

There is an over-reach on the other side of the mirror, and it is worth naming before the enthusiasm for models runs away. Grant that the model is a spec, authored from intent, and that the drift gate holds it honest against the code. It is still not a second place the truth lives. The code already holds the truth: it is exactly what runs. A model that only restates what the code plainly says has bought you a maintenance bill and nothing else — a faithful copy you now have to keep faithful.

What a model earns its keep with is what it lets an agent *do*. It puts the right place in one window, so the fleet navigates to a change instead of grepping for it. It carries the claim across a boundary two files should honor, so a root-cause walk reads the seam instead of guessing at it. It gives a benchmark somewhere to point. Each of those is an operation the raw code could not afford at the fleet's speed, and each is the reason to draw the model. None of them is "record what the code says a second time." So the test before you model a surface is not "could this be modeled" — almost anything can — but "what will an agent do with the map that it cannot do against the territory." Model where an operation needs the map. Where none does, the code is already the truth, and a model only adds a thing to keep in sync.

2.2.5 Just execute the model? A short history of CASE and MDA

If the structured model is where you now start, an old idea says to finish there too: skip the code and *execute the model directly*. The field has spent forty years on that idea, and where it landed marks the boundary this book keeps.

The dream had several names. Through the 1980s it was **CASE** — computer-aided software engineering[20] — tools to draw a design and generate code from it; by the early 1990s the market held

[20]Wikipedia, "Computer-Aided Software Engineering," 2026, https://en.wikipedia.org/wiki/Computer-aided_software_engineering.

nearly two hundred of them. Their prize was **round-trip engineering**: change the design and see it in the code, change the code and see it in the design, so the two never diverged. Tools like Rational Rose sold that to a generation. In 2001 the Object Management Group formalized the most ambitious version, **Model-Driven Architecture**[21] — write a platform-independent model, transform it into a platform-specific one, transform that into running code. With an action language, Executable UML went the whole way, to the promise this section is named for: do not write the code, *just execute the model*.

It did not take over the mainstream, and the reason was less a failure of the tools than a mismatch of tempo. The world went Agile, and software ate everything — most of it nowhere near safety-critical, and changing under you by the week. Heavy modeling could not keep that pace. Holding a model equal to code that moved every day cost a person's continuous labor, and outside the regulated niches (avionics, telecom, the shops where a mistake is catastrophic) the guarantees a model bought were not worth the rigidity it imposed. By 2006 one analyst house rated MDA "on the rise" while another had pronounced it dead on arrival. A survey of 450 practitioners found where it settled: teams rarely generate whole systems from models; they generate *key parts*, usually through small domain-specific languages built for one job.

Read that as a *spectrum*, not a verdict. At one end is no modeling: fast to change, nothing to keep in sync, no guarantee past the tests. At the other is the executable model that *is* the system: intent bound to code as tightly as it goes, and the least forgiving of change, because a model detailed enough to run is as large and verbose as the code it replaces. Every point between trades the same way: the further you model, the stronger the result — and yet the harder it is to move. Brooks still guards the far end. A model never makes the problem's *essential* complexity easy, so going all the way buys rigidity without buying simplicity. But where on that line you should sit was never fixed. It was an economic call, set by the cost of keeping the map equal to the territory, and that cost is what agents just changed.

So the book takes no strong stance on where to stop, because the sweet spot moved and is still moving. When keeping a model current costs a person's daily labor, you model sparingly and generate a slice or two. When a derived gate keeps the map equal to the territory for almost nothing, you can afford to sit much further along the line than the Agile era ever could. The old verdict against executable models is then worth reopening, not inheriting. This book sketches the spectrum and works a comfortable stretch of it: compact models you keep and reason through, with generation used narrowly where a model plainly governs an artifact. It does not claim that stretch is the optimum; the optimum shifted outward, and likely further than these pages go. The Executable Zoo is where the discipline for whatever point you pick gets its rules.

[21] Wikipedia, "Model-Driven Architecture," 2026, https://en.wikipedia.org/wiki/Model-driven_architecture.

2.2.6 The semantic gap

Enforce a property at the wrong level and it slips through; Figure 2.2-2 shows the fix — lift the check to where the meaning is legible.

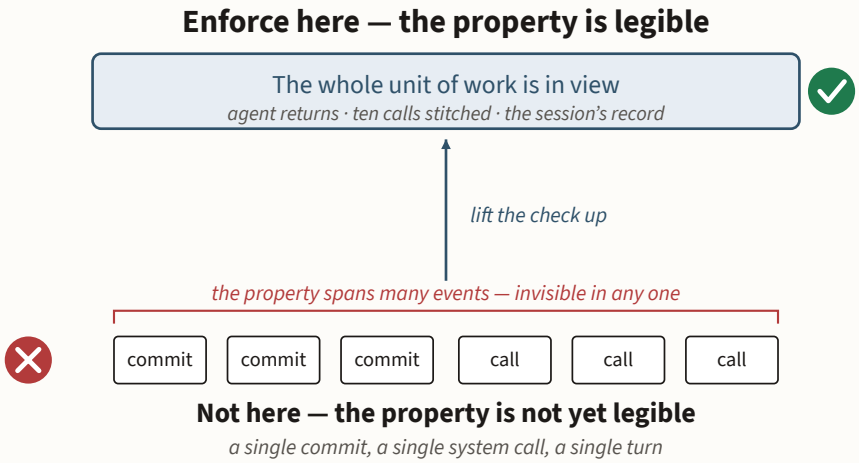


Figure 2.2-2: The Semantic Gap. *Bottom row: small events — single commits, calls, turns — where the property spreads across many, invisible in one, so enforcing here is the mistake. Top row: the whole unit of work, where it becomes checkable — lift the check to where meaning is legible.*

Now the trap. Say you want to enforce "if the agent substantially changed the call graph but did not update the overlaid model, reject it." It sounds like a pre-commit hook. It is not. An agent carrying out a real task makes *many* commits toward one goal; the model may be legitimately out of sync in the middle and correct again by the end. Enforce at the commit and you are checking a sentence for grammar before the paragraph is written.

This is the **semantic gap**: the failure you get whenever you enforce a property at the wrong level of abstraction. Operating-system security lives with it constantly — the OS sees individual system calls, but "this program is exfiltrating data" is a *pattern across* calls, invisible in any single one; you have to stitch ten network calls together before the leak is legible. Model drift is the same shape. The commit is the wrong granularity.

The fix is to find the level where the property is actually legible, and enforce there. For model drift, that level is the *agent returning from its task*, because at that moment the entire feature and its plan are visible. You can see whether the work changed the model, and whether the body of commits includes the tests, the documentation, and everything else the task was supposed to carry. This is exactly where the companion repository places its definition-of-done audit: a check that runs when an agent finishes an Epic, over the whole unit of work, rather than at any single commit inside it. "Correct" is a hard word — you cannot fully prove the model is right — but you *can* check that it was updated and that the surrounding work is complete, and that is the right semantic level to do it.

Learn more about this governance mechanism: definition-of-done audit.

2.2.7 A worked example: the context-fullness bank-hook

A second instance of the semantic gap lives not in the product but in the operator's own loop. It is small, it is concrete, and it shows the same trap from a different angle: the machinery you are given sits *below* the meaning you actually care about, so it guesses, and the fix is to move the check up to where the meaning is legible.

When I run a long session, the harness has a context window, and when it fills, the harness *compacts*: it summarizes the conversation so far and throws the raw history away to make room. Compaction costs something. It is a lossy save, and the harness decides on its own what to keep and what to drop. That decision is a **best-effort guess**, and it is a guess because the harness has no way of knowing which parts of the conversation were load-bearing. It sees tokens. It does not see that the three paragraphs where I settled the architecture matter more than the forty where an agent walked a directory tree. Only the operator knows what was worth banking and where it should go. The harness sits below the semantics of "what is worth keeping," so it cannot help but approximate.

That is the semantic gap again, in miniature. The property I care about — *is the durable record of this session current, so that a compaction loses nothing important?* — is not legible at the level where the harness operates. The harness knows one number: how full the window is. It does not know whether my strategy document and my hand-off notes reflect the last hour of decisions or whether they went stale ninety minutes ago while I was busy dispatching work.

The fix: a hook one level up, checking two things

So I moved the check up to where the property *is* legible: the operator's loop. Not inside the harness, which cannot see the meaning, but at a hook the orchestrator controls, which can. The hook fires as the session runs and checks two conditions together:

Learn more about this governance mechanism: orchestrator lifecycle hooks.

- **Is the context window getting full?** The harness exposes this. On its own it is not enough to act on. A full window with a fresh strategy doc needs nothing done.
- **Are the durable records stale?** Have the strategy and hand-off documents gone untouched for more than a few minutes while work continued? This is the part the harness cannot see, because it is a fact about *meaning*: whether the map still matches the territory of the session.

When both are true, the hook does one thing: it emits a reminder to bank the current state into the strategy and hand-off documents. Then, when a compaction does fire, a second hook runs on the far side — a post-compaction hook — and it knows exactly where to look to rebuild context, because the pre-work made the durable records the authoritative place to look. The bank-hook tells the operator to write the record; the post-compaction hook tells the freshly-compacted session to *read* it. The two ends meet at the same documents.

Why both conditions, and not one

Why insist on both conditions? Because the dual check is what keeps the hook from becoming a nuisance, and the nuisance here has a specific and nasty shape.

Suppose I fired the reminder on fullness alone. Then every time the window filled, I would be told to bank, even if I had just banked. Worse, suppose I fired it on staleness alone. Then every few minutes, whether or not compaction was anywhere near, I would be nudged to stop and write. Either way the

hook fires too often, and a hook that fires too often trains the operator to stop and re-summarize the session constantly. That failure has a name.

> **Sidebar — ouroboros.** *The ouroboros is the ancient image of a snake eating its own tail. In this loop it names the degenerate state where the model spends its turns summarizing its own recent work instead of doing new work — banking, then banking the banking, forever consuming its own output. A reminder that fires on a single loose condition drives straight into it. The dual check is the guard: fullness* and *staleness together are rare enough that the reminder lands only when it is genuinely worth acting on.*

Requiring both conditions is what makes the firing *meaningful*. A full window is common. A stale record is common. Both at once — the window filling *and* the durable state having drifted out of date — is the exact situation where a compaction would actually lose something, and that is the only situation where the reminder earns its interruption.

The honest part: the pre-compact hook is the wrong hook

I want to be straight about a compromise buried in this, because it is the kind of thing the semantic-gap idea should make you notice rather than paper over.

The *theoretically* right moment to bank is the instant before compaction — you would catch every last decision, lose nothing, and never fire early. The harness even offers a pre-compact hook that runs at exactly that moment. It is the wrong hook to use. Two reasons. First, it fires too close to the end: by the time you are at the edge of the window, you may already be past the point where a comfortable bank fits. Second, and worse, the bank *itself* costs tokens. Writing a thorough hand-off is real work in the window, so a bank kicked off at the pre-compact boundary can be the very thing that trips the auto-compaction it was meant to precede. You reach for the save and the save pushes you over. The hook that seemed to sit at the perfect level turns out to sit slightly past it.

So I do not use it. I approximate the right firing point instead — earlier, on the dual condition — and accept a little loss: sometimes the reminder fires with nothing new to bank, sometimes a decision lands in the gap and has to be reconstructed from the summary. This is not the sound placement the drift example gave us, where agent-return is genuinely the level at which the property is legible and you lose nothing by waiting for it. Here there is no clean rung, so a hook at *roughly* the right level beats both.

Two promissory notes come due later, and it is worth saying where. The special language for stating invariants *over* a model — linear temporal logic, the split between safety and liveness, bounded model checking, and formal invariant verification — arrives in The Process View, where a model stops being a picture and becomes something a checker can prove things about. And the question of how you actually author a model in a repo — a typed record, a schema file, a lint that reads it, a diagram that stays honest — is answered by example, not by lecture, across the Model Zoo in Part 3. You now believe models matter; those chapters show you how to build and check one. And the move underneath this one is bigger than models: *enforce a property at the level where it becomes observable*. You will meet the same discipline in security reading a leak across many system calls, in testing choosing the unit or the journey, in CI gating a merge and not a keystroke, and again in monitoring, distributed consensus, and formal methods — the semantic gap is one instance of a pattern that recurs across the field.

2.3 The Governed Environment: Ex-Ante and Ex-Post

Everything so far assumes you know, in advance, what should be true about your software. If you do, the path is easy, of course: write it all into a specification up front and hold the agent to it. But real software engineering rarely works that way — you discover what you are building by building part of it. So a governed environment has two halves: what you can decide *before* you start, and what you can only learn *along the way*. This chapter is about both halves, and about the two forms every governance mechanism finally takes.

That an agentic team needs governance of its own is beginning to draw formal treatment. Recent academic work proposes a domain-specific language for defining and enforcing governance policies across mixed human-agent software projects[22]. MAGE converges with that direction but works at a different grain: not a single policy language, but a repertoire of governance mechanisms drawn from a production case — the machinery this chapter and the companion catalogue lay out.

2.3.1 Ex-ante: everything you can decide up front

Some things you genuinely know from the beginning, and you should write every one of them down. This is specification-driven development, and where you can do it, you should. A coding style guide is the everyday example — Google has one, Airbnb has one — full of rules like "every function carries a comment" or "we never call `exec` outside this one approved place." Give the agent those rules, and then write *deterministic checks* for them. If a program's output has a required format, write the checker that confirms it — or, better, have the agent write the checker. Determinizing what you know up front gives you a much stronger starting point, and it costs you almost nothing, because the checks are cheap and the agent writes them.

2.3.2 Ex-post: everything you learn by building

But you will not know everything, and pretending otherwise is the mistake. You will discover that performance is a problem and you have no performance model. You will discover that a particular model version has a tic — every LLM's fondness for the em-dash is the running joke — and that the tics move when you upgrade. So you will build governance *ex-post*: after the fact, iteratively, strapping on new mechanisms as the failures reveal themselves and as the software's real properties emerge. You define what you can up front, and you stay willing to discover the rest. The rest is the mistakes you are making, the mistakes the model is making, the gap between what you thought the customer wanted and what they did, and the gap between how you thought the code would perform and how it did.

The honest version of the loop is less tidy than the ex-ante story makes it sound. You set up the governance you can imagine in advance, you launch the work, and then you **watch it burn and then try to fix it.** The failures you did not foresee are the ones that teach you which mechanism to build

[22] Adem Ait et al., "Towards Automated Governance: A DSL for Human-Agent Collaboration in Software Projects," in "Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE), New Ideas and Emerging Results Track," special issue, *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE), New Ideas and Emerging Results Track*, 2025, https://arxiv.org/abs/2510.14465.

next. Ex-post governance is not a fallback for a weak plan; it is where most of the real mechanisms come from, because most of the real failures are the ones no specification predicted.

The engine of the whole discipline is what you do to hold a quality goal: **you set up a mechanism that keeps it.** This is the organizing move of the companion catalogue. You place some mechanisms up front, from what you know about the domain, and you add more in response to a failure seen twice — a mistake caught in yesterday's manual audit becomes today's automatic check, enforced on every agent thereafter instead of re-caught by inspection. That conversion runs the printer loop's own shape (introduced in Loops and Models), escalated: when the loop keeps hitting one wall, judgment turns the failure class into a durable mechanism (Figure 2.3-1).

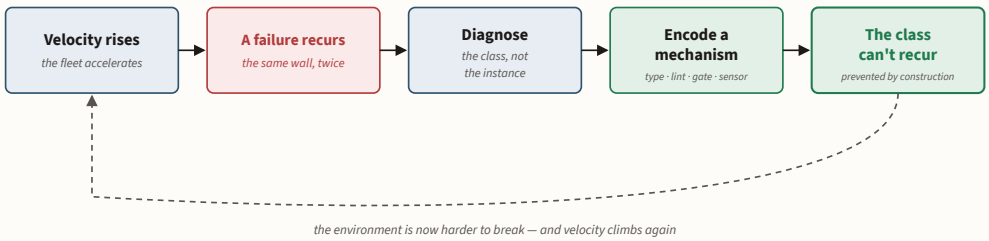


Figure 2.3-1: The Governance-Conversion Loop. *The loop that hardens the environment: when the printer loop keeps hitting one wall, judgment converts the failure class into a durable mechanism — the same loop shape as the printer loop, escalated.*

None of this is new as an instinct. Manufacturing stops the line at a defect and reshapes the work so the defect cannot recur; resilience engineering treats an incident as a signal about the system, not a verdict on the operator[23] site reliability practice turns each outage into a blameless postmortem whose action items become durable reliability work[24] Shingo built mistake-proofing into the fixture itself[25]. MAGE inherits that family and states its promotion rule for a fleet: an ex-post failure becomes an ex-ante control when judgment classifies the recurrence and encodes it in the environment. The changed ingredient is velocity: the fleet surfaces structural gaps fast enough that the environment can accumulate reliability one failure class at a time.

[23] E. Hollnagel et al., *Resilience Engineering: Concepts and Precepts* (Ashgate, 2006).

[24] B. Beyer et al., *Site Reliability Engineering: How Google Runs Production Systems* (O'Reilly Media, 2016).

[25] Shigeo Shingo, *Zero Quality Control: Source Inspection and the Poka-Yoke System*, trans. Andrew P. Dillon (Productivity Press, 1986).

2.3.3 The controls accumulate

Run the promotion rule for a season and the environment fills with controls. The case project shows the shape. Figure 2.3-2 counts its two hardest control surfaces — project-specific lint files and gate scripts — at the same four windows the build passed through; Table 2.3-1 gives the numbers behind the lines.

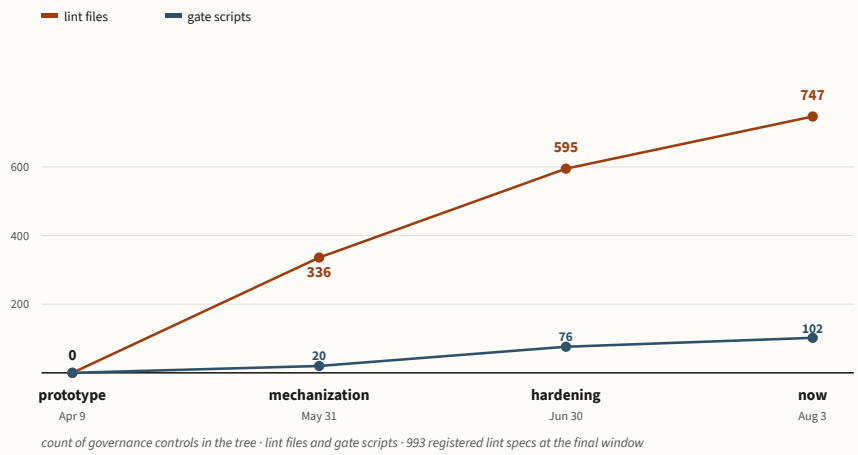


Figure 2.3-2: The controls accumulate across four windows — lint files (0 to 336 to 595 to 747) and gate scripts (0 to 20 to 76 to 102). Both start at literal zero: the substrate is post-prototype. The climb is steepest through mechanization and rises after, one class at a time.

Table 2.3-1: The control substrate across the four study windows — project-specific lint files and gate scripts counted from a tree scan at each dated commit.

Window	lint files	gate scripts
prototype	0	0
mechanization	336	20
hardening	595	76
now	747	102

The curve is the accumulation, not its cause. Those lint files carry 993 registered lint specs at the final window, and each spec is a policy the environment now enforces on every agent. What the count cannot separate is which controls a failure drove and which the domain suggested up front — the two modes interleave by design, and no commit trail cleanly splits them. But the discipline that converts a failure is itself documented and named. A paired fix-and-lint tag marks the commits that ship a code fix beside the check that forbids its recurrence, and it appears in 208 commits; 27 lints name a specific dated incident in their own text, spot-checked as genuine after-the-failure conversions. Read the curve as a documented, growing discipline, then — not a measured causal rate. For the data, see The Governed Environment: Ex-Ante and Ex-Post →

How large should the governed environment grow relative to the product? Conventional wisdom offers a rough rule — about one line of tests for every line of production. The timeline and the work measures the support ratio MAGE actually reaches against that yardstick.

Either way you arrive there, a mechanism takes one of two forms, and the difference between them is the difference between a firewall and a smoke detector.

2.3.4 Constraints and sensors

A mechanism is one of two moves — prevent the mistake, or catch it — and Figure 2.3-3 draws the split.

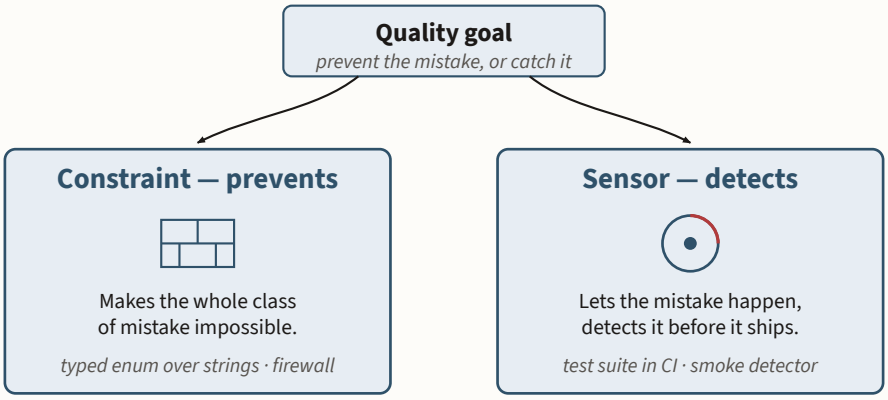


Figure 2.3-3: Constraint or Sensor. *The two moves a quality goal splits into. A constraint makes a class of mistake impossible (a typed enum; a firewall), while a sensor lets it happen but detects it in time to retry (a CI suite; a smoke detector).*

Before the two forms, the argument for reaching for a mechanism at all. Words alone do not hold. Tell the agent "make sure the briefs always follow this template" and it will say, *"Sure, boss"* — and then it will not reliably do it, not all the time, because it is a probability flip. A prompt raises the odds; it does not settle them. But you can *determinize that*. Instead of hoping, you check: before the agent launches, confirm every required part of the template is present, and refuse the launch when it is not. The soft instruction becomes a hard gate. That move — from a probability you nudge to a condition you enforce — is the reason the rest of this chapter exists, and it splits into the two forms below.

A sensor detects drift. It lets the mistake happen and catches it after the fact — and because it fires after, it fails the loop iteration: the agent has to run again to fix what the sensor caught. Your test suite in CI is a sensor, and so are your lints. It would be better if the model never wrote the bug; but if it does, the sensor catches it before the code ships. The cost of a sensor is iterations — detect, fail, re-iterate — and that cost is the reason you do not reach for one first.

A sensor is only as good as the signal it can read. **Observability** is the property that a running system exposes enough signal to explain what it did and to catch what went wrong. It is the wiring under every sensor: the event a gate fires on, the log a checker reads, the trace that turns "the build failed" into "the build failed *here, for this reason.*" Build the observability first and the sensor has something to watch; skip it and it stares at a blank wall.

A constraint prevents drift. Where a sensor detects an instance of the mistake after the fact, a constraint makes the whole class impossible: it scopes the agent's action space *within* the iteration, so the wrong move is never available to pick. The cleanest example is types. Models are, by default, careless with types — and a compiler exists to confirm that functions are called with the right arguments. Suppose the agent has represented a state with bare strings. Because it is a language model, it will cheerfully reach for a synonym next time, and the synonym silently fails to match — and you

only learn that an iteration later, when a sensor, the test suite, crashes. Tell it to use an enumeration instead, and the whole failure mode evaporates: it can no longer invent an unlisted value, and if it tries, the compiler rejects it on the spot. Choosing the enum over the string forbids the bug rather than watching for it, and because the wrong value can never be picked, no iteration is spent catching it. This is why the catalogue leans so hard on structured models and closed enumerations over free-form strings: a constraint that removes a mistake costs no iteration, while a sensor that catches it costs at least one.

Learn more about this governance mechanism: closed enumerations over free-form strings.

A footnote on control engineering. Constraint and sensor are the working terms of control engineering. A governed plant — a reactor, an autopilot — carries constraints on the inputs and states it may reach, and sensors on the outputs it produces, so the controller can read the error and correct the next cycle. I kept the terms because the parallel is exact. A fleet of probabilistic machines is a plant you cannot run open-loop; the governed environment is the loop you close around it. Constraints scope the action space, sensors read the output and fail the iteration, and the next iteration corrects. Engineers have run unruly physical processes this way for a century. Governing a fleet inherits the discipline along with the words.

Learn more: Nancy Leveson's *Engineering a Safer World*[26] applies the same systems view to safety engineering.

Two axes cross here, and it pays to hold them apart. A mechanism has a *form* — **soft**, it aims the agent, or **hard**, it holds regardless — and a *move* — a **constraint**, it prevents, or a **sensor**, it detects. The two are independent: a constraint can be soft (a structured model that aims the agent's reasoning) or hard (an enum the compiler enforces), and a sensor can be soft (a convention that reminds) or hard (a lint that blocks). Constraint-versus-sensor is *what* the mechanism does; soft-versus-hard is *how firmly* it does it.

Most real mechanisms are a *package* across both axes. A structured model is a soft constraint on the agent's reasoning — it aims the agent at the right shape without forcing it — and it ships with hard sensors, the lints and drift-parity gates that catch what the soft constraint only aims at. The definition-of-done review is the same shape: a runbook that constrains the steps, wrapped in sensors that flag a drifted one. So name the package, and tag its primary move rather than forcing it into one bin: prefer a constraint where you can build one, and add a sensor for the drift you cannot scope away. ("Architecture" is the design activity that builds constraints — not a third kind of thing.)

Seen from the outside, this is what a purely organizational answer leaves on the table. Faced with an unreliable worker, the instinct is to wrap it in process: a review, an approval, an evidence trail, a sign-off before the work counts. Every one of those is a sensor. Each lets the mistake happen and catches it after the fact: ex-post, an iteration late, an inspection of output the agent has already produced. A discipline built only of sensors can raise the odds and shrink the blast radius, and yet it never removes a failure class, because it watches the action space instead of scoping it. The constraint is the move that answer does not have: it works ex-ante, inside the iteration, so the wrong move is never available to pick. That is the difference between supplying reliability by construction and inspecting for it after the fact.

[26] Nancy G. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety* (MIT Press, 2011).

A footnote on the factory floor. The Engineer's Seat introduced Toyota's names for these two moves: the constraint is *poka-yoke*, and the sensor's fail-the-iteration reflex is *jidoka*. The lesson Toyota drew is the one this section just argued — inspection at the end of the line cannot build quality in. A fleet of agents is a production line for code, and it relearns the same economics.

Four classes, one bounded term

One boundary before moving on, because without it the word swells. Prompts, documents, lints, tests, metrics, hooks, models — everything in the environment starts to look like "a governance mechanism," and a term that covers everything governs nothing. Bound it:

> **Governance mechanism.** *A repeatable environmental structure that encodes an engineering policy and either constrains an action, detects a violation, requires evidence, or controls admission.*

The clause list yields four classes: **prevention**, **detection**, **evidence**, **admission**. The first two are the constraint and the sensor you just met — the catalogue's move axis tags them, with *package* naming a bundle of both. The other two sit at the loop's far edge. An evidence mechanism demands the change carry its proof: a provenance stamp, a re-derived test result, a completion re-checked rather than self-reported. An admission mechanism decides whether the result may advance at all — the admission rule of the engineered loop (1.3), holding the last door.

The evidence and admission classes compose into what is worth naming the **evidence staircase**: cheap checks run early and bind their result to an exact tree, so a later, more expensive stage can *check that binding* rather than *trust* a report; and the definition of done recomputes the full evidence at close instead of reading a marker. Cheap evidence early, full re-derivation late, never a trusted self-report — the same staged-admission-plus-re-derived-completion pair the catalogue lists as a composition, read here as how a loop earns the right to advance.

The definition also settles a boundary this book leans on: a model is not a governance mechanism until something reads or enforces it. A structured model on disk aims nobody. The lint that reads it, the gate that checks the code against it, the generator that derives from it — those are the mechanisms. The model supplies the policy; the mechanism supplies the holding.

What it adds up to

Step back and this is what you are really doing: describing policies for how the engineering work should proceed, and letting those policies produce a **governed engineering environment** — the rules of the road the agents operate within. That combination — determinize what you can, then convert every recurring surprise into a constraint or a sensor — *is* the governed engineering environment.

2.3.5 The residual: goals no mechanism reaches

That closing line reads as if it settles everything. It does not: some goals split into **neither**. No cheap sensor can see the failure and no constraint can scope it away, because the failure is an **absence nobody specified**, and you cannot watch for the violation of a rule no one wrote or forbid an action the design never mentioned.

Security remediation is the clean case. A missing authorization check has **no failing test — by definition.** If a test caught it, the check was specified; if it was specified, it would not be missing. The gap is

invisible because nothing ever named it. So the goal stays a **human** job: someone has to look at the system and *know* the check should be there. Call this the **residual**: the third outcome beside prevent and detect, the goals no mechanism reaches. Which of your goals land in it sets your real throughput ceiling. The mechanized goals run at fleet speed; the residual runs at yours — and it is the residual, not the size of the catalogue, that bounds how fast the whole system can go.

This is **not** the soft end of the soft-versus-hard axis you just met. A soft mechanism — a structured model that aims the agent, a convention that reminds — is still a *mechanism*: a named artifact that raises the odds and that you can point to. The residual has none. There is nothing to name, nothing to tag as a constraint or a sensor, nothing to add to the census. It is the *complement* of the two moves, not a third value on the axis.

Naming the residual costs the method nothing; a method that claims to mechanize everything is the one to distrust. The residual is the honest edge that makes the mechanized core credible — it says plainly what the catalogue does *not* cover, and that admission is what lets you trust the coverage it does claim.

Nor is the residual a fixed wall. You can sometimes pull a goal out of it by **authoring the missing spec.** The same absence that defeats a sensor defeats a *derived* model: a model read off the code records the missing check as correctly absent, and the drift gate agrees with the bug — a mirror cannot see what was never specified, which is the residual restated for models (2.2). And a journey test that only asserts “the flow ran” greens while the user’s task is broken, until a human authors the terminal task-closure assertion that says what *done* means — turning an un-sensable goal into a checkable post-condition (3.6). Both are the residual met again: once as a model that cannot reflect an unwritten intent, once as a spec that authorship supplies. What that human residual *becomes* when the model, not the code, is the thing you author — the enduring job that moves one level up — is the subject of the closing chapter (6.0).

2.4 Lifecycles and Runbooks

You know what to do when the disk fills up, when two branches conflict, when an incident pages you at 2 a.m. You know it so well you barely notice you know it. The agent knows none of it. Out of the box it knows how to commit to Git, of course; what it does not know is the life cycle of *your* engineered environment. The disk fills, and here is what to do when the disk fills — that instruction will never be in the agent's context unless you put it there. It has no operating context, no sense of "how we do things around here," and the more of that judgment your operation runs on unwritten, the more helpless the agent is the moment the healthy path ends. So a governed environment for real work writes the knowledge down — as lifecycles, and as the runbooks that ride on top of them. This chapter is about capturing the operational judgment you carry in your head so an agent can act on it.

2.4.1 Lifecycles first, then the failure map

Start with the lifecycles, because everything else hangs off them. A lifecycle is the correct operation of one recurring activity: how a bug report moves from filed to fixed, how a feature moves from design to shipped. Write the healthy path first. Then, from each node on that path, map the failure states — the ways this step can go wrong — and for each failure, the action to take. Think of it as a grid: every node crossed with every symptom, and in each cell, what to do.

The formal name for building that grid is **failure mode and effects analysis**. You enumerate what can go wrong, then the consequence of each, and then — the part people skip — how to get *out* of it again. You know how to get out. If two branches conflict, you read both sides, work out what each was trying to accomplish, and reconcile them. If the disk fills, you find the fat part of the system — the cache, usually — and wipe it. The agent knows none of this. It does not know where to find each commit's intent during a merge conflict, and it does not know where your system's fat points are — and it may, just maybe, decide the fix for a full disk is `rm -rf /`. You think I am joking. Read Twitter; agents do it all the time. So you coach it: where the hot points are, and, in every failure path, what is *not allowed*. Encode the prohibition as a mechanism at the right level where you can — a Git hook, an agent hook — and where you cannot, at least tell the agent plainly.

> *A footnote on the pink elephant.* Telling an agent not to do something raises an honest question: does naming the forbidden act make it more likely? A human cannot help picturing a pink elephant once told not to — though without the telling, the thought would never have come. I don't know about you, but when I read the law — or the police blotter — it gives me all kinds of wicked ideas I would never have thought of on my own. In my own system the naming helped, but only when I named the escape alongside the sin. My orchestrator kept reaching for a destructive Git command to bail out of a stuck merge — the one that resets the branch and silently discards the changes already applied — twice losing committed work that way. A bare prohibition ("never abort the merge") would have left it stuck with nowhere to go. What worked was pairing the forbidden command with the sanctioned one that clears the stuck state *without* dropping the applied work, so the rule read "not that lever, this one." Name the sin alone and you may plant the temptation; name the sin *and the way out* and you replace it.

2.4.2 Runbooks: split the deterministic from the judgment

Once the healthy path and the failure map are written, the thing you hand the agent is a **runbook** (or playbook): the steps to take, whether recovering from a failure or working a fresh ticket. The one discipline that makes a runbook work is to split it into two kinds of step — the deterministic and the judgment-laden — because the two are governed completely differently. Figure 2.4-1 fences the one judgment step between two deterministic ends.

Learn more about this governance mechanism: runbook.

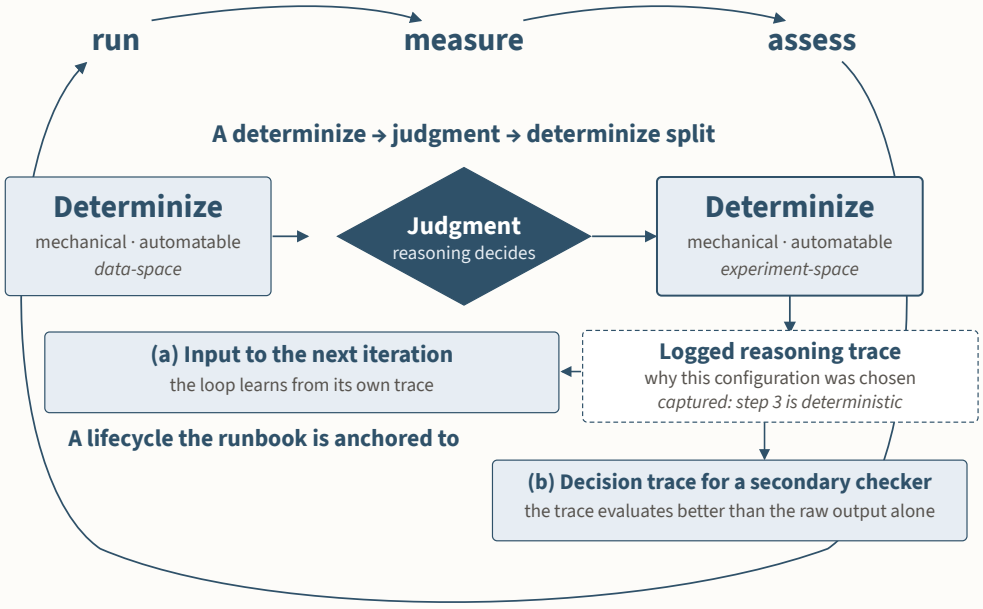


Figure 2.4-1: The Fenced Judgment Step. *A run-measure-assess loop wraps a three-step spine: determinize the data space, apply judgment in reasoning space, then determinize the experiment space again. Fencing the one judgment step between deterministic ends bounds the messy part and logs a trace a checker can grade.*

The deterministic parts get tools. Anything that has a single correct outcome should be an executable script the agent runs, not a procedure it improvises. Leave it to improvisation and it will get it right most of the time — and you will wish you had automated it the one time it does not.

Determinizing a step buys you more than a reliable outcome — it buys you a **trace**. When a step improvises, its reasoning is smoke: gone the moment it finishes, impossible to inspect or compare. Pin the step to an executable procedure and the reasoning becomes a record — the inputs it read, the branch it took, the value it produced — written down in a fixed shape. That record has two consumers, and between them they are the reason to capture the trace and not merely the outcome. The first is **the next turn of the loop**: last run's trace is this run's input, so the agent starts from what was already reasoned instead of re-deriving it, and the loop converges instead of wandering. The second is **a second checker that reads the trace itself** — a reviewer, human or agent, that never watched the step run but can now audit *how* it reached its answer and flag a trace that took a path it should not have. A judgment step gives you only its verdict; a determinized step gives you the verdict and the reasoning behind it, and the reasoning is the part a checker can hold to account.

The judgment parts: the roughest algorithm that still fits

The judgment parts get the roughest algorithm that still fits. For a well-constrained decision, write the algorithm down and let the agent do the glue. For an open-ended one — a cyber threat hunt, where the premise is that the attacker did something you never anticipated — do not over-constrain: give heuristics and strategies to explore, and stop there. Remember that you are conditioning a probability distribution: steer it down the wrong path and it will never reach the place you wanted. So if you do not know how to do the thing, do not write as though you did — give hints. If you truly do not know, say "solve this open-endedly," and accept the trade: the looser the guidance, the more tokens the agent spends, and the less likely it is to land.

When a judgment step is really a *measurement*, give it a **rubric**. You cannot write a checker for "is this painting good" or "is this student essay good," but you can operationalize it: how are the topic sentences, how is the technique, are there grammatical errors, are there blotches from bad storage? The agent will not deliver a final verdict on taste, but with a rubric it gives feedback good enough to act on.

The strongest form of this is the **pre-canned brief**. When an orchestrator agent has to hand work to a sub-agent, you do not want it inventing, each time, how to explain the task. You want a template it adapts — the work already articulated in writing. The companion repository leans on exactly this: every dispatch is composed from a template that pre-places the required instructions, and a brief-linter checks the template is intact before the agent launches. Templates constrain how work is described to agents, which is the whole business of this book, applied to the act of delegation itself.

Learn more about this governance mechanism: brief-linting.

The split has a mechanical form once you write runbooks at scale. Type every step as one of three kinds, and let the type decide how the step is governed. A deterministic step is *runnable*: it carries the exact tool line to run. The judgment steps divide by whether an agent can take them. A *judgment-automatable* step carries a pre-canned brief the orchestrator dispatches to a strong model, the missing middle between "run this" and "you decide." A *judgment-irreducible* step carries the words to surface to the person, because the call must stay theirs. Table 2.4-1 gives the three kinds and what each must carry. Typing the step makes the shape checkable: a tool line on a judgment step, or a dispatched brief on a runnable one, is a contradiction a linter catches before the runbook ships. The failure map from the last section joins the same way, each symptom routed to its runbook by its class rather than its exact error text, so a fresh failure that rhymes with a known one still finds the way out.

Table 2.4-1: The Three Step Kinds. *What each kind must carry and how each is governed: a deterministic step runs a tool, while the judgment steps split by whether an agent or a person decides. Typing the step makes the split mechanical — a mismatch is a lint finding.*

Step kind	What it is	What it must carry	How it is governed
Runnable	A deterministic step with one correct outcome	The exact tool line to run	The tool runs it; its logged trace is what a second checker reads
Judgment-automatable	Reasoning no script can do, but a strong-model dispatch can — the missing middle	A pre-canned brief the orchestrator dispatches	A brief-linter checks the brief is intact before launch

Step kind	What it is	What it must carry	How it is governed
Judgment-irreducible	Reasoning that must stay with the person	The words to surface, and why	Surfaced to the operator, never dispatched

Step back from the taxonomy and the chapter is one instruction wearing three shapes. A lifecycle writes down the healthy path and the way out of each failure. A runbook rides on top and splits every step in two: the deterministic part earns a tool, the judgment-laden part earns the roughest algorithm that still fits. The pre-canned brief hands that split to the next agent intact. What used to live only in your head — how the work is done here, and what must never be done — becomes something the environment holds and can enforce. Delegate the operation, not just the code, and it stops depending on who happens to be awake at 2 a.m.

2.5 Metrics

Every agent runs a loop. It takes an input, reasons, acts, and folds the result back around. The chapter on loops said the metric is what the agent steers by — skip it and the agent searches forever, never knowing when to stop. That chapter spent its words on the acting half. This one develops the sensing half, because a loop with no honest sense of its own state cannot steer at all.

A metric senses. A playbook acts. A mechanism enforces. Those three make up a governed loop, and the metric comes first because the other two are blind without it. A playbook that fires on a number that measures the wrong thing does the wrong work. A mechanism that grades on a number too coarse to name the fault blocks the wrong commit. So the question that governs every metric in this chapter is not "can I measure this?" — almost anything can be counted — but "does this number drive a decision?"

2.5.1 Measure one level deeper

John Ousterhout gives the rule in one line[27]: **measure one level deeper than the number you think you want.** A metric earns its place only when it drives a decision, and the surface number usually cannot; the deeper the measurement, the nearer it sits to the decision it must drive.

Think of the cloud bill. It tells you what the whole project spent last month. That number is real, and useless: it tells you to pay the invoice, and nothing about what to change — which service to slim, which call to batch, which cascade to cut. It affords the report; it cannot afford the engineering. Split the same bill by service and it starts to act. One service usually dominates, the rest trail off in a thin tail, and the dominant one names your first target. You measured one level deeper, and the deeper number carries a decision the shallow one buried.

> ***Worked example — boot time, one level deeper***
> Say a service takes 8 seconds to boot. As one number it means only *scale out* — add instances to hide the wait. Measure one level deeper — 6 of those seconds are the GenAI client warming up, the rest a thin tail — and it means *optimize the GenAI-client init first*. Same latency, one level deeper, and now the number drives an engineering decision (fix the slow phase) instead of a reflexive scale-out. (Illustrative figures.)

The same trap hides in test coverage. "The suite covers 87% of lines" is the invoice-level number — a headline that reports and does not act. It cannot tell you *which* untested code matters, because a line in a throwaway string helper counts the same as a line in the job state machine. The rest of this chapter walks a spectrum of metrics from the hard, deterministic end to the soft, judgment-laden one, and the flagship in the middle is coverage measured one level deeper — coverage that names the gap instead of hiding it in a percentage. Table 2.5-1 collects the move, and the wrong turn each surface number invites.

Table 2.5-1: Measure One Level Deeper. *For each surface number: the wrong move it invites, the decision it must drive, and the deeper cut that acts.*

[27] John Ousterhout, *A Philosophy of Software Design* (Yaknyam Press, 2018).

Surface number	The wrong move it invites	Decision it must drive	Measured one level deeper
Whole-project cloud bill	Pay the invoice; nothing to change	Which service to slim, which call to batch	Cost split by service — one dominates and names the first target
“Boots in 8 seconds”	Scale out — add instances to hide the wait	Scale out, or fix the slow phase?	Boot time by phase — 6 s is GenAI-client warm-up; optimize that init
Cold-start latency, per service	Pay to keep a warm instance (or blame image size)	Warm-instance lever, or re-architect?	The warm floor’s cause — a per-request subprocess launch; keep the runtime resident (4,057 → 109 ms)
Cold-start across a request	Trust the fleet average or per-service number	Which chain bounds worst-case latency	Longest-path sum over the journey graph — each step’s slowest cold-start, summed
“87% of lines covered”	Chase the percentage upward	Which untested code actually matters	Model-claim coverage — the named invariant (INV-18) no test exercises
“Is this doc-claim pinned?”	Wire it to a gate — which manufactures hollow tests	What to test next (a reader’s call)	Whether a seam exists to test through — pinning shape, not behavior, measures nothing

The spectrum tracks the book's soft-versus-hard governance axis, and the pair splits cleanly. A **hard** metric is deterministic: a machine reads it, and the same input yields the same number every time. A **soft** metric is judgment-laden: it points a human or an agent at a place worth looking, but it cannot be reduced to a threshold that decides on its own. Both are worth having. The mistake is to treat one as the other — to block a build on a soft signal, or to leave a hard fact to a reviewer's memory. Figure 2.5-1 lays the metrics out from the hard, deterministic end to the soft.

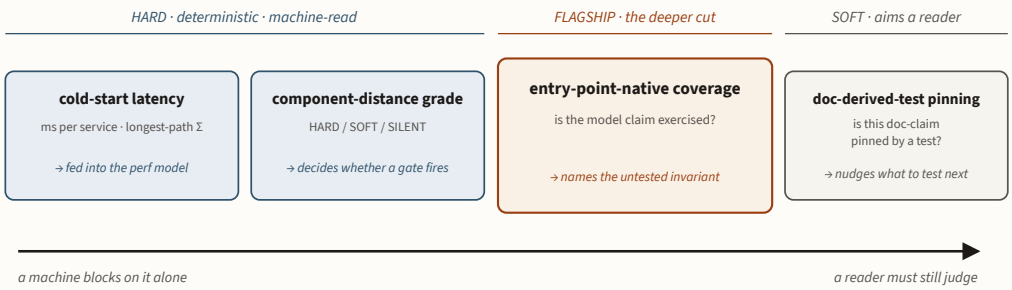


Figure 2.5-1: The Metric Spectrum. *The harder the metric, the more a machine can block on it alone; the softer, the more it can only aim a reader who must still judge. The flagship sits in the middle: entry-point-native coverage, the deeper cut that names the untested invariant.*

2.5.2 The hard end: cold-start latency, machine-read and model-fed

Start at the deterministic end, where a metric is a millisecond number nobody argues about.

A serverless fleet scales to zero. When a request wakes a cold instance, the user waits while the container boots and the runtime loads. That wait is the cold-start latency, and the obvious way to read it, of course, is as the penalty of scaling from zero: measure it, and if it hurts, pay to keep a warm instance running.

The obvious reading was wrong, and measuring one level deeper is what showed it. The conformance service — a PDF validator wrapping veraPDF, a Java tool — carried a 4,057 ms *warm* floor, a wait a warm instance did nothing to remove. The deeper number named its own cause: veraPDF ran as a subprocess launched once per request, so every call paid a fresh JVM startup. The tax was the process launch itself, and no warm-instance lever could reach it. The fix was a re-architecture — keep the runtime resident instead of relaunching it — and the warm floor fell from 4,057 ms to 109 ms, a thirty-seven-fold cut that took nearly four seconds off every request. The .NET service told the same story smaller: 494 ms of per-request CLI launch, down to 103 ms once the runtime stayed resident. And the tempting surface explanation, that the container images were simply too big, was also wrong — a 433 MB service cold-started fine. The barrier was runtime initialization, not image size, and only the deeper measurement said so.

That is a metric read at the level where the decision lives: resident-runtime versus warm-instance is a different fork than the shallow cold-start number implied, and a fleet-average number would have buried it. But I found that tax by hand — spot-checking the one service whose latency looked wrong — and spot-checking does not scale. It catches the service that happens to draw your eye, misses the ones that do not, and says nothing about how the services interact. The better instrument is a model that holds every service's numbers and reasons over the paths between them, so the costly case falls

out of the graph instead of out of a hunch. Because the sharpest question is the *chain*. A request that lands when every downstream service is cold triggers a cascade of cold-starts along the critical path. The revenue-core journey runs its steps in sequence; within a step, its calls fan out in parallel. So the worst case is a **longest-path sum**: each step contributes the slowest cold-start it waits on, and the path total is the sum of those per-step maxima. That sum, and neither the per-service number nor the fleet average, predicts the worst latency a real user can hit. It is the metric measured at the level where the decision lives.

Here the hard end shows its defining move: **the number feeds the model.** The cold-start measurements do not live in a dashboard a human reads and forgets. They land as typed fields on the deployment-topology model the fleet reasons through — a `cold_start_ms` and a `warm_ms` on each service's plane record. A recompute tool reads those fields and derives the longest-path worst case over the journey graph. When a later measurement writes a fresher number onto a service's annotation, the tool reads it live and the worst-case total refines on its own, no edit required.

Learn more about this governance mechanism: deployment-topology model.

The commit is automated. The tempting alternative is a runbook step — *after a measurement run, remember to commit the new numbers.* A reminder like that rots; someone forgets, and the model drifts from the fleet it describes. So the pipeline commits the refreshed measurements itself, under a bypass-prefixed housekeeping commit with no human in the loop, and the model's cold-start parameters stay current with zero manual action. Build the map's upkeep into the machine that changes the territory rather than asking a person to remember it — architecture over a reminder. The runbook note then documents an automated step instead of a chore.

Where this stands: the live-annotation path is built but not yet populated — today the tool runs off a snapshot table of measured values, each row citing the commit that measured it, because the fleet is mid-migration and no service carries the live annotation yet. The machinery to read the model live exists; the model is not fully fed. That is the ordinary shape of real work: the machinery lands first, and the data catches up. What matters for this chapter is the pattern — a hard metric, measured deterministically, written into a structured model, and consumed by a tool that turns it into a decision (which path, which service, which lever).

2.5.3 The flagship: coverage measured one level deeper

Now the middle of the spectrum, and the chapter's teaching example: the "measure one level deeper" rule applied to the number that most often gets it wrong.

Line coverage answers a **syntactic** question: did this line run under some test? That is a fact about source text, and it is genuinely useful. But it cannot answer the question an engineer actually has, which is **semantic**: is the *claim this code makes* exercised by a test? A file can sit at 95% line coverage while the one function that implements a critical invariant — call it INV-18 — is never called by any test at all. Its lines happen to be covered incidentally, swept up by an unrelated test that imports the module. The 95% is true. It is also a lie about the thing you care about, and the lie is invisible because a percentage has nowhere to put the gap.

The fix comes from joining two things the system already has. The first is a **traceability graph** that links each model claim — an invariant, a state-machine transition, a service-flow edge — to the code that implements it. The link points at an **anchor**: the entry point, the `(path, symbol)` of the function

that carries the claim. The second is the ordinary coverage oracle, which knows exactly which source lines ran. Compose them:

> anchor → its symbol → that symbol's line range → intersect with the covered-line set → *is this model claim's code exercised?*

That composition changes the unit of measurement. Line coverage counts lines. This counts **model claims**. The anchors supply the numerator's units; coverage supplies the ground truth of what ran. INV-18′s anchor resolves to its function, and the function's lines are checked against what the tests actually executed. The metric then reports a named verdict rather than a bare percentage: *INV-18′s implementing code is not exercised by any test.* An anonymous 87% becomes an actionable list of the exact model claims your suite walks past.

This is not a new invention, and it should not be. Run the genre check and the family has a name: **requirements-based coverage** — the discipline DO-178C mandates for avionics, and the shape the open-source Doorstop tool encodes with git-native requirement items linked to their tests. The system's traceability graph already adopted that schema. So there is no bespoke "model coverage" number to dream up. The work is to compute the requirements-based-coverage figure over a graph that was already built for it — and to add the one synthesis the off-the-shelf tools lack. Doorstop checks that a requirement *has* a linked test. It does not check that the linked test *executes* the requirement's code. Joining the trace edge to the live coverage oracle closes that last gap.

> Learn more about this governance mechanism: requirements-based coverage.

The single primitive underneath — `anchor_exercised`, which resolves an anchor to a line range and intersects it with the covered set — supports several aggregations, each answering a different decision:

- **Model-Element Coverage.** Of the model claims that anchor real code, what fraction is exercised? This is the headline: it turns 95%-line-covered into a named list of unexercised invariants, each a fix target.
- **Entry-Point Coverage.** Of the code roots the models declare as "changes here mean drift," which are untested? This is the number to watch when refactoring — it tells you which anchored roots a regression would slip past.
- **Chain Coverage.** Does each model claim close end-to-end: code exists, a test names it as its verifier, *and* that test actually runs the code? This is the strictest cut, and it will report low at first, because few claims declare a verifying test today. That low number is the metric doing its job: it names the verification chains still to build.

The honesty layer matters as much as the aggregation. A single percentage lies when its denominator is fuzzy, so every figure reports a breakdown: exercised, not-exercised, unmeasurable (the anchor could not be resolved to code), and out-of-surface (a database-tier claim that is legitimately not code you can cover). "80%" then reads as what it is — *80% of the fifty measurable claims are exercised; twelve more are unmeasurable and owe a burn-down; eight are out of surface by design.* That is a number that affords the engineering, because you can see which part of it you can act on.

The design is deliberately light. Computing all of this requires no change to any model — the anchors already carry their `(path, symbol)`, and the covered-line set is a plain read of the coverage database the test runner already writes. The metric is a new consumer of two things the system already paid

for: the traceability graph and the cross-language coverage oracle. Measuring one level deeper cost a join, not a rebuild.

2.5.4 The dual: draining unmodelled code

The flagship walks from a model claim *to* the code and asks whether a test exercises it. Turn the arrow around and a second metric appears. Walk from each *test* to the code it runs, and ask whether that code reaches any model-anchored symbol at all. A test whose exercised code reaches no model is an **orphan** — it guards behavior no model describes. The orphan rate, clustered by the unmodelled code the orphans touch, is a ranked list of the models still missing. Figure 2.5-2 shows what happened when the fleet worked that list down; Table 2.5-2 gives each re-run.

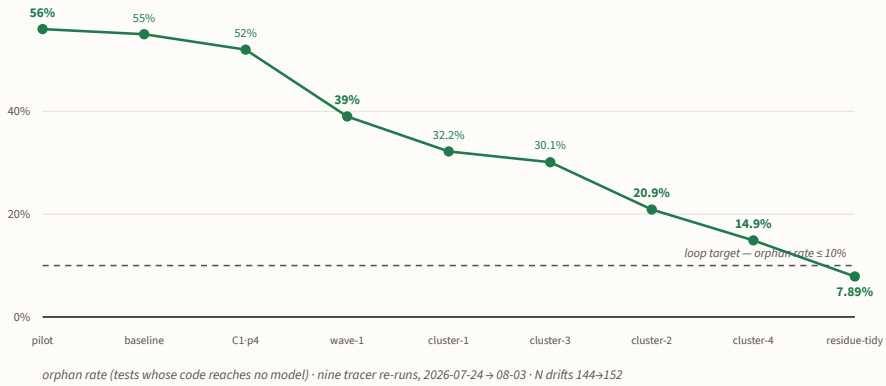


Figure 2.5-2: The Missing-Model-Metric Drain. *The orphan rate — tests whose code reaches no model-anchored symbol — falls from 56% to **7.89%** across **nine** tracer re-runs, each following an Epic that modeled the biggest remaining cluster. Coverage rises as the mirror image toward **92%**, crossing the loop's ≤10% target.*

Table 2.5-2: The Missing-Model-Metric drain, run by run — the orphan (code-unmodelled) rate and its code-modelled complement after each model-loop Epic modeled the biggest remaining orphan cluster.

Re-run	What it modeled	Orphan rate	Code-modelled
pilot	dispatch subsystem	56%	44%
baseline	post link-lint	55%	45%
C1·p4	after C1 phase-4	52%	48%
wave-1	after C1 / C2 / C5	39%	61%
cluster-1	persistence-sidecar seams	32.2%	67.8%
cluster-3	a11y-validate / editor-IR	30.1%	69.9%
cluster-2	redis-comm dispatch	20.9%	79.1%
cluster-4	cost-rollup	14.9%	85.1%
residue-tidy	web-route/seam links + control-nodes	7.89%	92.1%

The metric is a model-discovery instrument, not a gate. Each re-run names the biggest remaining orphan cluster; an Epic then builds the missing structured model and its invariants for that subsystem; the tracer runs again and the orphan rate falls, because that subsystem's code now traces up to

a model. Repeated cluster by cluster, the drain took the orphan rate from 56% at the pilot to **7.89% at the latest committed point** — code-to-model coverage rising from 44% to about **92%**. And once the residual dozen orphans are set aside as glue code — web-route and seam wiring with no model to miss — the **genuine-orphan rate is 0%**: every test whose code could trace to a model now does. The loop crossed its ≤10% target and, on genuine coverage, its tighter <5% goal. The metric drove the modeling, and the falling curve is the record of the modeling landing.

Two honesty notes bound the claim. The population is comparable across runs but not fixed: the denominator drifts from 144 to 152 tests as the suite evolves, and the first 56% point was scoped to the dispatch subsystem before the instrument widened. So read the curve as the same instrument re-run on a near-identical, evolving population, not as a fixed-N replay — and one interior point, the cost-rollup cluster's exact landing, is not separately enumerated in a committed record. A clean fixed-N series would re-run the tracer at each tagged commit over one frozen population; that measurement is pending and would sharpen the curve without changing its direction.

The same graph, read backward: a lens on accidental complexity

The link graph that drain walks has a second use, and it comes for free. Reading it forward — model claim to code — measured coverage. Read it *backward* and it starts to surface duplication the code hid. Two symbols that carry the same action but hang off different model nodes are a consolidation waiting to happen. A vocabulary mirrored across a service boundary — the same fact spelled twice, once on each side — is one model both sides should have read from instead. And a third pattern the loop turned up on its own: incomplete composition, where some paths reach the canonical seam and a sibling path quietly reaches a partial one. That last read caught a real resource leak on a cleanup path, not just a tidiness note.

So the model earns a second keep. The first is coverage — it tells you what is un-modeled. The second is a lens on accidental complexity — the traceability edges, inverted, point at duplication and half-finished composition the raw code does not advertise. This is an observation, not a tool. The loop fired it by hand, on some of the modeling efforts and not others, and left it a field note rather than a query surface — the yield has not yet earned the machinery, and building it before it does would be the over-engineering the method warns against. Named here because it is the kind of return a good model keeps giving after it has paid for itself: you drew the map to see what was un-governed, and it also shows you what was built twice.

2.5.5 The soft end: doc-derived tests

Slide to the far end of the spectrum, where the metric stops being a number a threshold can act on and becomes a judgment a human has to make.

A **doc-derived test** pins a behavior stated in a cited document or spec. The test carries a trailer naming the source it derived from, and when that source is edited, the test is regenerated so the pin and the prose stay in step. The metric it invites is: *is this doc-claim pinned by a test?* That question is genuinely soft. It has no clean denominator. "How many claims does this document make?" is a matter of reading — one reader finds six, another finds nine, and neither is wrong. You cannot reduce it to a percentage the way you can reduce line coverage, because the set of claims is not enumerable by machine. The signal points a reader at a document and says *these behaviors deserve a pin* it does not decide.

Learn more about this governance mechanism: doc-derived test.

The softness runs deeper than a fuzzy denominator. A doc-derived test is only as good as the seam it tests against. If the code interleaves its decision logic with live I/O — a poll loop welded to a queue, a validator welded to a database read — there is no pure surface to assert a value against. The test then falls back to pinning the *shape* of the source: that a function exists, that a docstring reads a certain way. That kind of test stays green even when you gut the function's body. It measures nothing about behavior. So the doc-derived-test signal is inseparable from a judgment about readiness: are the types sound, is there a seam to test through, is the contract even documented? Only a human (or an agent reasoning as one) can answer that, which is what plants this metric firmly at the soft end. Treat it as hard — wire it to a gate that blocks on a coverage-of-doc-claims threshold — and you manufacture the very hollow tests it was meant to prevent.

A null result means nothing until the benchmark can discriminate

The most useful honesty in measurement is the one that catches you before you publish a comfortable non-result. An ablation was run to see whether giving the fleet a structured model helped it on a fixed set of tasks: run each task with the model and without it, and compare the scores. The comparison came back a null — no difference. The tempting reading is "the model did not help." The correct reading is that the instrument could not have shown help if help existed, because both arms scored at the ceiling. When the with-model arm and the without-model arm both nearly max the rubric, there is no room between them for an effect to appear. A null from an instrument with no discriminative headroom is not evidence against the hypothesis. It is not evidence for it either. It is simply uninformative, and treating it as a finding is the mistake the honesty is there to stop.

There is a structural reason the per-task cut was blind, and it generalizes past this one study. A task small enough to freeze into a benchmark cell is usually small enough to solve without the map. The value a structured model buys is a system-level one — the fix that lands once and holds everywhere, the drift a gate catches over months, a context-bounded fleet operating a codebase no context can hold at once — and none of that shows up inside a single self-contained gate-pass. So the right lesson is not "the model did not help." It is "measure the effect at the level where the effect lives," and be willing to say, in print, that a chosen instrument could not see it. The study that produced this null also retracted an earlier claim that had read the same null as a measured absence — correcting the record when the data would not support the sentence is the discipline that makes the rest of the numbers trustworthy.

2.5.6 The model runs the machinery

The hard end and the flagship follow the same pattern.

The MBSE models do more than stay in sync with the code as documentation: they are **consumed at runtime to produce the metrics.** The cold-start numbers land as typed fields on the deployment-topology model — no spreadsheet in sight — and a tool reads the model to derive the longest-path worst case. The coverage flagship walks the traceability graph, follows each model claim to its anchor, and measures *that*, rather than scanning source text for functions that look important. In both, the model is not a picture of the machinery. The model *is* the machinery — the thing the metric tool reads to know what to measure and how to weigh it.

The sharpest instance ties metrics back to mechanisms. A pre-commit gate has to decide whether a lint finding should block a commit. The naive gate blocks on any finding anywhere, which drowns an agent in pre-existing debt it did not cause. The better gate grades each finding by its **distance from the change** in the typed component graph, and the grade decides the gate's behavior:

Learn more about this governance mechanism: model-graded finding severity.

- **HARD** — the finding sits on a file this commit touched, or is directly caused by a touched input. Block. This is the commit's own debt, and it fails closed.
- **SOFT** — the finding sits in the *same component* as a touched file, but not on a touched file. Report it, name it, ask the agent whether the change plausibly caused it — but do not block. Blocking here would make concurrent agents race to fix the same pre-existing problem and collide.
- **SILENT** — the finding sits in a different component entirely. Suppress it at commit time; the whole-tree deploy gate remains the backstop.

The grade is a metric — a distance, read from the model at check time — and it drives a sensor's decision directly. Same finding, three different gate behaviors, chosen by consulting the component graph about how far the finding is from the change. Here the model earns its keep at the moment a commit is made: it is queried, live, to grade a gate — the machinery, run by the model.

2.5.7 A compact reference

Table 2.5-3 collects every metric — the decision each drives, where it sits on the hard-soft axis, and who reads it.

Table 2.5-3: The governance metrics at a glance — each metric, the decision it drives, its place on the hard–soft axis, and who reads it.

Metric	Decision it drives	Hard / soft	Read by
Per-service cost	Which service to slim, which call to batch	Hard	Cost tooling, admin panel
Cold-start latency (per service)	Which service warrants a warm-up lever	Hard	Perf tool, admin panel
Cold-start longest-path Σ	Which critical-path chain bounds worst-case latency	Hard	Perf tool, deployment-topology model
Model-Element Coverage	Which model claim (invariant, transition, edge) is untested	Flagship (hard number, semantic unit)	Coverage tool, per-model backlog
Entry-Point Coverage	Which anchored code root a refactor regression would slip past	Flagship	Coverage tool
Chain Coverage	Which requirement lacks an end-to-end verifying test	Flagship	Coverage tool, requirements audit
Doc-claim pinning	Which documented behavior still needs a test	Soft	Author, reviewing agent

Metric	Decision it drives	Hard / soft	Read by
Component-distance grade	Whether a gate blocks, reports, or suppresses a finding	Hard (grade), soft (the causation nudge)	Pre-commit gate, the committing agent

Read the table top to bottom and the spectrum is visible in one glance: the top rows are stopwatch-hard and machine-consumed, the middle rows turn a hard measurement onto a semantic unit, and the bottom rows aim a reader who still has to judge. Every row names a decision, because a metric that names no decision was measured too shallow — and a loop steering by a shallow number is a loop that cannot steer.

This table steers a single loop iteration. A mature Governed Engineering Environment is operated one level up too, through a small set of formative and summative metrics — a few you steer by while the work is in flight, a few you certify the result with at maturity, a few that serve both. The complete operator's reference, every metric with what it counts and its healthy direction, is collected in Appendix D.

2.6 When Guardrails Collide

The stack gives you four layers to steer at, and a governed fleet fills them: a hook at the harness that fires at turn-end, a lint at the application that gates a commit, a skill that aims a task, a mediator that rations a lock. Pick a layer, inject a constraint — and the constraints accumulate. Each earns its place alone, of course. But the more mechanisms you install, the more pairs of them touch the same thing, and together they form a system — and a system of controls has the one failure a list of controls cannot show: two of them making **incompatible demands on the same thing at the same moment**.

Two collisions from the case study make it concrete. In the first, one path required a commit to take a squashed shape while a separate check flagged out-of-brief changes on that *same* commit-set — two controls, one resource, contradictory demands, caught only when a real commit tripped both. In the second, two hooks fired on the *same* turn-end with no order declared between them, each able to block, resolved by hoping they composed. Neither collision lives in any one control's code. Each lives in the space *between* two controls, and that space was drawn nowhere — until Figure 2.6-1 draws it: two correct controls, one shared resource, a contradiction on the edge between them.

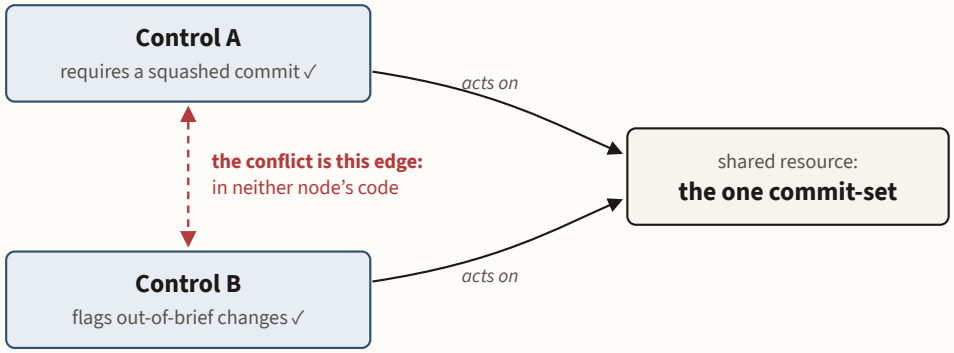


Figure 2.6-1: The Conflict Is the Edge. *Read Control A alone and it is correct; read Control B alone and it is correct. What they demand of the one resource they share collides — and that collision is drawn nowhere until the graph draws it.*

2.6.1 Drawing what lies between the controls

The Governance Graph draws it. The nodes are the mechanisms — each tagged with the event it fires on and the resources it reads, writes, or locks. The edges are the conflicts, in a closed four-value taxonomy: **contradiction** (two demand incompatible shapes of one resource), **contention** (two lock one resource, risking a deadlock cycle), **ordering** (one writes what the other reads on the same slot, with no guaranteed order), and **soft-versus-hard** (a soft aim overridden by a hard block on the same slot). The join key is the shared resource. Two controls that never call each other still collide if they both touch the turn-end slot, so the resource — not the call — is what makes them comparable. Type the turn-end slot and the context budget as resources too, and hook-pileup and injection-pressure fall under the same conflict machinery as a contended lock. One analysis covers all of it.

The shared resource is also what lets the graph reason *across lifecycles*. Most collisions are local — two controls in one lifecycle touching one nearby resource. But a hook that fires at turn-end and a check that fires at commit live in different lifecycles, and they still collide if they both reach for one typed

resource. The resource is the *only* coupling surface between lifecycles: controls in different lifecycles that share no resource cannot conflict, which is what keeps the graph sparse. And this is the semantic gap again — the collision is a property of the *interaction*, invisible in either control's own code, legible only at the level where both are drawn. The graph lifts the check to that level.

Learn more about this governance mechanism: Governance Graph.

A catalogue lists the controls; it can't show which two will crash into each other. The graph draws exactly those crashes: which pairs collide, and over what. It is the edges a list of nodes omits — the static catalogue's dynamic dual. And the two halves of that answer split cleanly. Some edges are **mechanically decidable**: a same-slot pair with no declared order, a lock cycle. A lint settles those. Others need **judgment**: are two constraints on a commit-set truly incompatible, or do they compose? No lint decides that; a reader must. So each edge carries its nature, derived from its type, and the decidable edges route to a sensor while the semantic ones route to a human prompt. This is where the motivating commit-set collision was really lost — the machine could flag that two controls touched the commit-set, but only judgment could call the contradiction. The graph makes both the flag and the prompt land *before* the control ships rather than at the collision.

That is the operational payoff. A governed environment is not just a pile of running mechanisms; it is a claim that those mechanisms compose. The claim was implicit, checked by collision. The graph makes it explicit and checkable: ask *what fires on this event, what touches this resource, is this new control consistent with the ones already wired* — and get an answer at authoring time. The fleet stops merely running its guardrails and starts modeling how they interact.

2.6.2 The other question a list can't answer

The graph settled one question a flat list could not: which two controls collide. A second question hides in the same blind spot. A list names every control the fleet installed, and the graph draws every pair that can crash — but neither says which *target* has nothing watching it at all. A governed estate grows toward the last painful failure, piling controls onto the target that just hurt while another accretes nothing, and that imbalance is a property of the whole set, invisible in any single control's code.

So the same control node-set carries a **third view**. The list is the controls that exist; the interaction graph is how two of them collide; the **coverage census** is how many watch each governance target, and which target has none. Classify every control by the target it guards, roll the set up per target, and a target with zero controls — or with only soft aims and no hard hold — becomes a first-class finding. The empty cell is not an absence nobody noticed. It is the estate's own blind spot, named, and pointing at where the next control should go.

This is a model whose *instances are the controls themselves*, keyed by target — a metamodel, in the layering the model zoo names M0 through M3, one level up from the models that govern the product. And it turns the book's own method on the governance system: the governance-conversion loop turns each recurring failure into a control; the census asks whether the portfolio that results is balanced across everything that needs governing. Governance audits governance. Run first on the real fleet, it read one target fully populated and two at zero — two of three complementary targets entirely un-watched, invisible until the roll-up made the gap queryable.

Learn more about this governance mechanism: control-coverage census.

2.6.3 Growing controls, and auditing the growth

The census has a counterpart, and the two are true opposites over one graph. The fleet grows its controls the reactive way this book argues for: let velocity expose a failure, convert each recurring one into a durable guardrail. That reflex *extends* the graph — toward the targets that already failed. A target that has never bitten draws no new control, so the reactive discipline, left to itself, leaves whole targets structurally un-watched. It is a feed-back loop with a systematic blind spot.

The census is the proactive audit that catches exactly that. One mechanism supplies controls, reactively, failure by failure; the other measures the supply, proactively, target by target, and names the target the reflex never reached. Soft-reactive-extends on the one side, hard-deterministic-measures on the other: the reactive extender grows the estate, the proactive census keeps the growth balanced across every target rather than only the ones that hurt.

And the census holds *itself* equal to the territory by the same discipline it enforces on everything else. Each control's target is **derived from its code anchor**, never hand-declared, so the map cannot drift from the controls it counts. A control the rule cannot place **fails loud** rather than dropping into a silent default bucket that would mis-credit the coverage. The roll-up never blocks a commit — it aims the next control, it does not gate — yet the honesty backstop beneath it is hard. The census even files its own anchor-drift guard, so the control set censuses the very governor that measures it.

2.6.4 When the hooks aren't yours

Every mechanism so far was one you wrote. But some hooks arrive from outside — a plugin registers one, a teammate adds a personal one, and, plausibly soon, a **regulator or a model vendor mandates** one: run your agents with these prescribed governance hooks, or you are out of compliance. The graph extends to these without changing shape.

An outside hook enters as a node with its firing event known (you can read *when* it fires from its registration) but its resource footprint unknown. Unknown is not empty; empty would mean "verified to touch nothing," and you have verified no such thing. So it sits on a *possible-conflict* edge against every mechanism sharing its firing slot, until someone analyzes it into a known node or flags it unanalyzable. That conservative default, *it might collide with everything on its slot*, is the honest reading of a black box, and it creates the right pressure: a mandated hook you have not reconciled shows up as unresolved conflict rather than as silent trust.

That makes the graph an instrument for *composing governance you did not author*. Satisfying one regulator is checking that your mechanisms plus theirs leave no unresolved conflict. Satisfying several at once is the same check over the union: the composition is compliant exactly when the merged graph carries no contradiction, and a contradiction between one regulator's mandated hook and another's is a compliance impossibility, surfaced at authoring time instead of at audit. Because the graph types soft-versus-hard, it can settle the property a mandate cares about most: that a required *hard* block is never quietly overridden by one of your own *soft* aims on the same slot. Your own guardrails were the first use; governance handed to you (by a vendor, a regulator, a standard) is the same graph with more nodes, and "do I satisfy all of them at once?" becomes a question the model answers.

2.6.5 Keeping the model honest

Two disciplines keep the model honest, and both are the book's. Anchor each node to the code that implements its control by symbol, never by line, so a drift lint re-resolves the anchor on every build and a control wired but unmodeled reddens the gate: the map stays equal to the territory, or the graph lies about which collisions are covered. And keep the model **descriptive** — it draws the controls that exist and checks proposed ones on request; it does not invent conflict-lints for collisions that have never happened. A governance graph is itself a governance mechanism, and the failure mode of governance is a tower nobody wants. This one confines itself to the collisions that actually happen.

> Learn more about this governance mechanism: symbol-anchored traceability graph.

This Part built the governed environment whole: the four layers you reach into, the soft aims and hard blocks that fill them, the metrics and lifecycles that run it, and the graph that keeps the guardrails from colliding. Every one of those checks the code against something it assumes already sits there — a schema, an invariant, a map of the controls. The environment enforces, but it does not supply what it enforces against. That is the model, and the next Part opens the zoo of them.

Part 3

The Model Zoo

3.1 The Executable Zoo

The previous chapter argued that a model is the sweet spot between prose too vague to enforce and code too verbose to reason over. It ended on a promise: this book would walk each kind of model on real code, with real invariants, and show how to keep the code from drifting away from them. This Part cashes that promise.

Recall the system behind all of this. There is a real codebase under every model that follows — DocAble, the production accessibility service this book runs on. It carries nineteen real, structured, drift-checked models and uses everything in this book to stay coherent. The full account is at the back, in The Built System — Part 5 (A MAGE Case Study).

> ***The Executable Model Pattern***
> Everything after this chapter is one schema, worn five ways. Every executable model answers the same five questions:
>
> - **What engineering question** does it let you settle that the raw code cannot?
> - **What invariant** must hold?
> - **How is it derived**, from the code or the code from it?
> - **How is that checked** on every build?
> - **What drift** does the check prevent?
>
> Read the rest of Part 3 as instances of it.

3.1.1 Why models are newly central

Part 1 derived the Modeling Thesis from the machine. This Part earns it: here are the executable models themselves and the work each performs. Read as economics, the thesis runs three ways at

once — agents **need** models (a raw codebase is too large for the window and too diffuse to reason over; the model is the compact representation the reasoner can hold), can **maintain** them (regeneration and reconciliation are the cheap, repeated cognition the substrate supplies — the next section prices this against the tradition that died for lack of it), and can **enforce** them (model-aware checks at the action and admission boundaries make a model a surface the build holds the code to, never advice).

Those three legs make a model more than a context optimization. One artifact serves as the agent's reasoning interface, the engineer's architectural specification, the generator's derivation source, and the build's assurance surface — four roles no prose document and no raw codebase can hold at once.

The architecture that delivers all four roles is the **executable source-of-truth**. The model is structured data. Agents read it to navigate and reason; generators consume it to emit the artifacts that must agree with it; drift gates block the build the moment code and model diverge. Every view in this Part is that one architecture, worn five ways.

3.1.2 The expense that buried MBSE is the one a fleet now pays cheaply

Model-based systems engineering never failed on its ideas. It failed on its upkeep. A model earns its keep only while it matches the code, and keeping a hand-drawn model matched to a moving codebase is human work — a person notices the code changed, finds the diagram it touched, and edits the diagram to agree. That labor is dull, unending, and the first thing a deadline cuts. So the diagrams drifted, then lied, then got ignored, and a discipline with sound ideas earned a reputation for ceremony that never paid its way.

The agent changes exactly that term. Re-inducing a model from the code, diffing it against the last version, regenerating the artifacts it feeds — this is the dull, unending work a fleet does for nearly nothing. The one expense that buried MBSE is the one thing an agent fleet supplies cheaply. A practice that never repaid its upkeep suddenly pays: the map stays equal to the territory because the recurring upkeep has moved off the human payroll. Someone still authors the model, the reconciliation rule, and the gate — what shrank is the standing maintenance, not the design. This chapter earns that claim: every model below is drift-checked on every build, not by a human who might forget.

The move now is to stop listing kinds of model and start walking them. The companion catalogue holds nineteen real, structured, drift-checked system-models — the actual embodiments of every kind the last chapter named. Reconciled with the general model types a systems-engineering course teaches, the zoo runs to twenty-nine named models. Treat that number as evidence rather than ambition; it grounds a single claim: the zoo is large enough that one model is never the whole picture, which is exactly Philippe Kruchten's thesis about architecture.

What admits a candidate to the zoo is the formal definition from The Agent Stack: a deliberately reduced, machine-readable representation of system intent, structure, behavior, policy, or evidence — one that supports reasoning, derivation, or checking, and is mechanically maintained against the system it represents. Registries, graphs, manifests, state machines, typed records: each earns the word the same way, by naming what it represents and carrying the gate that holds it true. A structure that fails a clause — a dashboard nothing derives from, a diagram nothing checks — stays out, however useful it is.

Scope, and where to read next. *"Model-based systems engineering" here means* software*: the models are typed records, state machines, and the code's own structures, because the system the fleet reads and rewrites is pure software. Classical MBSE speaks SysML*[28] *— blocks, constraint blocks, verify-and-satisfy traceability — a vocabulary these models quietly borrow, since a typed invariant with a pointer to the check that verifies it* is *a constraint block in all but the notation. What they refuse is SysML's usual* form*: a model drawn in a tool beside the code. That is a second representation of what the code already is, and a second representation is a snapshot — it drifts the moment the code moves, which is the very failure a derived model exists to kill. Borrow the words where they name something; keep the model* bound to *the code — derived in either direction, gated either way — not drawn alongside it. And it stops at* discrete *software: a system with continuous or real-time dynamics wants a different toolkit — the timed and hybrid automata of Alur and Dill*[29] *(timed automata*[30]*, hybrid systems*[31]*) — out of scope here, and the place to read next if your system touches the physical world.*

A larger claim starts here and pays off in the back matter. Working at the model level, with an agent doing the lifting, buys three things a team usually has to trade against each other: fewer tokens spent, more velocity, and higher quality — together, and without a corresponding cost. I did not model that claim; I lived it, as the preface already told. For the data, see The Timeline and the Work → The fewer-tokens leg is the Modeling Thesis paying rent: a model binds intent to implementation *and* shrinks what the agent must hold in-window to act on it, which is the book's answer to churn, the failure that limits an agent fleet. That combination is the shape of what Fred Brooks called a silver bullet, and Brooks argued no such thing exists. The careful version of the claim, defended in the implications, is that the model level is the closest thing to one, because it changes the representation the agent reasons over rather than promising to make the hard part easy. Each model page below is a small piece of the evidence.

Data from the case study

The fewer-tokens leg is the one we can measure directly. In a controlled A/B pilot, an agent answered four navigation questions about the modeled codebase — locate a component, trace an invariant, resolve a tier, find an anchor — once with the MBSE navigation guidance loaded (ON) and once without it (OFF), and we counted the tokens each arm spent to reach the answer. Table 3.1-1 reports the four tasks. For the data, see The Executable Zoo → (preliminary)

Table 3.1-1: Token cost of four navigation tasks with the model ON versus OFF — the measured saving from working through a structured model instead of raw code.

Task	Type	ON	OFF	Saving
NAV-01 component	modeled	12,463	15,507	−20%
NAV-02 invariant	modeled	13,520	15,421	−12%
NAV-03 tier	modeled	8,975	50,419	−82%
NAV-04 anchor	modeled	11,396	22,629	−50%

[28] Wikipedia, "Systems Modeling Language," 2026, https://en.wikipedia.org/wiki/Systems_modeling_language.

[29] Rajeev Alur and David L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science* 126, no. 2 (1994): 183–235.

[30] Wikipedia, "Timed Automaton," 2026, https://en.wikipedia.org/wiki/Timed_automaton.

[31] Wikipedia, "Hybrid System," 2026, https://en.wikipedia.org/wiki/Hybrid_system.

Preliminary — a controlled pilot (N=4 modeled navigation tasks; 2 placebo arms pending).

The median saving is about −35%, at **zero accuracy loss** — all four modeled arms answered correctly on both ON and OFF, so the guidance bought cheaper answers, not different ones. The biggest win (NAV-03) resolved a tier in one tool-call with guidance where the naive path took twenty-two, and that gap states the rule: the more derivation the naive path must redo, the more the guidance is worth.

3.1.3 Kruchten's 4+1, promoted from a closing sentence to the spine

The last chapter invoked Kruchten's **4+1 views** in a final sentence — "the reason one model is never enough." This Part promotes that sentence to its organizing structure. Kruchten observed that no single diagram captures a software architecture, and that four complementary views plus a fifth that ties them together cover what a working engineer needs to see:

- **The Logical view** — the functional structure: the objects, the types, and how they relate. What the system *is*, as a designer sees it.
- **The Process view** — concurrency and synchronization: what runs at once, the locks, the runtime dynamics. What stays consistent under interleaving.
- **The Development view** — the module and organization structure: how the source is packaged, layered, and owned. How the code is arranged for the people and agents who build it.
- **The Physical view** — the mapping of software to hardware: where processes run, across which hosts and network boundaries. Where the parts live.
- **The Scenarios view (the +1)**: the use cases and journeys that animate the other four and validate them. What the system does for someone, walked end to end.

One chapter of this Part takes each view. A reader leaves with four views seen built on real code, plus the scenarios that validate them — each view with its invariants and the checker that holds them true. Figure 3.1-1 draws the one trunk and its five views.

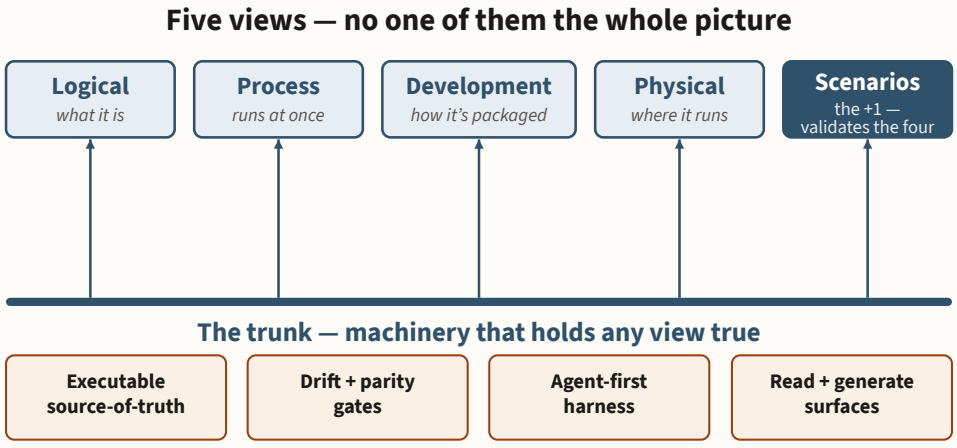


Figure 3.1-1: One Trunk, Five Views. *The trunk holds the shared machinery — executable source-of-truth, drift and parity gates, the agent-first harness — that makes every view executable and equal to the code. Five branches fan up: Logical, Process, Development, Physical, and Scenarios, the +1 that validates the other four.*

What this book adds to 4+1: the drift gate

Kruchten, of course, described the views; he did not have a drift gate. The drift gate is what this Part adds: every view here is not a diagram a human maintains but a structured model a tool reads on every build, with a gate that fails the build the moment the model and the code disagree. 4+1 tells you *which views you need*. The drift gate keeps each view from becoming a diagram that rots.

The framework is a choice; the core is not

Making the model executable is why this Part is not really *about* 4+1. Kruchten's five views are how *this book* organizes the zoo, because they map cleanly to an engineer's questions. But once the core is a structured model held equal to the code, the *framework* is a **projection over that core, not a second thing you maintain**. You keep one executable model and render whatever view-set the audience needs from it.

Change the audience and you change the projection; the model stays put. An engineer doing runtime analysis wants Kruchten's Process view. A customer asking "what are the moving parts, and what do you send my data through?" wants a C4 Context-and-Container view. A reader reasoning about invariants wants SysML's constraint-block vocabulary. A compliance reviewer wants a data-flow diagram. Rival methodologies, at first blush — yet each is a rendering of the same structure, and a system whose core is executable can derive any of them: the C4 view is the same projection move as the Process view, run over the physical/deployment model instead of the state machine.

[32] J. Nathan Foster et al., "Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem," *ACM Transactions on Programming Languages and Systems* 29, no. 3 (2007): 17:1–17:65.

The same move works one level finer. Even inside a single framework, a view must choose *what to show*: a customer's Container view hides the internal helper mesh a maintainer needs; a privacy review shows only the data stores and where they egress. That choice is a second projection, and it stays honest by a clean split. The **content** — which containers, edges, and external systems exist — is a pure function of the source models, authored by no one. The **lens** picks which of that content a view shows; it may hide a node, but it can never invent one. Name a container the sources no longer contain and the lens fails the same drift gate the view does, so even "the customer's slice" cannot drift from the truth. The word *lens* arrives with its theory attached. The programming-languages literature formalizes exactly this hide-but-never-invent contract as a *well-behaved lens*, one half of a bidirectional transformation[32] the drift gate checks at build time what the lens laws state as algebra. Figure 3.1-2 draws both projection axes — framework and lens.

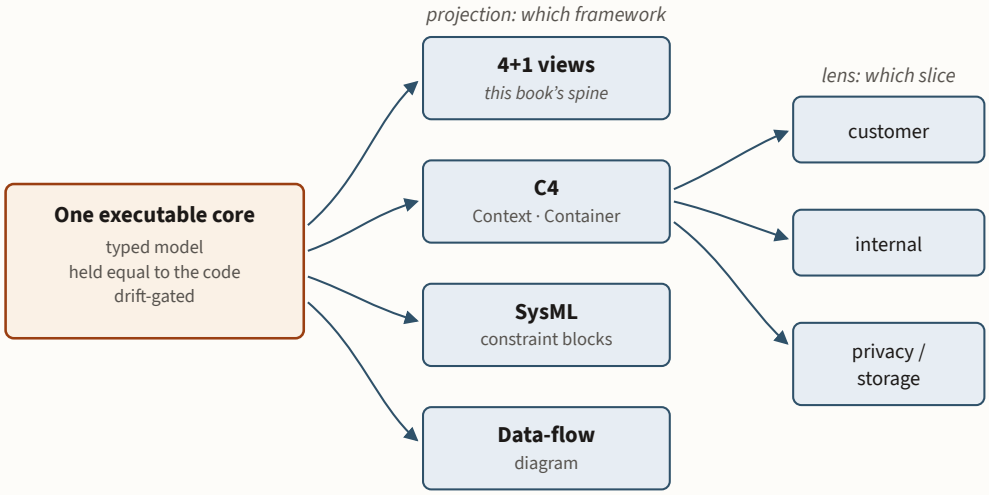


Figure 3.1-2: 4+1 is one projection among siblings; a lens is a projection within one. Choosing a framework, or a slice within it, is a rendering decision over a single core — never a second model to maintain.

What this Part teaches, then, is *modeling* in the general; 4+1 merely supplies the spine, chosen because it maps cleanly onto an engineer's questions, and each view gets built and drift-gated on real code. The transferable claim sits one level up: hit the executable core, and the framework wars dissolve into projection choices. Choose the core with care; choose the framework, and the lens, to fit the reader.

3.1.4 The trunk: machinery that holds any view true

Before the views, one chapter of framing. Most of the catalogue's method models do not belong to a single view. They are the machinery that makes *every* view executable and drift-proof, and rather than re-explain the drift gate in each view chapter, this chapter introduces it once and each later chapter revisits it.

- **The executable source-of-truth.** The pattern behind every view, named and defined at the top of this chapter.

Learn more about this governance mechanism: executable source-of-truth.

- **The drift and parity gates.** The lints and tests that enforce model-and-reality parity in both directions, so no model drifts unilaterally.

  Learn more about this governance mechanism: drift and parity gates.

- **The agent-first harness.** Build the models as a thin hand-rolled layer over frozen records under a few disciplines: adopt the genre's schema, skip its heavyweight runtime.

  Learn more about this governance mechanism: agent-first MBSE harness.

- **The read and generate surfaces.** One canonical query API over every model (read-don't-hardcode consumption), and generators that emit real artifacts *from* the models, each provenance-headed so a hand-edit is caught.

  Learn more about this governance mechanism: model query surface.

These are the framing chapter's home. The view chapters name them again where each view's model reconnects to them. A few concepts have to land before the views begin: what an invariant is and how you write one, what model checking buys you, and what drift is. Each is introduced at the point it is first needed, and later chapters link back rather than repeat it.

Three beats frame the whole Part.

Beat 1 — a model is derived and joined, never repeated

The last chapter warned against repetition: two models that say the same thing must now be kept in sync. This Part sharpens that warning into the zoo's organizing rule. A model here is **derived** — from the code, or the code from it — never kept beside the code as a parallel document. And it **joins** to its neighbours on a shared key rather than restating them.

The service-flow model *is* the list the access policy is generated from and the call graph is checked against — it holds no second copy of the endpoints to re-list. The journey model *joins* to those endpoints by call-site rather than copying them. Repetition is the failure. Derivation and join are the cure. The flagship demonstration is the third worked example in the Scenarios chapter, where four models join so a journey's criticality reaches all the way to which host a test runs on, with no fact stated twice.

Beat 2 — name the direction of derivation

A model and the code it concerns stay equal by derivation, and derivation runs in one of two directions. Say which up front, because the machinery differs.

- **Model-from-code (induce and reconcile).** The code is authoritative; the model is a projection of it, reconciled at build time. The component-zone model reads the real directory tree; a journey's dependency list is derived from its real call sites. Here the drift gate is a *reconciler*: it re-induces the model from the code and fails on divergence. Reach for this when the code is the ground truth and the model is a compact, queryable view of it — the usual case for a large codebase an agent must operate.
- **Model-to-code (specify and generate).** The model is authoritative; the code, config, or docs are generated from it. The access policy, the service catalog, the wire-contract types are emitted from

the service-flow model. Here the drift gate is a *freshness and provenance check*: the generated artifact carries a header, and a hand-edit or a stale regeneration is a finding. Reach for this when the model is the design intent and you want the artifact to obey it.

Most models in the zoo are model-from-code, because the codebase pre-existed the models. The service-flow model is the clearest model-to-code case, and it is bidirectional — some fields generated to code, others reconciled from it — which is why its gate checks both directions. Each model page names its direction.

Extracted Code Graphs and Intent-Bearing Models

The purest model-from-code artifact is not one of the zoo's own views. It is the repository graph a growing class of tools now hands a coding agent, so the agent stops rediscovering the codebase from raw files on every task. A parser walks the repository and records its entities and the relations among them: functions, classes, imports, calls, endpoints, schemas, tables, and the documentation links between them. The result is a persistent, queryable map. What calls this function? What depends on this module? Which path connects this endpoint to that table? What moves if this node changes?

That map earns its keep. It is richer than a flat symbol index and far cheaper to traverse than the whole tree. By the Modeling Thesis, that is the token argument already made for the zoo, pointed outward: the agent reasons through the derived graph instead of re-reading the code. Recent systems such as RepoGraph[33], CodexGraph[34], RANGER[35], and Codebase-Memory[36] derive such graphs for navigation, dependency traversal, and agent retrieval; Codebase-Memory reports roughly a tenfold reduction in tokens and 2.1 times fewer tool calls against a file-exploration agent, at some cost in answer quality (83 percent against 92). These systems give an agent an increasingly capable model of the artifact. MAGE's separate concern is how that descriptive structure joins to authored intent and mechanically enforced policy.

Here is where the join bites. An extracted graph describes what exists. It can show that component A calls component B. Extraction alone cannot say whether A is *permitted* to call B, whether B owns that responsibility, which user goal the edge serves, or what invariant the call must preserve. None of those are syntactic facts waiting in the artifact to be recovered. An engineer decided them.

Beat 2 named one axis: which artifact is authoritative, code or model. Lay a second axis across it. Not how the representation was made, but what claims it is authorized to make. A descriptive model recovers the territory. An intent-bearing model states the shape the territory is meant to have. The first is built to navigate and to estimate impact. The second is where specification, prediction, and governance live.

The boundary bends both ways. A MAGE model may begin as an induced description of existing code, and a code graph may fold in documentation, schemas, or inferred clusters. So provenance is not the question. Authority is. Does the representation report that an edge exists, or declare the edge permitted? Does it list the running services, or name which service owns a capability? Does it record observed behavior, or define the transitions the system is allowed to make?

---

[33] Siru Ouyang et al., "Repograph: Enhancing AI Software Engineering with Repository-Level Code Graph," 2024, https://arxiv.org/abs/2410.14684.

[34] Xiangyan Liu et al., "Codexgraph: Bridging Large Language Models and Code Repositories via Code Graph Databases," 2024, https://arxiv.org/abs/2408.03910.

[35] Pratik Shah et al., "RANGER: Repository-Level Agent for Graph-Enhanced Retrieval," 2025, https://arxiv.org/abs/2509.25257.

[36] Martin Vogel et al., "Codebase-Memory: Tree-Sitter-Based Knowledge Graphs for LLM Code Exploration via Mcp," 2026, https://arxiv.org/abs/2603.27277.

Put that way, the two kinds of model compose rather than compete. A code graph finds every caller of a raw PDF library. An authored architecture model declares that a single structured model owns mutation of that format. A gate compares the extracted callers against the permitted set and turns any extra edge into a build finding. The graph supplies the current fact, the authored model supplies the policy, and the gate reads one against the other. This is the same drift check the zoo runs everywhere, now with a descriptive map on one side and a normative model on the other.

That composition is a governed engineering environment in miniature, and it settles where these tools sit. A code knowledge graph is a strong input to MAGE: it can serve the observed-reality side of any drift or parity gate. What it does not carry by itself is the *ought*. The Modeling Thesis puts both representations in front of the reasoner, the map of what is and the model of what should be. The Alignment Thesis keeps the descriptive map, the intended design, and the running code from drifting apart in silence.

A shorthand, offered as a center of gravity and not a fence: a code knowledge graph compresses implementation; a MAGE model compresses engineering meaning. Both are models. Both can be induced. Both help an agent reason. What MAGE requires past the graph is a place for the ought, and a mechanism that holds the is against it.

The ought can also be authored a simpler way: by writing it down. Spec-driven development gives every meaningful feature a written specification before any code, and has agents implement against it, the spec standing as a contract between intent and code. It is a real move toward authored intent, the same normative commitment MAGE makes, and it holds until the corpus grows. Recent studies of spec-driven agent workflows converge on the same normative point: an industrial one-person-squad case that ran an explicit spec-driven process to deliver work scoped for a four-person team reports that specification quality and institutional knowledge — not raw model capability — decided the outcome[37]. One team's field report is exact about where it broke[38]. Their naive preload reached roughly thirty-five thousand tokens at startup, of which about six percent was relevant, and the agent degraded once the window passed half full. Size was not the worst of it. A specification the code had moved past produced *"syntactically correct code that was wrong,"* because the agent trusted it. *"A stale spec is worse than no spec, because the agent trusts it."*

Read against the two theses, that report is convergent evidence, not a rival. Its remedies climb the same hill this chapter has been climbing. Living specs, one per feature and edited in place, are the derive-don't-snapshot rule under another name. A thin router and index that load only the matching documents reach for a context-management layer, the answer to loading the right small slice instead of the whole corpus. Generated wikis and a CI check that flags a doc-against-code mismatch are a drift gate and derived traceability, bolted onto a prose corpus after the fact.

The difference is representation. A spec states the ought in prose, which does not compress and drifts unseen. A structured model states it in a form the machine reads: the right slice is small by construction, so the reasoner reasons through the model rather than through tens of thousands of tokens hoping the relevant fraction lands; and a gate holds the model against the code, so it cannot go stale-but-trusted, the failure the report names most sharply. The remedies spec-driven development reaches for by hand are first-class here.

---

[37] Marcelo Vilas Boas et al., "One Developer Is All You Need: A Case Study of an AI-Augmented One-Person Squad in a Brownfield Enterprise," 2026.

[38] Abdurrahman, "We Adopted Spec-Driven Development, Then the Specs Started Eating Our Context Window," Medium, July 2026, https://abdurrahman5.medium.com/we-adopted-spec-driven-development-then-the-specs-started-eating-our-context-window-6cb9476dfedc.

So three ways to hand a fleet a system it can reason about fall on one axis. A code knowledge graph recovers what the code *is*. A spec writes down what it *should be*, in prose the agent must re-read and can outrun. A structured model states that same ought where a compiler and a gate can hold it to the code. Only the last keeps intent both compact and true.

Beat 3 — every model traces to the code

A model earns trust only if you can get from a model element back to the lines that realize it, and forward from a line to the model element that governs it. That round-trip is **traceability**. It is exactly what the safety-critical standards demanded and exactly what the drift gate mechanizes: the reconciler *is* a traceability check that runs on every build. The forward direction is itself measurable — how much of the tested code reaches any model — and driving that number up is how the case study drained its unmodelled code. For the data, see Metrics → (preliminary)

Traceability is why the per-model template folds a one-line note into each page — the direction the model derives, and the join key from a model row back to the code. A reader always knows how to round-trip from the picture to the lines.

The round-trip is also what the whole zoo rests on. A set of views earns trust only while every view still tracks the code it depicts; the moment one describes a subsystem the code has retired, it misleads every agent that reasons through it. A zoo of drifted views is worse than none, because it lends false confidence. Kruchten gave the reader four views and a fifth to validate them. He did not say how a view stays equal to the code once both have hundreds of authors and a fleet rewriting the territory daily.

The answer this book defends is that the round-trip must itself be **derived**, not snapshotted. Picture the graph it walks — model element to lint to code entry point to test to registry row, each hop an edge. Write those edges down once and trust them, and the graph is a frozen snapshot that drifts the first time a symbol it names is deleted. Give each edge the mechanism that re-proves it — resolve the target against the code and see whether it still exists — and a deletion turns the edge red. A check that re-reads the code cannot fall behind it; a stored copy can.

The graph pays a second dividend. Because a resolving edge means the map is current, an agent can *walk* it to pull the slice of a codebase it cannot hold in one window — up from a line to the invariant that governs it, down to the code that realizes it. The systems model a senior engineer keeps in their head becomes a navigable index the fleet reasons through. Drift-detection and navigation are one property wearing two faces: an edge that resolves is both a true claim and a current map. The next section reads the concrete mechanism off DocAble's real drift; the catalogue carries it as its own entry.

Learn more about this governance mechanism: symbol-anchored traceability graph.

3.1.5 Keeping the models honest — the traceability substrate

DocAble did not design this up front. It tried the cheaper approach first, watched it fail on real work, and built the machinery from what broke. The story is worth walking, because it shows the exact shape a snapshot rots into and why a derived edge does not.

A model is only as good as its agreement with the code. Say a model still describes a *poll-based* dispatch plane after a serverless *push* plane replaced it — DocAble's real mid-project migration — and

every agent that trusts the model plans against a system that no longer exists. The map has to equal the territory, or the map lies.

The tempting way to keep them equal is a checklist: when you retire a subsystem, remember to update the model too. DocAble tried that and it failed the way it always fails. A cutover retired the poll plane, the retirement doc dutifully reconciled the architecture docs and the agent boot context, and it forgot the models. The forgotten edge stayed green because nothing re-checked it. That is the pattern under every silent drift: a hand-maintained list rots the moment the code moves without it, and no one notices until the stale fact bites.

Reading the fix off real drift

The right sensor was read off real drift rather than guessed. The models were left deliberately ungoverned for weeks and the fleet allowed to drift them, so the fix could come from observed failure instead of imagination. An audit of the closed work — every job green at its own done-check — turned up two dozen silent divergences at a signal-to-noise near one. A checker tightened past the producer it now rejected; a typed function shipped and wired to nothing; a reconciliation lint gone red the instant its work closed. Sort those cases by what caught them and one line separates the clean from the drifted. The clean ones had a sensor that re-read the code at check time. The drifted ones kept a second copy of the fact, maintained by hand, that fell behind.

So DocAble keeps a **traceability graph** in place of a checklist, and every edge in it is *derived* — re-checked against the code at scan time rather than trusted from the last time someone looked. The nodes are things the system already has: a model element, the lint that enforces it, the code entry point that realizes it, the test that verifies it, a registry row, a doc surface. The edges are the joins between them — *this invariant governs that code root*, *that test verifies this invariant*. Each edge carries the mechanism that re-proves it: run the resolver against the target and see whether the symbol still exists.

Two decisions that make the graph resist rot

- **Anchor to symbols, never to line numbers.** An edge points at a function or class *definition*, resolved statically — not at a file-and-line. Line numbers churn under every edit above them; a symbol survives every edit that does not delete it. When five journey anchors drifted stale under a refactor, the cause was line-anchoring. A symbol anchor moves with the code it names.
- **A missing symbol is a finding, not a crash.** When an edge points at a symbol that no longer resolves — deleted, renamed, moved out — the edge goes red, and that red is the drift becoming visible. The sharper case: when the model wants to point at logic for which *no clean symbol exists*, buried in a god-function, the absence signals the code is missing an abstraction the model needs. The anchor doubles as a probe for where the code is under-factored.

The sensor that walks the graph runs at a slower cadence than the per-commit gates, on purpose. Resolving a symbol against the code — a jedi or Roslyn round-trip per anchor — costs far more than the cheap membership lookups a commit hook can afford, and the costlier the check, the slower the cadence it earns: the derived-edge check fires at review and done-checking time, not on every commit. A second, cheaper guard sits on the anchors themselves, watching for the case a symbol walk cannot catch — a pointer left naming a dead implementation while a replacement was built beside it. That pointer-drift guard was earned by the most expensive drift of the build, a retired *deploy-orchestration script* whose stale pointers still named a dead implementation and fired a prod-blocking deploy.

This is the same commitment as the one structured model under every format, one level up. There, a fix applied through the model holds for every format because there is exactly one place to apply it. Here, a model stays true to the code because the join between them is re-derived, not remembered. In the system that underpins this book, the audit's line held without exception: **derived sensors defended; snapshotted ones drifted.** A sensor that reads the source of truth at check time cannot fall behind it. A hand-maintained copy of the same fact always can.

3.1.6 How to read a model page

Every model in this Part is rendered by filling the same five-field template. The uniformity is deliberate: a reader who has read one page can read any of them on reflex.

- **(a) Quality property it helps assess.** The property this model lets you *check* — stated as the question the model answers that the raw code cannot. Not "it models X" but "it lets you ask, and answer, *is Q true?*" A model with no quality property to assess is a diagram, not a model.
- **(b) Constructs and relations.** The typed elements the model is built from and how they relate — the record kinds, the fields that carry meaning, the edges between them.[39] This is the reference content: what a reader consults to understand the model's shape.
- **(c) Visual depiction.** The diagram of the real model, matched to its natural type — a state machine gets a state diagram, a topology gets a deployment diagram, a registry of related records gets an ER diagram. Each carries an accessible description that states the content *and* the takeaway.
- **(d) Invariants, and how they are checked.** The predicates that must hold, each with the mechanism that holds it true — a drift lint, a coverage floor, a model check, a parity gate. Rendered as a table of invariant, its temporal shape, and how it is checked. A model whose invariants are prose with no checker is flagged UNTESTED.
- **(e) Traceability and derivation direction.** One line: which direction the model derives, and the join key from a model element back to the code.

The first four fields map onto the four questions a reader brings to a model: what is it for, what is it made of, what does it look like, how do I trust it. Field (e) answers the fifth this chapter raised: how does the model stay tied to the code. Each view chapter opens with the Kruchten view it embodies, then walks its models through this template.

3.1.7 The coherence triangle

Every model page in this Part draws the same shape. Fields (b) and (c) give the **structured model**: records, edges, state machines. Field (d) adds the **invariants**, the predicates that must hold, each named by a stable id. Its checker supplies the **coherence-gate**: the drift lint or model check that holds a predicate true at build time. Model, invariants, gate — the constraint-and-verify vocabulary the chapter already borrowed from SysML, where a constraint block wires to the check that satisfies it.

The model provides two kinds of value. First, the schema — records, edges, state machines — names the parts and gives the tools something to read. Second, the invariants that get checked: the predicates that must hold. Value both. The triangle's point is that the invariants and their gate *complete* the model's value, not that the schema earns nothing. A structure that names its parts but promises

[39] The modeling tradition calls this field the model's metamodel, and names the full layering M0–M3: running instance, model, metamodel, meta-metamodel, bottom to top. We keep the layer and skip the tower — a frozen record schema is metamodel enough, and the host language's type system stands in for M3. The MDSE text (Brambilla, Cabot, and Wimmer, ch. 2) draws the tower in full.

nothing you can test is only half a model; field (a) drew that line, and field (d) stamps an unchecked invariant UNTESTED. Draw the schema and stop, and it rots the first time the code moves past it. The gate keeps it honest.

The invariants are not arbitrary. They come from the failures the system met. In the DocAble build, the gaps the models caught were violated-but-unstated invariants — a property the system had always needed and never written down, promoted to a checked predicate the moment it broke. Part 5 records the act behind each one: deciding, for every failure, whether it was a one-off to patch or the symptom of a missing mechanism. Promoting the property to a checked invariant is what makes the model the right abstraction, not merely a plausible one. Figure 3.1-3 draws the three nodes and the edge that closes them.

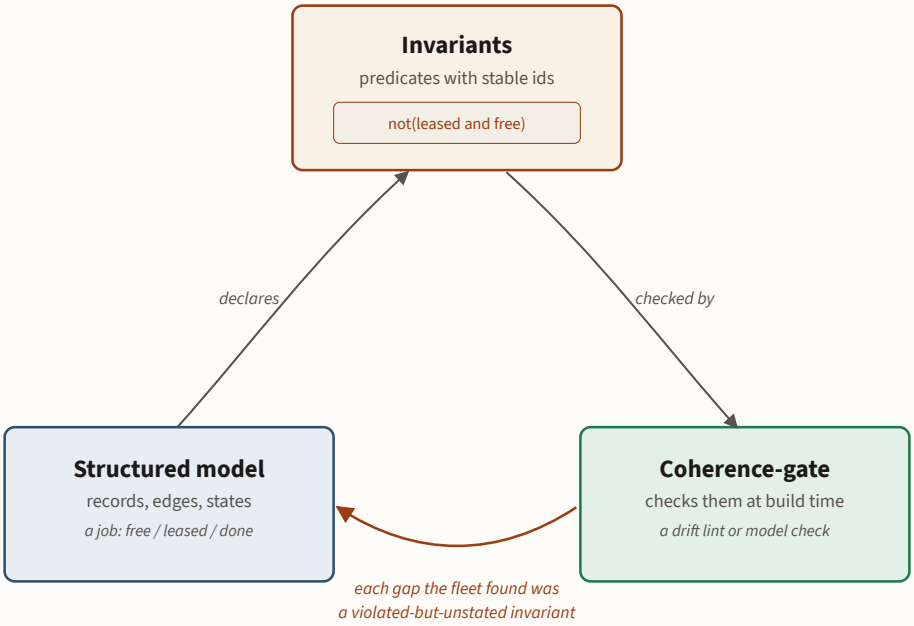


Figure 3.1-3: A model's value is both its schema and its checked invariants: the structured model declares predicates with stable ids, and a coherence-gate checks them at build time. The closing edge carries the proof — each invariant was earned from a failure the system never wrote down.

—

Inset I1 — What is an invariant, and how do you write one?

An **invariant** is a statement that must be true in *every* state your system can reach — a property the system promises never to violate. You write one as a predicate over the model's state, then you ask the model whether any reachable state breaks it. Take a job that can be `free`, `leased`, or `done`. The invariant "a job is never both leased and free" is a predicate over one state: `not (leased and free)`. Because it concerns a single state, you check it by visiting every reachable state and evaluating the predicate there. If none breaks it, the invariant holds. The move that makes this more than a comment: the predicate is *checked*, exhaustively, by a tool, not eyeballed on a diagram. That is the step from "we believe this holds" to "we proved it holds."

```
STATES = {"free", "leased", "done"}
  def invariant(state): return not (state == "leased" and state == "free")   #
never both
  assert all(invariant(s) for s in reachable_states())   # check every
reachable state
```

Inset I8 — Derivation direction, boxed for reference

Two directions, from Beat 2. **Model-from-code**: the code is authoritative, the model is a projection reconciled from it, and the gate re-induces the model and fails on divergence. **Model-to-code**: the model is authoritative, the code or config is generated from it, and the gate is a freshness-plus-provenance check that catches a hand-edit or a stale regeneration. Reach for model-from-code when a large codebase pre-exists the model; reach for model-to-code when the model is the design intent you want the artifact to obey. A model can be both at once, on different fields.

Inset I9 — What is drift, and what is a drift gate?

Drift is the condition where a model and the code it describes disagree — the map no longer matches the territory. A human-maintained diagram drifts silently the first time a developer changes the code and forgets the diagram. A **drift gate** is the build-time check that set-diffs the model against reality in both directions and fails the build on any divergence. Model ⊆ reality catches a model element with no code behind it; reality ⊆ model catches code the model never declared. Because the gate runs on every build, drift cannot accumulate: the first divergent commit fails, not the fiftieth. This is what makes an executable model different from a diagram — the diagram *can* drift, the gated model cannot.

The remaining insets appear in the view chapters where they are first needed: assertions over sequences and the temporal-logic step-up, bounded model checking, automata, data-flow diagrams, safety versus liveness, protocols and TLA+, and coverage over model nodes.

That is the trunk. The rest of this Part is the branches. This chapter built the shared machinery — the executable source of truth, the drift gate that fails the build the moment model and code diverge, the derived edges that keep the map equal to the territory — and set the admission test and the five-field template every model below obeys. What remains is to walk the five views on real code, each with its invariants and the checker that holds them true. The Logical view is first: what the system is.

3.2 The Logical View

The Logical view is the functional structure — the objects, the types, and how they relate. It answers the designer's question: *what is this system, as a set of concepts?* Before you ask what runs at once, or where the parts live, or how the source is packaged, you ask what the parts *are*. That is the logical view, and it is the one most engineers already draw without naming — the boxes-and-arrows sketch of the domain. Held as a drift-checked model rather than a sketch, it is the representation the fleet reasons through to know what the system is: the Modeling Thesis at the grain of one view.

Two general model types live here. A **data-flow diagram** traces what data exists and who may touch it. An **inheritance hierarchy** shows which specializations of an abstract concept exist. Two of the catalogue's real models embody the view on live code: the service-flow model, which is the functional structure of a service-oriented architecture, and the domain registries, which are the typed facts that functional model rests on.

Learn more about this governance mechanism: service-flow model.

Learn more about this governance mechanism: domain registries.

> ***Inset I5 — What is a data-flow diagram?***
> A **data-flow diagram** traces *data* moving through transforms — a source, the stages that reshape it, and the sink — where each edge names *what flows across it*, not *what decides*. It differs from a control flowchart: a flowchart's diamonds branch on decisions; a data-flow's edges carry a document, a JSON blob, a rendered page. It is the natural lens for **security and privacy**, because the question there is exactly "where does this data go, and who may touch it on the way?" Draw it left to right, label each edge with the data on the wire, and a reader sees the path a piece of data takes and every component that handles it.

An **inheritance hierarchy** — the second logical-view type — shows an abstract concept and its concrete specializations, so an engineer sees which extensions exist and where a new one would attach. Think of a remediation tool's hierarchy: the abstract "document format" at the root, and its concrete leaves — the slide format, the word-processor format, the spreadsheet, the PDF — each carrying the same operations against a different file model. The hierarchy is the logical statement of *what kinds of thing the system knows how to handle*, and it is where a reader looks to answer "can this system be extended to a new format, and what must the new leaf provide?"

The two real models below carry the view. The data-flow and inheritance types are the vocabulary; these are the worked embodiments.

3.2.1 The service-flow / API model

The typed source-of-truth for a service-oriented architecture — its services, their endpoints, their authentication, and the wiring between them — from which the access policy and environment wiring are generated and against which the real call graph is checked.

(a) Quality property it helps assess. Three, each a question the scattered deploy config and handler code cannot answer without a single model.

- **Wiring correctness**: *does every declared caller-to-callee edge have a real call site, and every real cross-service call a declared edge?* An undeclared call is an ungoverned one.
- **Access-policy soundness**: *does the generated network access policy match the wiring the model declares?* The policy is emitted from the model, so a service can reach only what the model says it may.
- **Contract parity**: *does each endpoint's declared request and response shape match the handler that serves it?* A drifted contract is a runtime failure the model turns into a build failure.

(b) Constructs and relations. A typed catalog in the dialect of a service-catalog schema, adopted for its hard-won structure and read by the project's own tools.

- `Service`: one deployable unit, with its name, its owning layer, and the endpoints it serves.
- `Endpoint` — one API surface on a service: its path, its auth requirement, and its declared request and response contract.
- `Wire`: one declared caller-to-callee edge, joining a calling service to a called endpoint. The set of wires is the graph the access policy is generated from and the call graph is checked against.

(c) Visual depiction. The natural diagram, Figure 3.2-1, is a data-flow — the model on the left, the generators and parity gates it feeds on the right. Reused from the model's appendix Structure slot:

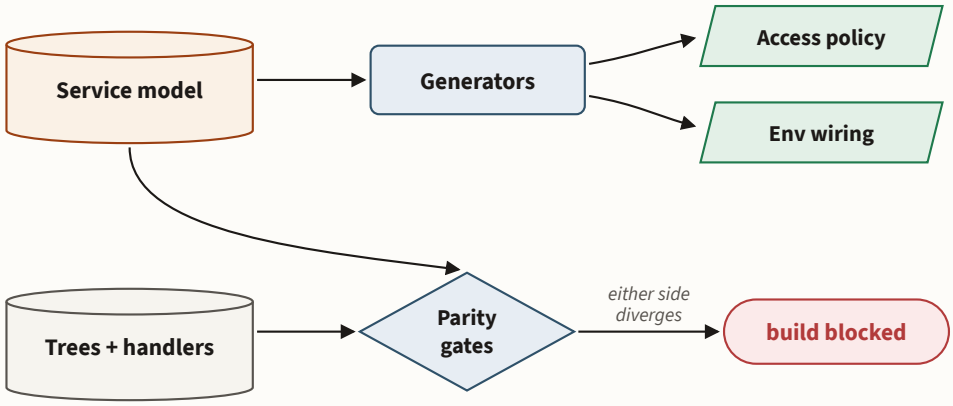


Figure 3.2-1: The service model feeds generators and parity gates. It is authoritative for what it generates and reconciled against reality for what it checks; either side diverging blocks the build.

(d) Invariants, and how they are checked. The checkers are the trunk drift-and-parity machinery, pointed at the service graph, collected in Table 3.2-1:

Table 3.2-1: Invariants of the Logical view — each with its temporal shape and the check that holds it.

Invariant	Temporal shape	How it is checked
Every declared wire has a real call site	$\Box P$ (safety)	Call-graph parity lint: each `Wire` edge must resolve to a real cross-service call.
Every real cross-service call is a declared wire	$\Box P$ (safety)	Same lint, reverse direction — an undeclared call is a finding.

Invariant	Temporal shape	How it is checked
The generated access policy matches the wiring	□*P* (safety)	Freshness lint over the generated policy: regenerate from the model, diff against the committed artifact.
Each endpoint's contract matches its handler	□*P* (safety)	Contract parity check joining the `Endpoint` shape to the handler signature.

(e) Traceability and derivation direction. *Bidirectional.* The access policy and environment wiring are model-to-code — generated from the declared model. The call-graph parity runs model-from-code — it re-reads the real call sites and reconciles. The join key from a model row to the code is the endpoint path, which indexes both a `Wire` edge and the handler that serves it.

Also seen in: Physical (the generated access policy is a placement artifact). Rendered in full here; referenced from the Physical chapter.

3.2.2 The domain registries

The domain registries are the frozen tables the service-flow model rests on: the file types the system handles, its known conformance gaps, its scheduled jobs, its user-facing surfaces. Each is the single source of truth for its slice — the competitor doc, the coverage table, and the code that switches on file type all read the same frozen registry, so they cannot disagree. The quality property is **fact consistency**: every place that presents a domain fact agrees, because each joins to the registry by key rather than keeping its own copy. A copy is where drift starts, so copying a registry value into code is itself a finding; the consumer must read the fact live or be caught. That turns "keep the docs in sync" from a discipline into a build failure. Its full construct-and-invariant treatment is in the appendix, and one slice — the governance rules' own metadata — is walked in full in the Development chapter as the rule-metadata registry.

—

The Logical view names what the system is. Next, the Process view: what happens when many of the parts run at once.

3.3 The Process View

The Logical view named the parts. The Process view asks the question the logical view cannot: what happens when many parts run at the same time? Concurrency turns a correct-in-isolation design into a source of races, deadlocks, and torn writes, and the more parts run at once, the more interleavings the design must survive. The Process view is the lens for that — the states a system moves through, the locks it takes, and the invariants that must hold across every one of those interleavings. Those invariants are not comments: a checker holds the running system to them, so a race cannot reach production unseen — the Alignment Thesis, read at the grain of concurrency.

This is the chapter where invariants stop being predicates over one state and become predicates over an *order of events*. That step-up pulls in a cluster of ideas: what an automaton is, safety versus liveness, bounded model checking, and the temporal-logic step from a state assertion to a sequence assertion. This chapter defines each as it is first needed, and The Book's Language collects them for quick reference. Two general types live here — the **state machine** and the **automaton / formal invariant** — and four real models embody the view: formal invariant verification as the verification backbone, the synchronization model, and the mediator and single-writer registries.

Inset I4 — What is an automaton / state machine?

An **automaton**, or **state machine**, is a finite set of states and the labeled transitions between them. It models computation as "which state am I in, and which moves are legal from here." A microwave is *idle*, *cooking*, or *door-open* pressing start moves idle to cooking, opening the door moves cooking to door-open. The value of drawing it: the illegal states and the missing transitions become visible. If a path leads back to *cooking* while the door is open, the diagram shows you the way to cook yourself. A state machine is the logical spine the Process view's invariants attach to — the invariant is a predicate over the states, and the checker walks the transitions.

Inset I2 — Assertions over STATE vs over SEQUENCES

Some properties are about a **single state**: "a job is never both leased and free." You check one by visiting every reachable state and evaluating the predicate there (the invariant primer). Other properties are about **order over time**: "a preempted job eventually re-runs." No single state can be inspected to settle that — it is a claim about the *sequence* of states, that a good state is always eventually reached. Claims about order need **temporal logic** (LTL, linear temporal logic), which adds operators over sequences: *always* (□), *eventually* (◇), and *leads-to* (↝). The step-up matters because the two kinds route to different checkers — a state search for the first, a temporal model checker for the second.

The four operators the rest of this chapter leans on all describe the shape of a *run* — a path the system walks, one state after another. They are claims about that path, not math:

- **Always (□P)** — P holds at every step of every run: nothing bad ever slips through.
- **Eventually (◇P)** — some step of the run satisfies P: the good thing arrives, sooner or later.
- **Leads-to (P ↝ Q)** — every step where P holds is followed by a later step where Q holds: once P, then Q eventually.
- **Until (P U Q)** — P holds at every step until a step where Q becomes true, and Q does become true.

Inset I6 — Safety vs liveness

Two shapes cover most invariants. **Safety** says *a bad thing never happens* — written $\Box P$, "always P." "Never both leased and free" is safety. **Liveness** says *a good thing eventually happens* — written $P \rightsquigarrow Q$, "P leads to Q." "A preempted job eventually re-runs" is liveness. The shape is not decoration: it **derives the checker**. A safety property is settled by a state-space search that looks for any reachable bad state; a liveness property needs a temporal model checker that reasons about infinite or cyclic behaviors. Declare the shape wrong — route a liveness property to a safety runtime — and the checker structurally cannot see the violation, and reports green. So the shape is checked too: a lint asserts the declared temporal form matches the routed checker.

Inset I3 — What is bounded model checking?

A test *samples* the input space — it runs the cases you thought of and sails past the one adversarial schedule you did not. **Model checking** instead *exhaustively explores* the state space: it visits every reachable state, or every interleaving of a concurrent system, and either proves the invariant across all of them or hands you a concrete **counterexample trace** — the exact sequence of steps that reaches the bad state. **Bounded** model checking does this out to a fixed depth: it explores every state reachable in k steps. Within that bound the proof is total; a bug that needs $k+1$ steps is out of scope, the honest limit you trade for a decidable check. Why it matters for a fleet of concurrent agents: a distributed race has failure traces no hand-picked example hits, and only an exhaustive walk of the interleavings finds them. A green sampled test over that race means nothing; a clean bounded model check means no bad state is reachable within the bound.

The general type behind these ideas — an **automaton with a formal invariant** — gets no separate model page; it is introduced here as trunk machinery and embodied by the first real model below: formal invariant verification, which takes an invariant's temporal shape and uses it to route the exhaustive checker that proves it.

3.3.1 Formal invariant verification

Formal invariant verification is the backbone the other three models hang their invariant tables on. It assesses **invariant soundness under interleaving** — whether a property holds on every reachable state, not just the schedules a test happened to try — and answers it by an exhaustive check that either proves the property or returns a counterexample trace. The distinctive move: the invariant's *temporal form is a routing input, not a label*. A safety form ($\Box P$) routes to a state-space search; a liveness form ($P \rightsquigarrow Q$) routes to a temporal model checker. Get the form wrong — declare a liveness property as safety — and the safety runtime structurally cannot see the violation and reports green. So a lint checks the form against the checker that ran: the shape is consumed, and a mis-routed invariant is a build failure rather than a false all-clear. Its full construct-and-invariant treatment is in the appendix; the temporal operators it routes on are defined above.

Learn more about this governance mechanism: formal invariant verification.

That backbone proves invariants once you can state them. The next three models are about the invariants concurrency hands you: the ones nobody writes down until an inverted lock ordering deadlocks the build. Start with the locks themselves.

3.3.2 The synchronization model

A typed registry that models the system's synchronization behavior — every OS-level lock, which shared resource it guards, and the required acquisition ordering — so concurrency contracts are declared and checkable, not tribal.

(a) Quality property it helps assess. Two, both catastrophic at runtime and invisible in the code.

- **Lock coverage**: *is every OS lock in the codebase declared, and is what it guards known?* An undeclared lock is one nobody can reason about; the coverage lint makes it a build failure.
- **Deadlock-freedom**: *can two locks be acquired in an order that deadlocks?* The declared ordering graph lets a lint answer "which locks, in what order" before the code runs, so an inverted acquisition fails at author time instead of hanging in production.

(b) Constructs and relations. Three record kinds compose the registry.

- `SyncLock` — one OS primitive: its lock-file path, its cap (1 for a mutex, M for a semaphore), the resource it guards, its bypass-env, its audit-log.
- `LockAcquirer` — one declared acquisition site: where in the code a lock is taken, or a declared "takes none" with a rationale.
- `LockOrdering`: a before/after constraint between two locks, with a rationale. The set of these is the ordering graph the deadlock lint walks.

One `SyncLock` is acquired at many `LockAcquirer` sites and constrained by many `LockOrdering` edges.

(c) Visual depiction. The natural diagram, Figure 3.3-1, is an ER schema — three related record kinds with crow's-foot cardinality. Reused from the model's appendix Structure slot:

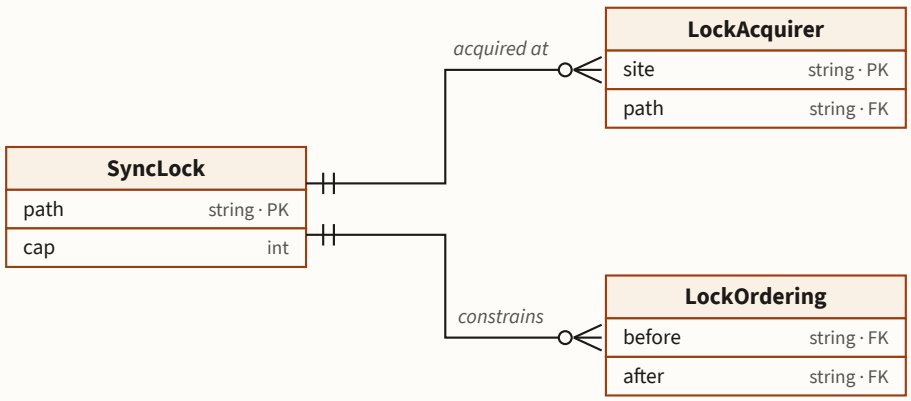


Figure 3.3-1: One lock record, many acquisition sites, many ordering constraints — the ER schema the coverage and ordering lints walk.

(d) Invariants, and how they are checked. A coverage lint and an ordering lint, the second doing genuine graph reasoning, collected in Table 3.3-1:

Table 3.3-1: Invariants of the Process view — each with its temporal shape and the check that holds it.

Invariant	Temporal shape	How it is checked
Every real lock call site is a declared acquirer	$\Box P$ (safety)	Coverage lint scans the lock call sites; each must be declared or carry a "not a sync lock" rationale.
No acquisition inverts a declared ordering	$\Box P$ (safety)	Ordering lint walks each code path's acquisition sequence against the ordering graph; an out-of-order acquire is a finding.
Every exempt site carries a rationale	$\Box P$ (safety)	The closed set of exempt rationales — an unexplained exemption fails the coverage lint.

The ordering lint walks a lock-acquisition sequence against the declared graph and fails on an inversion, so a deadlock-inducing order is caught at author time:

```
import sys

# Declared ordering: a lock in `before` must be acquired before its `after` lock.
ORDERINGS = [("db-lock", "cache-lock")]      # db before cache, always

def ordering_lint(acquire_sequence: list[str]) -> list[str]:
    """A held-lock acquiring one that must precede it is an inversion (deadlock
risk)."""
    findings, held = [], []
    for lock in acquire_sequence:
        for before, after in ORDERINGS:
            if lock == before and after in held:
                findings.append(f"acquired '{before}' while holding '{after}' —
order inverted")
        held.append(lock)
    return findings

if __name__ == "__main__":
    findings = ordering_lint(walk_acquisitions())   # a code path's acquisition
order
    for f in findings:
        print(f"LOCK-ORDER: {f}")
    sys.exit(1 if findings else 0)
```

(e) Traceability and derivation direction. *Model-from-code.* The coverage lint scans the real lock call sites and requires each to be a declared acquirer, so the code is the ground truth and the model is the checked view. The join key is the lock `path` (which lock a site takes) and the acquirer `site` (where the acquisition happens).

Also seen in: Physical (a lock is held per-host, so placement matters). Rendered in full here; its higher-level siblings, the mediator and single-writer registries, sit beside it below.

3.3.3 Two ways to make a concurrent write safe: the mediator and the single-writer registries

The synchronization model catalogs the raw locks. Two higher-level registries sit above it, and together they draw a distinction worth keeping sharp: **there are two ways to make a concurrent write safe, and they fail differently.** You can *cap how many* run at once, or you can insist that *exactly one* may write. A mediator does the first; a single-writer contract does the second. Confusing them is how a torn write hides behind a semaphore that was only ever rationing load.

Learn more about this governance mechanism: synchronization model.

Learn more about this governance mechanism: concurrency contracts.

The mediator registry

The mediator registry is the *how-many* half. It declares which heavy or shared-resource subprocess invocations must run through a serializer — the test runner, the build tool, the whole-repo lint mutex — and at what concurrency cap (1 for a mutex, M for a semaphore). Its quality property is **mediation coverage**: the serializer enforcer can only guard a call that reaches it, so a newly-added raw call that skips the serializer is invisible to the enforcer. The coverage lint closes exactly that blind spot: an unmediated call to a mediated class fails the build at author time rather than surfacing as a host-trampling race in production. The failure it prevents is *resource contention*: too many heavy processes on one box. Its full construct-and-invariant treatment is in the appendix.

The single-writer registry

The single-writer registry is the *exactly-one* half, and its failure mode is what makes it a separate model. A mediator caps how many may run; a single-writer contract says one function, and only one, may write a given piece of shared state. The failure it guards against is a *torn or interleaved write* — two writers corrupting the same state — which a concurrency cap of two would happily permit, because two mediated processes racing on the same record is still within cap. Its quality property is **single-writer coverage**: every mutator of shared state has a declared monopoly, and a second discovered writer of that state is a finding. The mediator bounds contention, the single-writer registry bounds *ownership*, and only naming both keeps a corruption bug from hiding inside a load limit. This one cost me a day of chasing a corrupted registry file before I saw that two “safely rationed” writers were the whole problem: the semaphore was doing its job and guarding nothing that mattered. Its full construct-and-invariant treatment is in the appendix.

—

The Process view holds the system consistent under interleaving. Next, the Development view: how the code is packaged, layered, and owned.

3.4 The Development View

The Process view held the running system consistent. The Development view steps off the runtime entirely and looks at the source: how the code is packaged into modules, which layer may depend on which, and who owns each piece. It is the view the people and agents who *build* the system reason through. Before an agent changes a line it must answer two questions — where am I in this codebase, and what may this file touch? — and this view is the map that answers them.

One general type anchors the view. A **bill of materials** captures a project's dependencies, direct and transitive — the software supply chain a build rests on. Two real models embody the view on live code. The first is the component and zone model, the ownership and boundary map. The second is the rule-metadata registry, the slice of the domain registries that models the governance rules themselves — the same typed rule index this book's own catalogue is generated from, so the book itself is partly governed by a model it describes.

Learn more about this governance mechanism: component and zone model.

Learn more about this governance mechanism: domain registries.

A **bill of materials** — the view's general type — is the dependency graph of a build: every third-party package it pulls in, direct and transitive, the SBOM lens on the code. It is a development-view concern because a dependency is a packaging fact, not a runtime one: it decides what the build fetches and what a fresh checkout must resolve. The discipline it enforces is completeness. Every package the code or the quality gates use must appear in the matching manifest, or a fresh checkout's build breaks on a resolve the developer never saw fail. An out-of-band install that never lands in the manifest is invisible to the bill of materials, and invisible is where a supply-chain surprise lives.

So much for what the build depends on. The next model answers the question an agent asks first: where am I, and what is this file allowed to touch?

3.4.1 The component & zone model

A typed catalog of every component's code zone — its focus directories, its tags, its boundary kind, its external seams, its read surfaces — so "which component owns this file, and what may touch it" is a queried fact rather than a per-tool guess.

(a) Quality property it helps assess. Two, both drifting silently the moment a directory moves.

- **Ownership correctness**: *which component owns this file, and does that answer match the real directory tree?* A reverse-mapping test reconciles the declared zones against the tree in both directions, so a moved directory fails the test instead of quietly staling every tool's private inference.
- **Boundary soundness**: *may this component's zone reach that seam?* The declared boundary kind and sanctioned seams make an out-of-bounds reach a lint finding, not a slow erosion.

(b) Constructs and relations. A typed registry keyed by component.

- `Component` — one unit of ownership: its focus directories, its tags, its boundary kind (internal, a trust edge, an external seam), and its declared read surfaces.
- **The zone relation**: each `Component` claims a set of directories; the union must partition the real tree, and the reverse-mapping test asserts the match both ways.

- **The seam relation**: each boundary component declares the external seams it may cross, so a reach outside the declared set is a finding.

(c) Visual depiction. The natural diagram, Figure 3.4-1, is a component flow — the registry feeding the tools that read it, with a reverse-mapping test joining registry to real tree. Reused from the model's appendix Structure slot:

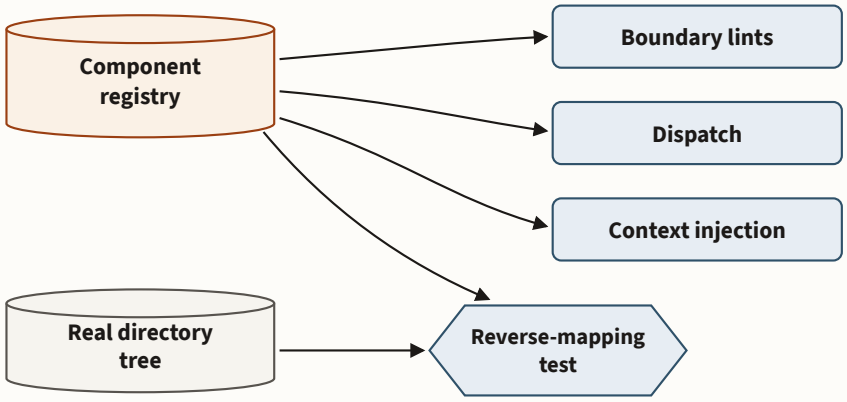


Figure 3.4-1: The component registry feeds the boundary lints, dispatch, and context injection — every tool reads the same ownership answer — and a reverse-mapping test joins it to the real tree in both directions.

(d) Invariants, and how they are checked. A reverse-mapping test and boundary lints, collected in Table 3.4-1:

Table 3.4-1: Invariants of the Development view — each with its temporal shape and the check that holds it.

Invariant	Temporal shape	How it is checked
Every declared zone matches a real directory	□*P* (safety)	Reverse-mapping test, model ⊆ reality — a declared zone with no directory is a finding.
Every source directory is owned by exactly one component	□*P* (safety)	Reverse-mapping test, reality ⊆ model — an unowned or double-owned directory is a finding.
No component reaches a seam its boundary kind forbids	□*P* (safety)	Boundary lint over the declared seam set.

(e) Traceability and derivation direction. *Model-from-code.* The reverse-mapping test re-reads the real directory tree and reconciles the declared zones against it, so the tree is the ground truth. The join key is the focus directory: a reader round-trips from a `Component` record to the files it owns by that path prefix, and dispatch, lints, and context injection all resolve ownership through the same key.

Also seen in: Logical (a component is a functional-structure unit). Rendered in full here.

That model maps the code. What maps the governance itself? The next registry turns the lens on the rules — and, because this book is generated from that same index, on the book you are reading.

3.4.2 The rule-metadata registry

The rule-metadata registry is the self-referential slice of the domain registries: the governance rules' own metadata, and so the model the governance machinery — and this book — are partly governed by. It assesses **rule-index consistency**, whether the published rule index matches the rules declared in the code. The distinctive move: the metadata is *extracted from inline blocks at each rule's own site*, then generated into the index. The description of a rule lives next to the rule it governs, so it cannot rot into a separate document nobody updates. Two gates hold the round-trip — a completeness lint fails a rule that omits a required field, and a freshness lint fails a committed index that differs from a regeneration. The result is a registry that can round-trip from a published rule straight to the code site that declares it, checked on every build. The appendix walks its schema and both gates in full; the component & zone model above is the worked example this view teaches through.

—

The Development view maps how the source is organized for the people and agents who build it. Next, the Physical view: where the built parts actually run, and under what per-host cost and load policy.

3.5 The Physical View

The Development view mapped how the source is organized. The Physical view — Kruchten's deployment view — maps where the built software actually runs: which process lands on which host, across which network boundaries, under what per-host cost and load policy. It is the placement lens, and it earns its keep on a failure the other views cannot see: a system that is functionally correct can still be mis-deployed. Only a view that names *where things run* catches it. Written down as a placement model the fleet can read, that *where* becomes something an agent reasons over before it deploys, not a fact discovered after — the Modeling Thesis reaching the one view the others cannot see.

One general type anchors the view, and it is the honest exception in this zoo. A **performance and cost model** models the computation itself — what it costs to run, what it costs to move data, whether to compute locally or remotely, whether to cache or recompute. The next section takes that gap head-on: the framework supports the cost view, and one real model *embodies* the load-and-cost half of it, but the pure-latency slice has no dedicated instance, because no failure had justified building one until one arrived. Three real models carry the view: the invariant-DAG execution policy (the idea this chapter turns on — per-host cost and load rationing), the deployment and tier topology model (placement parity, worked in full below), and the substrate-dependency model (a computed migration blast radius).

Learn more about this governance mechanism: deployment and tier topology model.

3.5.1 The performance/cost view: mostly embodied, one honest gap

The zoo names a performance and cost model as a general type, and the framework supports it. But the catalogue holds only a partial instance, and the book would rather state that flat than invent a model to fill the slot.

The load-and-cost half of the view *is* embodied — by the invariant-DAG execution policy below. Its Scheduler reads a per-host profile of a concurrency ceiling and a budget, and rations both. The more elastic the host, the wider the Scheduler fans work out; the scarcer the box, the harder the plan serializes — and a cost gate is honored everywhere, a scarce-resource gate only where a box demands it. That is a real cost model in action: cost-to-run and resource-contention, driven from a declared per-host profile.

Learn more about this governance mechanism: invariant-DAG execution policy.

What has no dedicated instance is the *pure-latency* slice: a model of request latency, cache-hit ratios, and data-movement cost. The framework could carry it the way it carries the others — a typed profile, invariants, a drift gate. The zoo names the gap rather than padding it, on the discipline the whole harness leads with: **model only where a failure lives.** A model built ahead of its failure rots before it is ever checked. But that discipline cuts both ways, and the latency slice is where it drew blood.

DocAble has paid for this gap once. To cut idle cost, the fleet let its services scale to zero — a change the cost half of this model welcomed and the latency half could not weigh. Cold-start latency then surfaced in production, and the surface number hid it: a health check answered in about 150

milliseconds while the PDF-conformance service — which shelled out to a Java validator once per request — carried a *warm* floor over four seconds, worse on a cold boot. Chasing it cost days of telemetry and a re-architecture that kept the validator resident, cutting that floor from 4,057 to 109 milliseconds. A latency model here — request latency typed per service, the cold-start cost of scaling to zero an invariant over it — would have flagged that penalty before a user ever waited on it. The gap stays honest; this is what it costs to leave open.

3.5.2 The deployment & tier topology model

The typed statement of where things run — each service's name, its layer, and its tier — from which the deploy scripts and layering lints reason about a declared topology rather than scattered constants, and against which the real deploy table is reconciled.

(a) Quality property it helps assess. Two, both a placement fact that drifts silently the moment a service moves.

- **Deployment parity**: *does every service the model declares actually deploy at the tier the model says, and does every deployed service appear in the model?* A tier that drifts in the code without a matching model edit fails the build instead of surfacing as a production surprise.
- **Layer-boundary soundness**: *may this layer import that one?* The same model carries the layer-boundary graph a cross-layer-import lint holds, so an expedient shortcut across a declared boundary is a finding, not a slow erosion.

(b) Constructs and relations. A frozen record per service, keyed by name.

- `Service` — one deployable unit: its name, its owning layer (web, worker, shared), and its tier class (critical, batch).
- **The tier relation**: each `Service` declares the tier it must deploy at; the set of declared tiers is what the deploy table is checked against.
- **The layer relation**: the layers form the dependency graph a cross-layer-import lint reads, and the same topology is the ground truth the migration blast-radius query joins against.

(c) Visual depiction. The natural diagram, Figure 3.5-1, is a deployment diagram — the build host that produces the image, and the runtime cluster it deploys into. Reused from the model's appendix Structure slot:

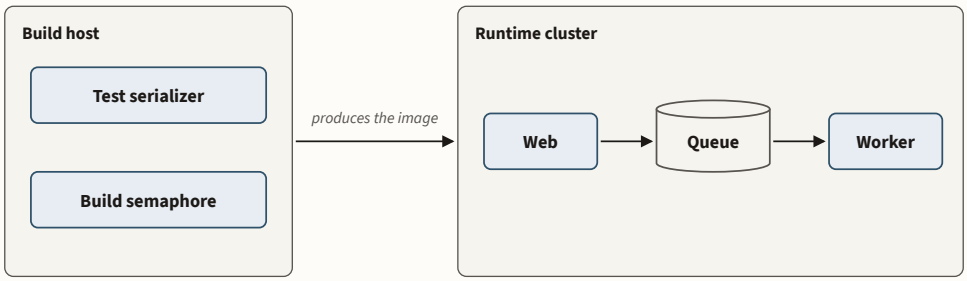


Figure 3.5-1: The model declares placement — the build host that produces the image, the runtime cluster that deploys it.

(d) Invariants, and how they are checked. A bidirectional set-diff and a boundary lint, collected in Table 3.5-1:

Table 3.5-1: Invariants of the Physical view — every service deploys at the tier the model declares, held to the real deploy tables.

Invariant	Temporal shape	How it is checked
Every declared service deploys at the tier the model declares	□*P* (safety)	Parity lint, model ⊆ reality: a declared service missing from the deploy table, or at the wrong tier, is a finding.
Every deployed service appears in the model	□*P* (safety)	Parity lint, reality ⊆ model: a service in the deploy table the model never declared is a finding.
No layer imports across a boundary the model forbids	□*P* (safety)	Cross-layer-import lint over the declared layer graph.

The parity check is a set-diff run both directions against the live service-to-tier map. Read the model's declared tiers, read the real deploy table, and print each mismatch: a declared service whose tier differs from the deploy table's, and every deployed name absent from the model. Neither direction is optional — model ⊆ reality alone would miss a service that shipped without a model row, and reality ⊆ model alone would miss a tier that drifted in the code.

(e) Traceability and derivation direction. *Model-from-code.* The parity check re-reads the real deploy table and reconciles the declared topology against it, so the running system is the ground truth for what deploys. The join key is the service name, which indexes both the `Service` record and its row in the deploy table. The model is authoritative for the layer graph it holds — the import lint reads it directly — and reconciled against reality for the tier placement it checks.

Also seen in: Development (a layer boundary is a packaging fact). Rendered in full here.

That model says where the parts run. The next one says where the *checks* run — the same physical-placement thinking, applied to the gates themselves rather than the services.

3.5.3 The invariant-DAG execution policy

A deploy graph whose edges carry a typed intent — correctness, cost-gate, or load — kept host-identical, plus a typed Scheduler that reads a per-host profile and rations load and cost so one host fans work out while a scarce one serializes it.

(a) Quality property it helps assess. Two, both about how a host runs the work it was handed.

- **Graph portability**: *is the deploy graph the same on every host?* Load-specific edges are banned from the graph and migrated to the Scheduler, so the graph itself carries only host- identical correctness and cost intents, and cannot fork per host.
- **Rationing correctness under cost and resource pressure**: *does the deploy fan work out when it safely can, and serialize only when it must?* The Scheduler honors a cost gate everywhere but rations concurrency only where a scarce box demands it, all from one profile table.

(b) Constructs and relations. An edge-intent axis plus a per-host Scheduler.

- **The edge intent** — each deploy-graph edge carries a typed intent. `CORRECTNESS` means B is wrong without A, honored on every host. `COST_GATE` means A is a cheap check gating an expensive B, honored by default and relaxable only under an unbounded budget. `LOAD` means B contends with A for a scarce box; it is *banned in the graph*, migrated to the Scheduler.
- `HostLoadProfile` — a per-host record: a concurrency ceiling (1 for a single scarce box, large for an elastic one) and a budget (finite to honor cost gates, unbounded to relax them).
- **The Scheduler** — reads a host's profile and emits an execution plan: how many roster items may run at once, and whether to honor the cost gate. Load rationing is a semaphore, not a graph edge.

(c) Visual depiction. The natural diagram, Figure 3.5-2, is a data-flow — edge intents route, load leaves the graph for the Scheduler, and a per-host profile drives the plan. Reused from the model's appendix Structure slot:

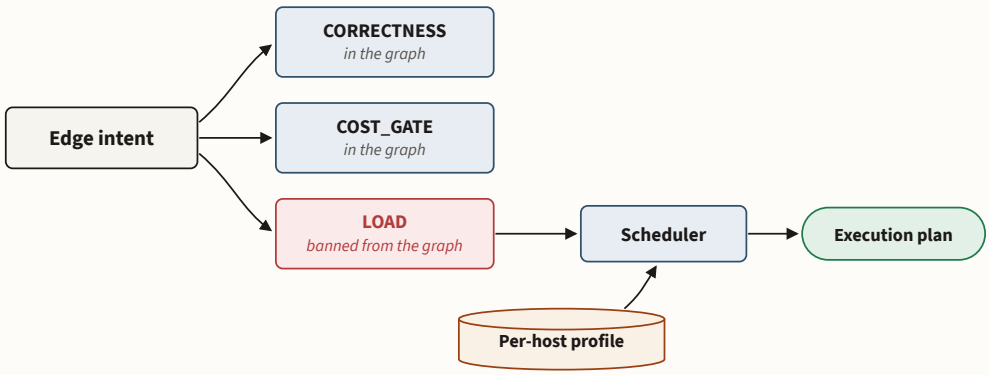


Figure 3.5-2: Profile Edit, Not Graph Edit. *Correctness and cost-gate edges stay in the deploy graph; a load edge is banned and migrates to the Scheduler, which reads the per-host profile and emits the execution plan. Moving stress between hosts is a profile edit, never a graph edit.*

(d) Invariants, and how they are checked. A load-edge ban and a plan derivation, collected in Table 3.5-2:

Table 3.5-2: Invariants of the host-stress model — load edges migrate to the Scheduler, never the deploy graph.

Invariant	Temporal shape	How it is checked
No deploy edge carries the `LOAD` intent	$\Box P$ (safety)	Load-edge lint reads the graph and the Scheduler's graph-resident intent set; a `LOAD` edge is a finding.
The deploy graph is identical on every host	$\Box P$ (safety)	Graph-parity check: the same graph is emitted for every host; a per-host fork is a finding.
The execution plan is a pure function of the host profile	$\Box P$ (safety)	Derive-and-assert: the plan is recomputed from the profile, never hand-stored.

(e) Traceability and derivation direction. *Model-from-code.* The plan is derived from the per-host profile by a pure function, and a hand-stored plan or a `LOAD` edge is banned outright. The join key is the host name, which indexes both the `HostLoadProfile` and the deploy phase that runs under the

emitted plan. This model is walked in full, with its Scheduler code, in the Scenarios chapter's join example — here it is the Physical-view resident that the join reaches into.

Also seen in: Process (concurrency rationing is a runtime-dynamics concern) and Scenarios (the join example). Rendered in full in the Scenarios chapter; referenced here for the placement and cost half.

3.5.4 The control↔substrate dependency model

The substrate-dependency model makes each governance mechanism declare, as typed metadata, the substrate assumption it bakes in — bound to one substrate, substrate-aware, or substrate-agnostic. It assesses **migration safety**: before a cross-cutting substrate change, exactly which mechanisms assume the old substrate? That declaration turns a grep into a *computed blast radius*. Joining the declared stances against the topology model on the migration target prints the in-scope mechanisms as a table — a real answer, not a hopeful search that misses the one whose assumption is implicit. A declaration lint requires the stance, so a mechanism that reads the substrate silently cannot hide from the join. The appendix carries its typed schema and the join code in full.

Learn more about this governance mechanism: control↔substrate dependency model.

—

The Physical view places the parts and rations their cost. That completes the four static views. Next, the +1: the scenarios that set them in motion and validate that the other four hold up.

3.6 The Scenarios View

The four views so far are static portraits. The Logical view says what the system is, the Process view what it does at once, the Development view how it is packaged, the Physical view where it runs. The +1 view sets them in motion. A **scenario** — a use case, a journey — walks a real goal end to end, and in walking it exercises the other four and validates that they hold up. Kruchten made scenarios the +1 for exactly this reason: a view is validated by walking it, and the scenario is the walk. Because the walk re-checks the other four against a real goal, the scenario is where the models are held honest — the Alignment Thesis turned back on the views themselves.

One general type anchors the view. A **user journey** names the interfaces a user moves through and the actions they take to accomplish something of value. Four real models embody the view: the user-journey model (the product-goal-to-code bridge), the agent-orchestration model (the *developer*-journey counterpart, the scenarios view pointed at the fleet rather than the user), journey-criticality-to-test-placement, and coverage-to-model-node mapping. A fifth, journey task-closure, sits alongside them: it types what the journey's terminal assertion must *mean*, a concern the join's four do not touch. The chapter closes with the zoo's flagship demonstration — four models joined so a product goal reaches all the way to how the deploy rations its tests.

Two background ideas come first — how node coverage differs from line coverage, and why a protocol needs a temporal-logic model checker.

> ***Inset I10 — Coverage over model nodes vs line coverage***
> **Line coverage** counts *lines* — which source lines a test suite executed. It is easy to game and easy to misread: a high percentage can hide the one untested branch that matters. Node coverage counts *meanings* instead. Project the coverage onto a model's nodes — its states, its seams, its invariants, its journey endpoints — and ask which *nodes* a test exercised. Now an untested invariant cannot hide inside a 95% line number, because the invariant is a node, and the node is either covered or it is a visible gap. The shift is from "how much code ran" to "which meanings are checked," and only the second answers "is the thing I care about tested?"

> ***Inset I7 — Protocols and TLA+***
> When a property is not about one component but a *protocol* — many actors interleaving over time, each taking steps in an order you do not control — a single state assertion is not enough and even a bounded walk of one component misses the cross-actor races. You specify the protocol in a **temporal-logic language** (TLA+ is the best-known), stating the actors, their steps, and the safety and liveness properties the interleaving must satisfy. A model checker then explores *every* interleaving the actors can produce and either proves the properties or hands you a trace. This is the heaviest tool in the zoo, reserved for the invariant whose failure lives in a schedule no example test will ever pick — a distributed reservation, a lease-and-preempt loop, a two-phase hand-off between services.

The four models that feed the flagship join each set up their part of the composition, with the full construct-and-invariant treatment of each in the appendix.

3.6.1 The user-journey model

The user-journey model makes the product's journeys first-class typed entities — each an actor pursuing a goal through ordered steps, every boundary-crossing step joined to the endpoint it calls. It assesses **dependency correctness**: does every step a journey declares have a real call site, and is every real call declared? A call-site drift lint checks both directions. The idea that makes it the join's entry point is that the journey *copies nothing* — a step's endpoint references the service-flow model, its call-site anchor references the real code, and those two references are the keys the coverage and placement models reach through. Name the journey once and the rest of the zoo joins to it.

Learn more about this governance mechanism: user-journey model.

3.6.2 The agent-orchestration model

The agent-orchestration model is the developer-journey counterpart — the scenarios view pointed at the fleet rather than the user. It models the agent lifecycle (`Dispatched→Working→Landed→Tombstoned`, with `Abandoned` and recovery) as a typed state machine, and the orchestrator's own refill-and-bank loop as a journey. It assesses **lifecycle soundness**: a tombstone before a commit, or a landed agent that never tombstones, is an illegal transition a checker catches. The distinctive move is **method parity** — the substrate that produces the software is governed by the same tier-derivation and drift machinery pointed at the product, so "is the fleet as checkable as what it ships" is a checked property. One of its invariants is liveness (*a landed agent eventually tombstones*), which routes to a temporal checker per the Process view's form-match rule. Its full construct-and-invariant treatment is in the appendix.

Learn more about this governance mechanism: agent-orchestration model.

3.6.3 Journey-criticality → test-placement

This model — the **Selector** in the join below — derives which environment tier a journey's tests run in from the journey's criticality: a `MAJOR` part runs the fast local tier and the full staging matrix, a `MINOR` part runs staging only. It assesses **placement soundness**. The tier is *derived, never stored*: a stored tier literal is banned, so you cannot quietly push a major path off the slow local gate to speed a run up. To move it you must demote it to `MINOR` — a visible edit that says out loud "this path is no longer major." A coverage-floor lint then holds the promise "every major part has a fast-tier test," which makes a green local run mean something. Its full construct-and-invariant treatment is in the appendix.

Learn more about this governance mechanism: journey-criticality test-placement.

3.6.4 Coverage → model-node mapping

Coverage-to-node mapping projects the test suite's coverage onto the model's *nodes* — states, seams, invariants, journey endpoints — instead of its lines. It assesses **node-coverage adequacy**: is every meaning I care about exercised by some test? An untested invariant cannot hide inside a 95% line number here, because the invariant is a node, and a node is either covered or a visible gap. An uncovered node is a backlog item; an uncovered node in the critical subset is a build-blocking gate. In

the join below it answers the plain question the criticality floor rests on — are the journey's endpoints tested at all? Its full construct-and-invariant treatment is in the appendix.

Learn more about this governance mechanism: coverage-to-node mapping.

3.6.5 Journey task-closure — asserting the goal completed

The three models above ask whether a journey's test *exists*, *runs*, and runs *in which tier*. None asks what the test asserts at the end. Split the pair the model turns on: a *flow* assertion says the journey ran — a page returned 200, a card appeared — while a *task* assertion says the user got the thing they came for. A test built on flow alone can green while the task is broken, the artifact never painted, the file never downloaded. The gap is one hop past the last flow assertion, and it is where a real production break hid — a navigation to the editor route succeeded while the accessibility view it should have shown failed to load. Journey task-closure closes that hop. It promotes the journey's terminal "what *works* means" out of a review-checklist paragraph into a typed `closure` post-condition: a small boolean expression over reusable observable predicates, joined by AND, OR, and NOT, *"the artifact rendered AND a control is inspectable AND no error-fallback is shown."* The leaves come from a shared library a journey subclasses rather than hand-rolling, and each prefers an accessibility observable — a role, an accessible name — because that is how the user perceives that the task is done.

From the expression a pure function derives a **closure-strength** (`TASK_CLOSED`, `FLOW_ONLY`, or `DISABLED`), stored nowhere by hand. Two leaves are flow-only: *navigated to a URL*, *a route returned 2xx*. An expression may reference them, but a closure built only from them derives `FLOW_ONLY`, and on a major journey that is a build finding — the mechanical form of "a journey test should have caught it." That asymmetry is the model's teeth. This model assesses **closure meaning**, the axis its three siblings leave to human review: presence and tier versus the meaning of the terminal assertion. And because the closure is typed rather than prose, the same definition resolves two ways: a browser assertion in the fast local tier, and a headless probe against the pre-promotion canary. That restores "staging catches anything local catches" for the task dimension, a containment the fast post-deploy battery had been blind to because it checks page integrity, never task closure. Its full construct-and-invariant treatment is in the appendix.

Learn more about this governance mechanism: journey task-closure.

—

3.6.6 Worked join — the journey↔coverage↔deploy-policy composition

Four models composed on a shared key, so a product goal reaches all the way to how the deploy runs its tests. A journey's criticality derives which tests run on which host tier, and that placement joins to the deploy execution policy, which rations each host by fanning tests out or serializing them.

Watch one fact — a journey is *major* — travel through four models with nothing said twice. The other model pages each show a single model; this one shows them **joining, not repeating.** The user-journey model (M13) names the journey and its endpoints. The coverage-to-node map (M5) asks whether those endpoints are tested at all. The journey-criticality-to-test-placement model (M6, the

Selector) derives which host tier each test runs on from the journey's criticality. And the invariant-DAG execution policy (M7, the **Scheduler**) decides *how* each host runs the tests it was handed — fan out, or serialize. No model owns another's facts; each joins on the journey and its endpoints.

The three questions the join answers

Three questions the raw deploy scripts and test config cannot answer on their own.

- **Goal-to-gate coverage**: *does a green local run mean every major journey actually ran?* The criticality floor makes "local-green implies every major path was exercised locally" a checked property, not a hope — and journey task-closure sharpens what *ran* means, proving the terminal assertion closes the user's task rather than merely reaching the last flow step.
- **Placement soundness**: *is any major journey mis-filed as staging-only?* Because the host tier is derived from criticality and stored nowhere by hand, you cannot quietly shove a major path off the local gate; you must demote it to minor, a visible edit.
- **Rationing correctness under cost and resource pressure**: *does the deploy fan tests out when it safely can, and serialize them only when it must?* The Scheduler honors a cost gate everywhere but rations concurrency only where a scarce box demands it, so a single-worker host serializes while an elastic host fans out, from one profile table.

The four models and their join keys

The four models compose through shared keys.

- **The** `Journey` **(M13)**: an actor, a goal, ordered steps; each boundary-crossing step names its **endpoint** and records a **call-site anchor**. These are the join keys the other three reach through.
- **The criticality axis (M6)**: each journey-part carries `MAJOR` or `MINOR`. A total derivation maps `MAJOR` to local-depth and `MINOR` to staging-only. The tier is derived, never stored.
- **The coverage join (M5)**: the suite's coverage projected onto the journey's endpoint nodes: covered, uncovered, by-which-tests. An uncovered major endpoint is a floor violation.
- **The edge-intent axis and Scheduler (M7)**: the deploy graph's edges carry `CORRECTNESS`, `COST_GATE`, or `LOAD` `LOAD` is banned from the graph and migrated to the Scheduler, which reads a per-host `HostLoadProfile` and emits a `ConcurrencyPlan`.

The through-line: a journey's endpoint is the coverage join key (M13→M5); its criticality is the Selector's input (M13/M6); the Selector's per-host test set is what the deploy phase runs; and the Scheduler's profile decides whether that host fans those tests out or serializes them (M6→M7).

The composite picture

The join has no single appendix diagram, so this composite is composed from the four real panels, adding only the join edges between them. The data-flow reads left to right — criticality derives placement, placement joins to coverage, and the deploy Scheduler rations the resulting test set per host, in Figure 3.6-1:

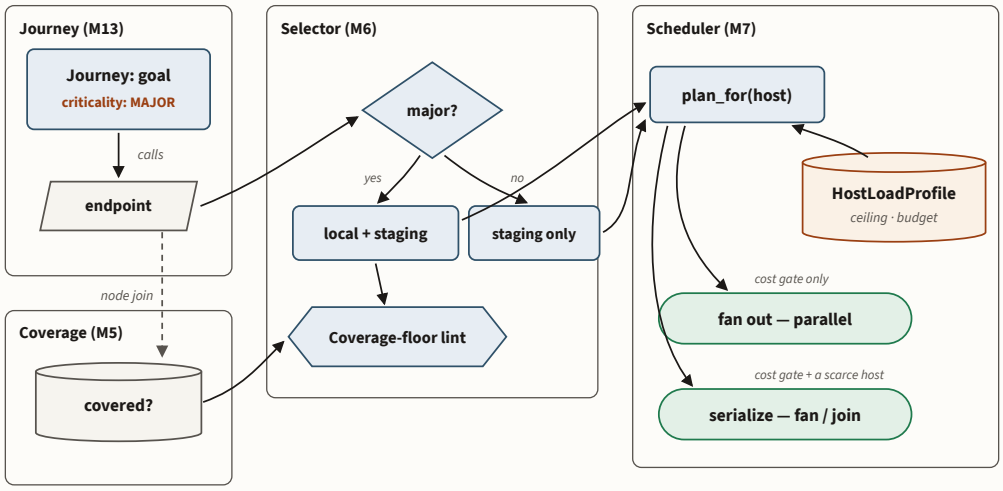


Figure 3.6-1: The Four-Model Join. *A journey's criticality derives its test tier, coverage joins on the same endpoint node, and the Scheduler rations the resulting set per host — parallel when only a cost gate applies, serialized when a scarce host is in play.*

The invariants the join spans

The join's invariants span all four models, collected in Table 3.6-1. All are safety properties; the liveness that concurrency hands the fleet is stated and routed in the Process view, whose form-match rule sends a liveness invariant to a temporal checker.

Table 3.6-1: Invariants of the Scenarios view — the properties the journey-to-test join spans and how each is checked.

Invariant	How it is checked
Every major journey-part has a test in the fast local tier	Coverage-floor lint (M6) walks the model; a major part with no local test is a finding.
A journey's declared deps match its real call sites, both ways	Call-site drift lint (M13): every declared dep has a real call site, every call site is declared.
Every declared journey endpoint is exercised by some test	Undertested-journey audit (M5): join coverage to the endpoint nodes; an uncovered endpoint is a gap.
The host tier is a pure function of criticality	Derive-and-assert (M6): a stored tier literal is banned; the lint recomputes the derivation.
No deploy edge carries the `LOAD` intent	Load-edge lint (M7) reads the graph and the Scheduler's graph-resident intents; a `LOAD` edge is a finding.

Invariant	How it is checked
Staging's test set is a superset of local's and of prod's	Containment property test plus a live-model lint (M6): recompute containment by calling the selector, not auditing a matrix.

The Selector and Scheduler in code

Two pieces of real policy code make the join concrete. The **Selector** derives a host's test roster from the journey model; the **Scheduler** decides how that host runs it:

```
from dataclasses import dataclass
from enum import Enum

# --- M6: the Selector — criticality derives the host tier, tier derives the test
roster ---
class Criticality(Enum):
    MAJOR = "major"
    MINOR = "minor"

def tier_for(crit: Criticality) -> frozenset[str]:
    """Pure derivation: a MAJOR part runs local + staging; a MINOR part runs
staging only.
       Stored nowhere by hand — a tier literal is banned, so moving a part off the
local
       floor forces a visible MAJOR->MINOR demotion, not a silent tier edit."""
    if crit is Criticality.MAJOR:
        return frozenset({"local", "staging"})
    return frozenset({"staging"})

def roster_for(host: str, journey_parts: dict[str, Criticality]) -> list[str]:
    """The test set this host runs: every part whose derived tier contains this
host."""
    return [name for name, crit in journey_parts.items() if host in
tier_for(crit)]

# --- M7: the Scheduler — a per-host profile decides fan-out vs serialize for that
roster ---
@dataclass(frozen=True)
class HostLoadProfile:
    concurrency_ceiling: int   # 1 = single scarce box; large = elastic
    budget: float              # inf = unbounded (cost gates relaxable); finite =
honor $ gates

@dataclass(frozen=True)
class ConcurrencyPlan:
    permits: int               # how many roster items may run at once
    honor_cost_gate: bool      # run an expensive test only if its cheap gate
passed

def plan_for(profile: HostLoadProfile) -> ConcurrencyPlan:
    """The load + cost rationing decision, per host, from the profile alone.
       Fan OUT freely when only a $ (COST_GATE) gate applies (elastic box, permits
high);
       fan/join — SERIALIZE — when a $ gate AND a scarce-resource (LOAD) gate both
```

```
apply
       (single box: permits collapse to 1, the serialize case a scarce box
forces)."""
    permits = profile.concurrency_ceiling          # LOAD rationing: a semaphore,
not an edge
    honor_cost_gate = profile.budget != float("inf")  # COST_GATE: honored unless
budget unbounded
    return ConcurrencyPlan(permits=permits, honor_cost_gate=honor_cost_gate)

# Three named host profiles — moving the stress burst between hosts is a one-row
edit:
PROFILES = {
    "local":   HostLoadProfile(concurrency_ceiling=1,    budget=100.0),   # scarce
VM: serialize
    "staging": HostLoadProfile(concurrency_ceiling=999,  budget=float("inf")),  #
elastic: fan out
    "prod":    HostLoadProfile(concurrency_ceiling=4,    budget=50.0),    # low
finite smoke ceiling
}
```

The behavior the code makes concrete: **staging** (ceiling 999, unbounded budget) permits the whole ready wave and relaxes cost gates — pure fan-out, no box scarce. **Local** (ceiling 1, budget 100) collapses to one permit and honors cost gates — the serialize case, where a cost gate and a scarce-box load gate both apply, so the locally-placed tests run one at a time behind their cheap gates. Raising the local ceiling is a one-cell profile edit, never a graph edit — which is why `LOAD` stays out of the DAG.

Tracing one fact through four models

Model-from-code, joined. Every leg derives from the code: the journey's deps from its real call sites (M13), coverage from the real test run (M5), the host tier from the criticality field by a pure function (M6), the rationing plan from the per-host profile (M7). Nothing is hand-stored — a stored tier or a `LOAD` edge is banned outright. The join keys chain: the **endpoint** ties M13 to M5, the **criticality** ties M13 to M6, the **derived tier** ties M6 to M7′s roster, and the **host profile** ties the roster to the plan. A reader round-trips from "this journey is major" to "these tests serialize on the local box" by following those four keys, and the drift lints mechanize each hop.

Also seen in: Physical (the Scheduler half). This join is why the Physical and Scenarios chapters cross-reference each other most; it is rendered in full here.

—

Five views, walked on real code, each with its invariants and the checker that holds them true. The zoo has twenty-nine named models, and the number makes Kruchten's point for him: no single view is ever the whole picture. What the safety-critical world did because lives were at stake — keep the models honest, trace every requirement to the code that realizes it — the rest of us now do too. We do it because an agent will pay the upkeep, and because there is no cheaper way to trust what a fleet ships.

3.7 The Scope of Modeling

The Scope of Modeling: where to read next

This book is an introduction. It gives you enough kinds of model to start, and one discipline — keep the map equal to the territory, so the model stays the interface the fleet reasons through — that makes them pay. It does not teach any single formalism in depth, and the zoo is not the whole field. Modeling software systems is a deep discipline with decades of literature behind it. This chapter is the doorway out: it teaches you to recognize the moment a sketch stops being enough, and points to where each of our models graduates when that moment comes.

3.7.1 When you graduate to a heavier tool

There is a moment when a working sketch stops carrying its weight, and it is worth learning to see it coming. **You graduate from a sketch to a heavier analytic modeling tool when your quality goal becomes precise enough — or rises in priority enough — that you need real analytic power to meet it.** The sketch is fine while the goal is a soft preference; the analytic tool earns its cost the moment the goal becomes a hard constraint you can measurably fail.

The signal is the goal itself hardening. “Fast is good” is a preference, and a napkin sketch of the call graph carries it fine. Sharpen it to “every frame must finish inside 16 milliseconds” and you have a real-time system — you now need schedulability analysis, not a vibe about which parts look slow. “Less memory is nice” is a preference too. Sharpen it to “this must fit on a board with 64 kilobytes of RAM” and you need an honest memory model, budgeted byte by byte. The concern did not change; the preference became a constraint, and the day a constraint can fail is the day the sketch has to graduate. Learn to feel that hardening, and the rest of this chapter is a map of where to go once you feel it.

3.7.2 Where our models stop

Each view here is a working sketch. When one of your goals hardens past what its sketch can say — the moment above, arriving for a particular concern — an established tool is already waiting, with a literature behind it. Here is where each view sends you when you hit that moment.

- **Logical view — structure.** Our types and their relations are a thin notation. For the full one — class, sequence, state, and component diagrams as a standard — graduate to *UML* for systems that span software, hardware, and requirements together, to *SysML*.
- **Process view — concurrency.** We teach automata, safety versus liveness, and bounded model checking as ideas. For machine-checked proofs of them, exhaustive rather than argued, graduate to a real model checker: *TLA+* or *SPIN*.
- **Physical view — performance.** We model load and cost, and we name our own gap: no request-latency model, no economics of contention. For those, an established field is already there — *queuing theory* for throughput under load, *Petri nets* for concurrency-and-resource analysis. Thermal, power, and cooling are past even those, and past this book entirely.

- **Continuous and real-time systems.** The zoo is discrete: states and transitions, not differential equations. A system whose behavior is continuous or hard-real-time wants *timed and hybrid automata* — the pointer the opening of this Part already gave, in Alur and Dill. Named again because it is the most common place a reader's real system steps outside the discrete world this book stays in.

3.7.3 A short shelf

If you read a few things past this book, read these. The first is the source of this Part's spine.

- Philippe Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, 1995[40] — the five views this Part is built on.
- Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*[41] — the standard notation for structure and behavior.
- Sanford Friedenthal, Alan Moore, and Rick Steiner, *A Practical Guide to SysML*[42] — MBSE's working vocabulary of blocks, constraint blocks, and traceability.
- Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*[43] — architecture in the large.
- Christel Baier and Joost-Pieter Katoen, *Principles of Model Checking*[44] — the textbook behind the Process view's insets.
- Leslie Lamport, *Specifying Systems*[45] — TLA+, for specifying and checking concurrent designs.

3.7.4 Digital twins — the branch this book does not enter

One branch of modeling this book leaves untouched is the *digital twin*: a live model of a physical thing, kept in step with its real counterpart by a stream of sensor data, and used to predict and control it. Digital twins are the modeling practice of embedded and cyber-physical systems — aircraft engines, factory lines, power grids, anything where the model shadows a running machine. Ours shadow a codebase; a twin shadows a turbine. The spine is the same, a model kept equal to its territory and reasoned over in the thing's place, but the territory is physical and the toolkit differs. The concept is commonly credited to Michael Grieves; for the engineering discipline under it, Lee and Seshia's *Introduction to Embedded Systems: A Cyber-Physical Systems Approach* is the standard on-ramp.

3.7.5 The through-line

One thread ties every entry here to the book behind it. Each named tool is a way to keep a model honest against some territory — code, a schedule, a queue, a turbine. That is the formal definition of The Agent Stack doing its work one last time: whatever the territory, the thing that earns the word *model* is a deliberately reduced representation that supports an operation — reasoning, derivation,

---

[40] Philippe Kruchten, "The 4+1 View Model of Architecture," *IEEE Software* 12, no. 6 (1995): 42–50.

[41] Grady Booch et al., *The Unified Modeling Language User Guide*, 2nd ed. (Addison-Wesley, 2005).

[42] Sanford Friedenthal et al., *A Practical Guide to Sysml: The Systems Modeling Language*, 3rd ed. (Morgan Kaufmann, 2014).

[43] Len Bass et al., *Software Architecture in Practice*, 3rd ed. (Addison-Wesley, 2012).

[44] Christel Baier and Joost-Pieter Katoen, *Principles of Model Checking* (MIT Press, 2008).

[45] Leslie Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers* (Addison-Wesley, 2002).

checking — and is held to what it represents. What changes per field is only which operation, and what keeps the map true. This book taught one instance of that move, on software, with an agent paying the upkeep that once buried the practice. The rest of the field is the same move on other territories, and the day your quality goal hardens past the sketch — the day you feel that graduation — it is worth your time.

3.8 Keeping the Models in Sync with the Code — A Measurement

3.8.1 The question: the map must equal the territory, or the bridge lies

A context-bounded fleet governs a context-exceeding codebase *through* structured models. The models are the bridge the agents reason across. So the models must equal the territory — if a model claims something the code no longer does, every agent that trusts the model inherits the lie.

"Keep the model in sync with the code, automatically" is the thesis question. It has a cheap-sounding answer and an expensive-sounding one, and the honest result is that neither alone is enough.

3.8.2 The mechanism: a two-layer net, and liveness as a property

Sync holds through two layers stacked over the model↔code gap.

- **The soft layer — the reading pass.** A definition-of-done review reads the model's prose surface against the code: a frozen status header, a phase index missing a landed phase, a stale count in a sentence. A human or an Opus catches these because they live in prose. This layer aims; it cannot block, and it is fallible.
- **The hard layer — the derived floor.** A reconciliation lint resolves each model anchor against a live symbol in the code. The anchor points at a function or class definition a static resolver can find, not at a line number and not at a snapshotted copy. Reading the anchor *is* reading current truth. There is nothing to re-sync, so nothing can lag between runs.

The load-bearing property is **liveness as a property, not a process.** A snapshot-plus-sync-process can drift between runs; a resolver over live code cannot. This is the whole finding compressed: **derived defends, snapshotted drifts.** Anchor to a symbol, not a line — and where there is no clean symbol to anchor to, that absence is itself a signal that the model is missing a point of abstraction (a refactoring target, not a gap to paper over).

3.8.3 The model grows with the system: two worked examples

Sync is a living practice, not a one-time alignment. The territory grows; the model is edited to match; a derived gate fails the build if it is not. Two real cases from the study period show the loop.

Adding a service endpoint (2026-08-01). The "pictures-not-pages" PDF pilot added a new editor endpoint, `GET /api/pdf-page-image/{job_id}/{page}` — the PDF analogue of the existing slide-image and docx-page-image endpoints. The service-flow model (the Backstage-dialect Component/API catalogue under `system-models/services/`) had to gain that endpoint or the generated catalogue would no longer match the source. The commit that landed it (`6fd0b6d51a`) says so in its own words: the endpoint "was added to the editor cluster YAML but the generated Backstage entity was not regenerated," which "Restores `gen-web-api-entities --check` to green." The added model fragment is real:

```
/api/pdf-page-image/{job_id}/{page}:
  get:
    summary: Return the rendered page image for a completed PDF job
             (pictures-not-pages pilot ... the PDF analogue of
             /api/slide-image / /api/docx-page-image)
    responses:
      "200": { description: Rendered PDF page image (PNG) for the SPA
PdfViewAdapter }
      "400": { description: Job not completed, or not a PDF job }
      "404": { description: Job / output file / page not found }
      "502": { description: Render service failed (systemic) }
```

A new service or endpoint is, by construction, either a model edit or a parity failure. The service-flow parity gate (`gen-web-api-entities --check`, backed by the service-flow-model and service-call-graph drift lints) would have failed the build had the model not been edited to match.

Adding user journeys (2026-07-16; the model kept growing through the drift window). The user-journey model gained three Journey entities — `batch-remediate`, `editor-edit`, `editor-edit-remediate` — authored in commit `01699c8c47`, which in the same change extended the service-flow lint to admit the `Journey` kind. A Journey names an actor, a goal, and ordered steps, and each step names the service endpoints it calls. The `batch-remediate` Journey, for instance:

```
kind: Journey
spec:
  actor: uploader
  goal: Get an uploaded document remediated end-to-end (chunk, remediate, merge,
validate).
  steps:
    - seq: 1   # chunk-decision
      calls: [genai-service-capacity, lo-service-convert]
    - seq: 2   # per-chunk C# remediation
      calls: [genai-service-batch, genai-service-complete, genai-service-
transcribe,
              genai-service-detect-bboxes, ocr-service-ocr, font-svc-fonts, font-
svc-cmap]
    - seq: 3   # post-remediation validation
      calls: [render-service-render, accessibility-conformance-service-check,
              accessibility-quality-service-validate, ...]
```

The two-way call-site drift lint (`lint-journey-endpoint-coverage.py` / `lint-journey-service-drift.py`) holds each journey's declared endpoints to the real `internal_service_client(...)` call sites, in both directions — a journey that names an endpoint no code calls, or code that calls an endpoint no journey declares, fails. The model kept growing in the drift window too: the same journey model gained a typed MAJOR/MINOR criticality tier and a `JourneyClosure` join-key substrate (late July), each with its own derived check.

Honesty note: the three Journey entities were authored 2026-07-16 — inside the broader study period, before the 07-22 start of the drift-audit corpus. No brand-new Journey entity landed in the 07-28→08-04 refresh window; the journey model's growth there was the criticality-tier and closure-key expansion. The service-endpoint example (08-01) is squarely in-window.

3.8.4 The evidence

The claim is a field-report result, so the numbers come with their provenance. The model↔code sync evidence under load is collected in Table 3.8-1; a documentation-hygiene aside, kept strictly separate so it is never miscounted as sync evidence, is in Table 3.8-2.

The derived floor under load

Documentation drift is excluded here by construction; this table is model↔code sync only.

Table 3.8-1: Model-sync evidence — the derived floor under load, model↔code sync only (documentation drift excluded by construction).

Signal	Value	Caught by
Model-bridge churn, window 07-28→08-04	**+8,970 / −173 lines across 63 commits** (query/reactor + governance-graph + frontend-build models)	— (this is the load the net held under)
Model↔code drift reaching a post-close reopen	**0** across cumulative N=56 closes	—
Genuine model-drift catches, refresh window	traceability-broken ×3 (unregistered model consumer / missing component entry / service-call-graph), stale-anchor / stale-test ×3	the **derived floor re-run at HEAD** (symbol-anchored drift lint · consumer-registry-fresh · service-call-graph drift), not human reading
Pre-floor retro-audit (before the derived floor existed)	**~27 genuine model↔code drifts at S:N ≈ 1.0**, zero false positives, in closed Epics with green DoDs — including a **prod-blocking pointer-drift incident** and a fully-typed function shipped to zero consumers	the audit re-ran each Epic's own lints; the drift had escaped the green DoDs

The pre-floor row is the "hope-for-the-best fails" evidence: real drift, used to escape, silently, past green definitions-of-done. The floor was built to close exactly that class, and the top rows are it holding. For the data, see Keeping the Models in Sync with the Code — A Measurement → (preliminary)

Documentation-hygiene aside (NOT model sync)

Stale headers and stale prose numbers are **documentation drift, not model drift.** They are the soft layer's true positives, and they are cheap: a reader catches them, and the close tool heals them. Kept here, clearly labelled, so they are never counted as model-sync evidence.

Table 3.8-2: Documentation-hygiene aside — stale headers and prose numbers the soft layer catches; documentation drift, not model sync.

Doc-hygiene drift	Refresh-window count	Caught by	Healed by
STALE-HEADER (a status line frozen at a pre-close phase)	**11**	reading (human / Opus)	the Epic-close tool auto-rewrites the status atomically
DOC-CLAIM (a stale prose number: 105→107, 76%→90%, 266→234)	**9**	Opus re-derives the number from code	a routed one-line `[FIX]` / `[AUDIT]`

Documentation hygiene, not model sync. These rows describe prose the lints do not parse. They are real, and the soft layer is the only practical control for them — but they say nothing about whether the model equals the code.

The whole shape — the gap, the two layers, and the two places drift still escapes — is drawn in Figure 3.8-1.

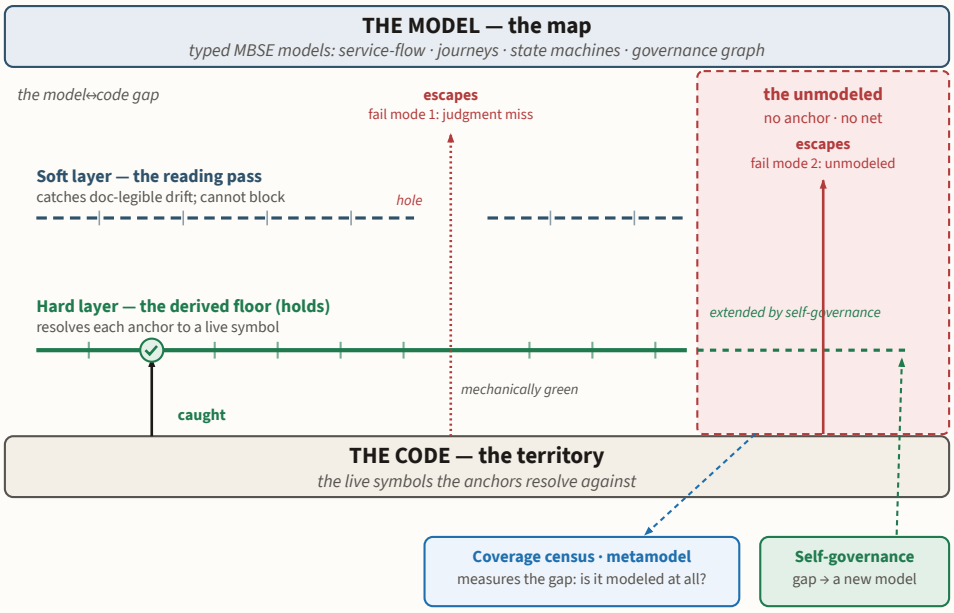


Figure 3.8-1: The Two-Layer Net. *The model↔code gap spanned by two layers, with two escape hatches drawn as gaps in the net. Derived defends, snapshotted drifts — and what isn't modeled can't drift-check.*

3.8.5 The two failure modes: where drift still escapes

The net has two escape hatches. Neither is a footnote; they bound the claim.

Failure mode 1 — the judgment miss. A mis-alignment that is not mechanically decidable needs subjective judgment. The anchor resolves (the code exists, so the hard floor reads green), yet the *meaning* drifted. The soft reading layer is the only net for this, and it is fallible by construction — a

subtle semantic drift can pass a green definition-of-done. This is the drift that slips through a hole in the soft layer.

Failure mode 2 — the unmodeled. A fact or relation that *should* be modeled but is not has no anchor, so no drift-check can fire over it. You cannot detect drift of a thing you never modeled. Here the net has no coverage at all — not a hole in a layer, but a region with no layer. When there is no clean symbol to anchor to, that absence is the signal: it names a missing point of abstraction, a refactoring target.

The systematic backstops (cross-references)

Both failure modes lean, by default, on an engineer or an agent *noticing*. The book has two mechanisms that convert that noticing into machinery — the right in-book targets to cross-link:

- **The coverage census** (`control-coverage-census`, appendix-b) measures failure mode 2. It classifies every control by which target it guards, derives the classification from each control's own anchor, and reports an empty target as a first-class finding. It turns "the estate's blind spot" into a queryable map instead of a thing you learn about when it bites.[46]
- **Self-governance — convert failures to controls** (the governance-conversion / "failures become machinery" loop; the reflection-facet substrate that extends the governance graph, and chapter 2.6′s treatment) converts a recurring gap into a new model or control. Where the census *measures* the gap, this *closes* it: the unmodeled fact becomes a modeled one, and the net gains a segment it did not have.

3.8.6 The honest reading

The result is narrow, and stating its limits is part of stating it. Four readings bound what this measurement does and does not show.

- **Preliminary, small N.** Per window, the count of *genuine* model↔code drifts is small — a handful, not a headline. The cumulative 23/23 catch-rate mixes doc-legible catches with modeled ones; the model-sync claim rests on the small set of genuine model drifts the derived floor caught, plus zero model-drift reopens under heavy churn. This is a field-report result, not a proof. Do not quote "23/23" as a model-sync catch-rate — it is a *combined* rate, and its denominator is small for the model-sync slice.
- **The catch-rate rose by composition, not by loosening.** The refresh window caught more drift (17 of 20 closes vs. the original's 6 of 36) because it moved more territory — product-heavy Epics with more doc surface — exactly where the reading layer is the only practical control. The net did not get weaker; there was more for it to catch.
- **The claim is narrow and it is about sync, not hygiene.** The three legs: the drift is real and used to escape (the ~27 pre-floor findings); the mechanism is derived reconciliation, not snapshot-and-sync (liveness as a property); and it held under load (the churn row, with zero model-drift reopens). The doc-hygiene aside is kept separate on purpose.
- **The two failure modes are the boundary of the claim.** Sync is enforced for the modeled and mechanically-decidable; it is *aimed* for the semantic; and it is *absent* for the unmodeled until a

[46] A note on metamodels. An object model maps the code — service-flow, journeys, state machines — and its drift-check asks "does this model still match the code?" A metamodel checks the completeness of the model set over the territory: "is the right thing modeled at all?" The coverage census and the missing-model metric are metamodels in this sense — the systematic backstop for failure mode 2, converting "an engineer happens to notice something isn't modeled" into "a metamodel measures the coverage gap."

metamodel or a self-governance reflex extends the net. That boundary is the honest shape of the result.

That honest reading closes the zoo. This Part walked each kind of model on real code and showed the drift gate that holds each one to the code, so the map cannot quietly rot away from the territory. What it has not shown is how you reach that state from a codebase that carries none of it. The next Part starts from spaghetti.

Part 4

Putting It to Work

4.1 Brownfield

You have the mindset, the governed environment, and the models; this Part puts them to work, and it starts where most readers actually stand.

Everything so far has assumed a clean start. Greenfield is the easy case: sketch the new thing, have an agent vibe-code a first prototype, and work your way down into an increasingly governed environment, avoiding the mistakes your last project taught you. Most readers, of course, are not on a clean start. Most are staring at brownfield — a legacy codebase, started before agents existed, and since made worse by an agent that added a hundred thousand lines nobody can read.

That codebase comes wrapped in documentation: tests of uneven quality, an architecture known mostly by reputation, and a body of writing everyone consults but nobody quite trusts. Usually that writing is a wiki. It matters because of the distance involved. The path from code to a complete executable model is long. The path from code to a *better wiki* is short. The wiki already exists, engineers already know how to use it, and everyone already agrees it should describe the system more accurately than it does. So most engineering organizations already own the beginnings of a governed engineering environment, one where engineering knowledge and policy live in shared models, tools, and checks instead of in a few people's memory. They just call it the wiki. This chapter starts there. It turns a drifted wiki into a linked and increasingly trusted picture of the system, then carries that picture toward the stronger models and mechanisms the earlier Parts built.

Neither high deployment velocity nor the risk that rides with it is new. The industry learned years ago that shipping to production hundreds of times a day is possible, and that it is unforgiving when the discipline behind it slips. High velocity, high risk: just ask Amazon, which was deploying to production every 11.6 seconds a decade before an agent could write the code for you.

MAGE rests on two moves. The **Modeling Thesis** makes intent and system structure explicit in representations an agent can reason through. The **Alignment Thesis** turns important obligations

into mechanisms the environment applies to later work. Brownfield adoption runs both moves incrementally over the artifacts an organization already has: first make the map more explicit, then add the checks that keep the map and the code from drifting apart.

> ***Where this chapter fits***
> MAGE separates two concerns.
>
> - **Models** give an agent a compact representation of what the system is and what it should do.
> - **Mechanisms** keep the implementation aligned with those representations.
>
> This chapter builds both from what you already have. The wiki is an accessible first model; links and audits give it traceability; lints and completion checks hold the repaired relationship in place; selected high-value claims later graduate into structured, executable models.

4.1.1 Start with the model you already have

Do not dismiss a drifted wiki as failed documentation. Treat it as an immature **model** — a cheaper representation of the system that keeps the structure some engineering purpose needs and drops the rest.[47] A wiki qualifies. It is already a lightweight knowledge graph: its pages are nodes, its links and backlinks are edges. Its tags and categories carry loose metadata, and its search makes all of it navigable. The semantics are weak, but the graph is already there.

What the wiki lacks is a disciplined join to the territory. A page describes a service but never names the code that implements it. A test guards an invariant but never links the decision that says why the invariant matters. A subsystem runs in code with no explaining page at all. The graph floats beside the repository instead of joining it.

[47] Recall that a model here need not be formal or executable — a diagram, a wiki, or a prose spec all qualify, as long as the properties it claims to represent are clear.

[48] The earlier Parts develop the stronger form of this join: every model node points to the code it governs, and that code points back to the model node that explains it.

So the first brownfield move is not to replace the wiki. It is to connect it. Establish traceability as a round trip: the important wiki claims point down to the code and tests that realize them, and the important code and test regions point back up to the pages that explain their purpose. This is traceability in the book's specific sense — a two-way join, not a heap of hyperlinks.[48] Figure 4.1-1 draws the join, and draws why the wiki you have and the model you want are the same shape at different strengths.

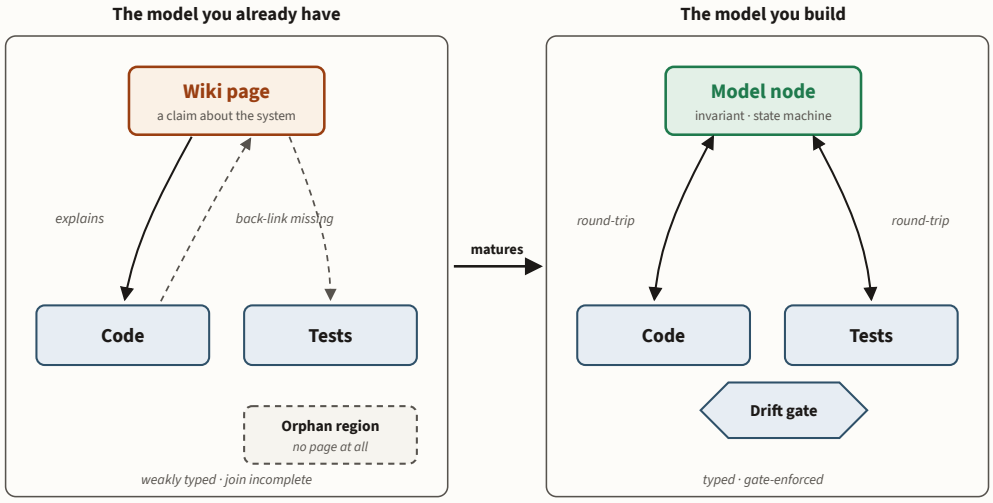


Figure 4.1-1: The wiki is the model you already have. *Pages, links, and backlinks already form a graph; the first move joins it down to code and tests, the same round-trip join a structured model carries. Audit, Synchronize, Govern, and Extend mature the left panel into the right.*

Once the links run both ways, an audit can sort every gap into one of three cases:

1. a missing page or model;
2. a missing anchor to a page that already exists;
3. code that sits legitimately below the representation's grain — too fine or too generic to earn a page of its own.

The distinction keeps the audit honest. The goal is not one wiki entry per function. The goal is a map whose claimed surface agrees with the territory it represents.

The minimum package

Four small additions turn the wiki from optional documentation into the start of an engineering environment.

- **Bidirectional links.** Wiki pages name the code and tests that back their claims; code and tests name the pages that explain their purpose.
- **Agent briefs.** Agents are told how to reach the wiki, how to walk its links and tags, and when a page's claims should shape the task in front of them.
- **Design templates.** New work names the wiki pages it touches, and records when a missing page has to be created.
- **Definition-of-Done checks.** Before work is accepted, the affected pages are read against the current repository, not trusted because an agent reports it updated them.

One rule governs all four: **never add metadata without adding consumers.** A tag, a backlink, an ownership field, a machine-readable mirror — each earns its place only when an audit, a brief, a retrieval step, an analysis, or a gate reads it. Metadata with no reader is one more surface free to drift.

Audit, synchronize, govern, extend

The migration runs in four stages. Each one reaches a useful stopping point, so an organization does not have to arrive at executable models before the work starts paying for itself.

Audit. Spend a bounded set of agent sessions building the graph. Inventory the major wiki regions, find the code and tests each one claims to describe, add the links in both directions, and create or propose pages for the large implementation regions with no explaining home. The exit criterion is coverage of the surface that matters: every major wiki region links to the code and tests that realize it, and every major code and test region is linked, marked below the wiki's grain, or recorded as owing a new entry. This is an audit, not a gate. The wiki is known to drift; making it mandatory before repairing it would only promote old errors to official instructions.

Synchronize. Once the graph exists, drain the drift with ordinary feature and maintenance work. An agent that touches linked code or tests reads the attached pages, checks their claims against the implementation, fixes the nearby factual errors, adds the missing links, and flags the disagreements that need a human to settle. This is a boy-scout rule, not a whole-wiki rewrite: leave the linked surface no less accurate than you found it. The exit criterion is a fresh audit that shows the important wiki claims and the repository substantially in agreement, with no major orphan left unclassified.

Govern. After the wiki earns trust, promote it from optional context to required input. Agent briefs now call for task-specific wiki analysis before implementation. Design templates name the pages the work should touch. The Definition of Done requires the relevant claims to be checked against the repository at HEAD. The promotion order is the one this chapter later applies to lints: audit first, drain the findings, then make it mandatory. This is the Alignment Thesis applied one increment at a time — an obligation that used to depend on someone remembering to update the wiki becomes part of the environment's acceptance criteria.

Extend. A trusted wiki can stay a human-facing model for as long as that serves. Where the value justifies the cost, selected pages go further. Add structured correctness criteria, invariants, dependencies, ownership, allowed transitions, evidence obligations, and links to the validators or tests that discharge them. Mirror those fields in a machine-readable form once a real consumer exists. The data can then drive task-specific context retrieval, drift analysis, test-obligation derivation, semantic lints, or admission gates. The exit criterion is not "the wiki holds structured metadata." It is sharper: every structured field added during extension drives at least one retrieval path, analysis, generator, validator, or gate.

4.1.2 The starting point and the destinations

Name the starting point and both destinations before you move.

The starting point is familiar: some spaghetti, a drifted wiki, thin or uneven tests, and invariants that live only in one person's head — the person you hope never retires.

The first destination is a trusted map. The major wiki claims link to the code and tests that realize them; the major implementation regions link back to the pages that explain their purpose; agents read and maintain those joins during ordinary work. You reach it in weeks, and it pays off long before the far destination arrives.

The far destination is the full model-based environment the earlier Parts built. You have the models needed to express every view of the system that bears on a quality you care about — performance, correctness, security, privacy. You want code that reflects those models, so the map equals the territory — the code agrees with every property the model claims to represent. Zoom in on the map and you see county lines, not Switzerland. And you want **traceability** for your quality case, the ability to state the argument automatically. For any property you claim, you can point to the code that implements it, the static analysis that enforces it, the protocol you model-check it against, the tests that confirm it across the inputs you try, and the record of every change and why it was made. With that in hand, soft and hard governance hold the whole thing in place and keep it current.

The wiki path does not replace the three migration approaches that follow. It gives them somewhere to meet.

- **Top-down** work contributes the intended architecture, the requirements, and the correctness claims.
- **Bottom-up** induction finds the structure the implementation actually holds.
- **Lint and coverage** make the code legible and safe enough for the two to converge.

The wiki is the first shared surface where those views can be compared. Some organizations stop at a trusted, linked, human-facing map. Others keep going, pulling the highest-value claims into typed, executable, drift-gated models. Either way, a real migration uses all three approaches, so take them one at a time.

4.1.3 Approach one: top-down, from a whiteboard

Start coarse and refine. Your first model can be as crude as "on this input, the test suite passes and we get the right output" — and you should be able to write that today. From there you ask what happens *between* input and output: insert a file, and out comes a score, a spreadsheet, whatever your program produces; insert a picture, and out come the cats and antelopes it found. Refine until the model captures the abstractions you need, and stop when you hit diminishing returns — the territory is the territory, and the map only has to be detailed enough to carry the assurance you are after.

You can seed this from a literal whiteboard: photograph the diagram, hand it to the agent, and have it produce a data-flow diagram in whatever representation you like. Then iterate against the code — "what nodes am I missing, what flows am I missing?" — and tighten the policies as the model sharpens: static assertions ("we never call `exec`"), then the invariants of each module (its preconditions, its post-conditions), then the properties preserved across module interactions. As you do this, every mismatch between model and code means one of exactly three things: the model is wrong and needs refining; the model is at the wrong level of abstraction for what you are checking, so you need a different model; or there is a genuine bug in the code — and the modeling just found it. Expect to find many. Shifting a legacy codebase from "the tests pass" to "these properties hold across all behaviors" surfaces defects by design. Some you will fix; some you will decide are working-as-intended and fix the *model* some you will scope out by narrowing the model's assumed inputs. All three are legitimate moves.

4.1.4 Approach two: bottom-up, inducing the model from code

The second approach starts from the code and flies upward. Ask the agent to induce the model: "look at the codebase and help me develop the model inductively." Where the first approach starts from your coarse whiteboard understanding and refines *down* toward the code, this one starts on the ground and rises. Think of standing on a street and flying a drone up: the individual buildings blur, the street grid appears, then the highways, then the cities. As you climb, the levels of abstraction reveal themselves — and whether you came top-down or bottom-up, you are trying to meet in the middle, at the level that lets you enforce the properties you care about.

Which level that is depends entirely on the assurance you need, and the trade runs in proportion: the finer the model, the stronger the guarantee it can carry — and yet the more it costs to write and to run. For low assurance, just run the tests — the coarsest model is "this input, that output, checked dynamically." For the strongest guarantees, you need elaborate analyses over the control-flow graph and every function's inputs and outputs — expensive, because the checker must search every acre of the whole county. So begin by writing down what *correct* even means, because correctness is not one thing: it means one thing for security, another for semantics, another for privacy, and the definition tells you the abstraction level you need. Memory safety demands a fine-grained model that sees every access — and you will pay for it. A semantic property is often far cheaper.

Think of an audit trail on a spreadsheet: "every modification is stamped." You do not model every function to enforce that. You just require that every code path which modifies the ledger calls the stamp routine on the way out — a property about the *order* of two calls in the control-flow graph, not about what either call does. That is a high level of abstraction and therefore cheap to enforce. The companion catalogue ships this exact pattern as a product sensor: a lint that checks every format-mutating routine is wired to its provenance stamp, so an unstamped mutation cannot ship. The catalogue names this mechanism the provenance-stamp wiring lint. Proving the stamp routine *itself* is correct would need a much finer model — but if a human review or a test covers that, the coarse model is enough. Choosing the fitness function, then the abstraction that enforces it cheaply, is the new job, and it varies by codebase. It is yours to think through.

4.1.5 Approach three: lint, cover, then induce

Both approaches assume you can make sense of the code — and often you cannot, because it is a mess. Models read code the way humans do: they like comments that give rationale, names that reflect purpose, and shallow nesting, because deep nesting has to be held in working memory, and an agent pays for that context just as a human pays in confusion. So before you extract models from spaghetti, improve the code — and there is a loop for that, the same loop from Loops and Models. Define the inputs, the judgment, the output, and the metric. The metric here is a **smell detector**.

A smell is code that works but will break the moment it is touched, because no one can reason over it. You measure smells with cheap static analyses — **lints** (the dryer-lint kind, the stuff you pull out to make things tidy) — that parse the syntax tree and count the tells: a high density of bare `int`s and strings, deeply nested conditionals. Some rules are universal. No codebase should have a 2,000-line function, a module imported by every other file, a switch with 400 string cases, structs nested three anonymous levels deep, or functions returning unnamed many-element tuples. So run the lints, and have the agent fix them in a behavior-preserving way — which means you need tests. If you have

none, start there: have the agent write tests for line or branch coverage, module by module, because coverage is what lets you refactor safely, using the tests and the lints together as your success metric.

Squash, zero, promote

Figure 4.1-2 draws how a lint is worn into a legacy tree without breaking any in-flight agent — the order is what keeps it safe.

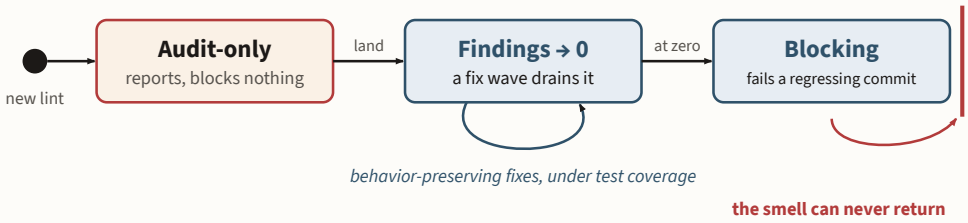


Figure 4.1-2: Audit, Drain, Promote. *A new lint lands audit-only — every finding reported, no commit blocked, so it never breaks an in-flight agent. A fix wave drains it to zero, and only then is it promoted to blocking, so the class of smell can never return.*

Then comes the move that makes it stick: **as each lint reaches zero, make it blocking.** Now the success metric is "all tests pass *and* that lint still passes" — no more unnamed tuples, no more imports buried mid-function, ever again. This is the Alignment Thesis worn into a legacy tree one lint at a time: each promotion is a governance mechanism that bounds what any later agent can reintroduce. Order matters, and the companion repository holds to it deliberately — flipping a lint to blocking while findings remain would break every agent's commit at once, so the figure's sequence is the safe one: audit-only, drain to zero, then promote. The smells that leave never return. When you are done, you have cleaned-up spaghetti with enough shape that model induction has something to grip: classes and enums and named structs give the induction real structure to reason over, the structure that spaghetti hid. A companion mechanism closes the loop from the other side: it projects your test coverage onto the model's nodes, so an untested invariant shows up as a visible gap rather than hiding inside a line-coverage percentage. The catalogue names it coverage-to-model mapping.

The metric that points at what you missed

Induction gives you a first model, but it never tells you where it stops. You built the map of the concurrency-critical spine and the durable-state contracts; the rest of the code sits below the horizon, and nothing in the model announces its own edges. You need an instrument that points at the blank spots. The **Missing Model Metric** is that instrument, and it is the exact dual of coverage.

Coverage, run backwards

Coverage walks one direction: model to code. For each thing you modelled, find its code and ask whether a test touches it — that finds the *modelled but untested*. The Missing Model Metric walks the other way, code to model. For each test, reach the production code it exercises and ask whether any of that code traces up to a model node — an invariant, a state machine, a service edge. A test whose

code reaches no model node is an **orphan**. Cluster the orphans by the unmodelled code they share, and you hold a candidate list of the models you have not written yet.

The duality is the point, and it is what makes the metric a modeling instrument rather than a testing one. You move a coverage number by adding a test. You move this one by adding a **model**. An orphan is not a hole in your test suite; it is a hole in your map. So the metric seeds the modeling backlog directly: it reads out where the induced model still owes the system a view.

Data from the case study

A pilot makes the shape concrete. Run it over the most heavily-modelled slice you have — for this system, the dispatch and orchestration subsystem, across 144 tests — and you would expect near-zero orphans, because if any subsystem traces cleanly this one should. It reported **56%**. More than half the tests in the best-modelled corner exercise code that traces to no invariant, state machine, or service edge. A second pass sharpened that from a scolding into a map. Every orphan reached code the model *did* know — anchored at the coarse structural level, where a component owns a directory — but that carried no fine behavioral model. The structural map was complete; the behavioral map covered the concurrency-critical spine and stopped. The metric had found a **granularity gap**, and it named exactly where to close it: three genuine candidate models — the durable-state data layer, the subprocess-and-cost-rollup pipeline, the chunk merge-and-validation gate.

What earns the instrument its trust is that it discriminates. Two of the clusters it surfaced were *not* missing models. One was code below the model's grain — typed-id aliases, config loaders, enums — code that carries no behavior of its own to model. The other was a missing *anchor*: code that belongs to a node the model already has, just lacking the link back to it. The metric separates the three: missing model, missing anchor, and legitimately below the grain. Without that discrimination it would flag every unmodelled line and drown you. With it, the orphan clusters read as a work queue you can act on.

I ran the metric on my own system, as the last rung of the road in The Road to MAGE. There it drove a loop that closed the first several clusters and, in the process, caught real shipping bugs the passing tests had missed. That is the inward turn of this whole chapter: the brownfield recipe applied by the codebase to itself.

You will need all three approaches in combination. There is no magic in them, only hard work and judgment. But applied together, and scaled to the quality of your starting point, they land you at the destination: the right models, encoding the right views, with the right correctness criteria, traceable to where each is enforced — statically, dynamically, or by a model check.

I owe you honesty about the evidence here. This is a solo field report, not a controlled study. I did not run the migration twice, once with governance and once without, and measure the delta — I had one codebase and one pass through it, so I cannot hand you a clean before-and-after number. Take what follows as lived experience, weighed as such.

How much governance is enough

Which raises the real question: how much governance is enough? There is no formula. There is a judgment, and it turns on two factors multiplied together — how much a failure *costs* you, times how *often* it happens. Cost is more than product risk, of course. An agent that trips over a failure burns tokens working around it, and tokens are real dollars, so a cheap-looking failure that every agent stumbles into ten times a day is expensive in a way the incident log never shows. Multiply the two. The costlier and the more frequent the failure, the harder the mechanism it earns — a blocking lint, a gate, a structured model that makes the mistake impossible. Low times low you leave to judgment; a

mechanism there costs more attention to maintain than the failure ever costs you. The middle band is where taste lives, and where you will spend it. This is also when to *stop*: a mechanism guarding a failure that is both rare and cheap is the first brick in the teetering tower The Skills warns against.

Draw the two factors as axes and the judgment gets a shape. Table 4.1-1 crosses how much a failure costs against how often it happens, and each cell names the proportionate move — the smallest change that still closes the class. Two of the cells carry the part people skip. In the costly column the first move is not a control but the architecture: make the mistake impossible before you reach for something that catches it, because a caught mistake still happened. And the costly-and-frequent corner earns *both* — the structured model that forecloses the mistake and a gate that holds the line while the model is still spreading. The reflex to reach for a control on every failure is the expensive habit; float the larger scheme, and let the cost of the failure justify building it.

Table 4.1-1: The Sizing Matrix. *Failure cost against failure frequency, with the proportionate move in each cell. Reach grows with the product: leave the cheap-and-rare corner to judgment, reserve architecture-plus-a-gate for the costly-and-frequent one — make the mistake impossible before catching it.*

Cost × frequency	Rare failure	Frequent failure
Cheap	Leave it to judgment. A mechanism here is the first brick in the teetering tower.	A cheap lint or script. It trips agents often, and the tokens they burn working around it are real cost.
Costly	A runbook with a named escape; a blocking gate once the failure's shape is decidable.	Architecture first: a structured model that makes the mistake impossible. Then a gate on top. This corner earns both.

The litmus for when to stop falls out of the same two axes:

> Before you build a control, price its own upkeep. Reach for the smallest change that closes the whole class, not the one failure in front of you, and float the larger scheme until its cost is justified. If the control would guard a failure that is both rare and cheap, its upkeep will cost more attention than the failure ever will — so leave that one to judgment.

4.1.6 How I know this

I know this because I lived it. By April I had 300,000 lines of code that were very janky — a Rube Goldberg machine that worked but was a Rube Goldberg machine, because I had started by needing an MVP to see if the thing was possible at all. I had imposed real control only on the parts I knew were sensitive, where I pushed hard for a strong architecture. The rest — the web layer, the front end — I hoped the agent would just get right. It did not, and those parts turned crufty. I told it to use typed Python; it did, and it was typed, in a sense. It specifically used the type `string`. That was the type it used. So it was stringly-typed: no actual safety the compiler could give you, just the word "type" satisfied to the letter and voided in spirit. I told it to retrofit real types; it turned on the type checker and made everything strings and ints, without building the type abstractions I actually wanted.

So I spent a month retrofitting these ideas onto the code base, and when I was done I had 200,000 lines of code that were not janky. The system got smaller as it got better — structure is compression, and the primitive sprawl was the bloat. Those round numbers, 300,000 janky lines in and 200,000

clean out in a month, are my April estimate of the episode, not a reconstructed count. What lifts them past memory is that the project's measured churn corroborates their shape. For the data, see The Timeline and the Work → (preliminary) The add-and-delete peaked exactly while I did this work, then the deletions collapsed and the later windows went net-additive — the signature of a one-time replacement, not steady thrashing. And the churn was productive: what I deleted was the stringly-typed sprawl, what replaced it was modeled, typed structure, which is why the code shrank as it improved. So read the numbers as the estimate that churn backs, not a hand-counted ledger — the add and delete totals sweep in generated and vendored files, so they signal motion, not hand-authored lines. It is real because I did it. The same thing had happened porting JavaScript to TypeScript that same week — a hundred thousand lines of "typed" Python and a hundred and fifty thousand of TypeScript, each ported the cheapest way possible, because the agent had no model and did not know the names of things. It is probably better that it did not guess.

So I retrofitted the models onto the mess — moving the code toward a clean model-view-controller architecture, moving the web stack to reactive, loop-based structures friendly to auto-scaling. And I did it exactly the way this chapter describes. I turned on every lint from Ruff, Pylint, and ESLint — hundreds of commodity smell checks — and for several weeks the agents squashed them, making each one blocking as it went to zero so it could never come back. I knew the coarse architecture I wanted and said "pursue this model," but I had to get there iteratively: find the dense primitive regions, induce structure there, and inch toward the target.

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 found these the crude way, by reading for the clots: where the primitive calls piled up thickest, the missing model was loudest. Inducing structure means extracting a structured model that owns those primitives — one class the raw calls now route through — and rewriting the region to go through it. You do this region by region, not all at once, because each induced model gives the next region something to lean on, and the target architecture assembles out of the pieces. That is what the squash-to-zero does at the fine grain, too: each lint I drove to zero was one primitive smell given a name and a home, and the model-view-controller decomposition is just the same move at the coarse grain. After a while, it worked. Now I have the models, the architecture, the code that implements it, and enforcement holding all three together every time the code changes.

In MAGE terms, that month was governance conversion at brownfield scale: the recurring failures and the same rediscoveries, over and over, became the models, links, lints, and gates that now shape every later task.[49]

[49] Governance conversion turns a recurring failure class into a mechanism the environment applies for you, so a later agent meets the check instead of rediscovering the failure.

4.2 The Skills

A method only helps while someone remembers to apply it. Left as prose in a book, it has to be re-taught to every fresh agent and every new project by hand, and a discipline that lives in memory rather than in the environment is the first thing velocity drops. So a book of method ought to ship with batteries, and this one does. There is a repository that goes with it, and it contains three Claude skills you can install and use today: **self-operate**, **self-governance**, and **self-communicate**. Each is the methodology of the preceding chapters, packaged as something an agent triggers and follows. This chapter walks all three.

These are an unusual kind of skill. You have probably seen plenty of *tool-skills* — a skill that teaches the model an interface: to do XYZ with this MCP server or CLI, here is the shape of the call. What follows is a different animal: a *mastery-skill*, which installs a body of knowledge and the judgment to do a whole task *well* — not an interface to one tool, but a way of doing the work. That is why the three below take a chapter to walk, where a tool-skill would take a paragraph.

> **Two kinds of skill.** *A* tool-skill *teaches an interface — the calls, flags, and shapes for operating one tool (an MCP server, a CLI, an API). A* mastery-skill *installs a knowledge-base and the judgment for doing a task well, independent of any single tool. This book's three skills are mastery-skills; Appendix E is the full recipe for building one.*

The three are complements: one hardens the fleet, one runs it, one explains it — the same system seen from three angles, drawn in Figure 4.2-1.

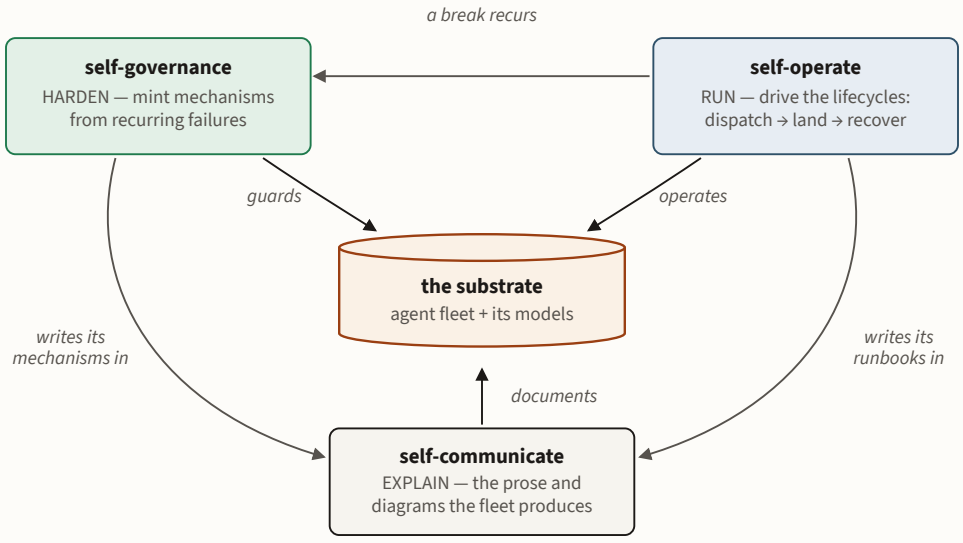


Figure 4.2-1: The Three Skills. *One substrate, three complements: self-governance hardens it, self-operate runs it, self-communicate explains it — and the skills feed each other, a recurring break becoming a mechanism written in the register the third governs.*

4.2.1 Self-operate: the lifecycles, made runnable

Self-operate is the starting point, because it holds the lifecycles of running a repo — everything an engineering team actually does. Pull a ticket and build a feature. Pull a ticket and fix a bug. Optimize the codebase. Handle resource exhaustion. Each of those is written down as a model — I did say models everywhere, and I meant it — because that is how you constrain the agent's search and bias it toward doing the work the way you want.

> Learn more about this governance mechanism: self-operate (operator runbook skill).

On top of the models sit the playbooks, and there is a playbook for everything, each with briefs for its judgment steps. This is what lets an orchestrator agent — itself running as a reactive loop — know how to respond when an event arrives. An agent finished? Reach for the next step: *agent landed*. Is the feature done? Open the Epic file, which lists the launches needed to complete it, and check. Not done — launch the next item. Done — run the definition-of-done audit over the whole Epic, the one that asks, among other things, whether the model drifted from the code. Passes — mark it done and move it to the closed pile. Still active — record the status and the commit that made progress, and launch the next agent from the Epic's list. The playbook keeps the loop turning on inputs from the environment, which is precisely the *dispatch → work → land → tombstone* lifecycle from Loops and Models, now written down for the operator to run.

That definition-of-done audit exists because of one hard-won rule.

> "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. So you do not trust the claim. You re-run the gates yourself, at HEAD, and let a fact answer where a claim was offered.

That rule is one of several the skill teaches. Three worth quoting, because they show the shape of the whole:

> ***Self-operate — three principles***
> - **Determinize the runnable, brief the judgment.** A runbook step is one of three kinds. If a machine can do it, make it a runnable line. If it needs judgment a brief can carry, write the brief. If it needs judgment no brief can carry, escalate. A runbook that blurs the three sends a human to do a machine's work, or a machine to make a human's call.
> - **The lifecycle is a state machine, not a habit.** Dispatch, work, land, tombstone — an agent's life is a sequence of states with legal transitions, and you write it down as one. A habit lives in the operator's head and rots; a state machine names every state and the move out of it, so the loop turns on events from the environment instead of on memory.
> - **Orient positive before you hunt a break.** Know the healthy baseline first — what "normal" looks like for each lifecycle. Most sessions are steady-state; you cannot recognize a break until you can state the shape it broke from.

4.2.2 Self-governance: the catalogue and the posture

The second skill, self-governance, triggers on a simple signal: a bad thing happened more than once — sometimes just once, if it was bad enough. Recurrence is the clue that you are either failing to *detect* the problem (you need a sensor) or failing to *prevent* it (you need a constraint, a wall). You have not put the wall in the right spot, or you have not put a badge reader on the door.

At its center is a catalogue of governance mechanisms and worked examples — around eighty at last count, and still growing as new failures surface new mechanisms. I did not know these were the mechanisms I would need until I started building and kept hitting failures; the higher the velocity rose, the more failures appeared, and new *kinds* of failure with them, and each mechanism I invented to kill one went into the catalogue. The hope is that this becomes ex-ante governance for *you*: the mistakes I made and solved, you inherit for free — install it and say "audit my codebase and help me find room for governance." You will, of course, find your own failures too, because you run a different model, different constraints, different customers, different operators, and all of it shapes your agent's behavior. So the skill ships not just the catalogue but a **posture** — the heuristics for minting new mechanisms well.

Where do the minted rules live? In a project's `CLAUDE.md` — the one document loaded into every agent's boot context, so it is where governance either binds or does not. Its structure is three parts, coarse to fine. First a **Mission**: what the project optimizes for, the values that break a tie. Then a **portable method**: the engineering principles that hold across any project of this kind, independent of the domain. Then the **numbered project rules**: the specific, stably-numbered guardrails this codebase enforces, each a one-line boot-context statement pointing at the deeper doc that carries it in full. Mission orients, the method teaches judgment, the rules bind — and a self-governance skill grows the third part as failures earn new rules. If you want a scaffold to start your own, the CLAUDE.md rule index appendix ships a **CLAUDE-starter** download built from a real, mature one.

The posture: minting new mechanisms well

The first heuristic is to **avoid a teetering tower of governance.** Do too much of this and you end up in trouble: so much code governing and constraining the behavior that tending the tower becomes the work itself. The failure has a specific shape. The agent starts spending its time patching and fixing the guardrails — *was that even a mistake?* — instead of doing the work you actually want. It side-quests off into optimizing the linter's behavior over here while the product you meant to build waits over there. So part of your judgment as the operator, as you construct the governed environment, is to watch for exactly this: are we making high-velocity progress in the direction we want, or have we wandered off to polish a guardrail nobody asked for? Finding the sweet spot is your judgment, and no mechanism makes it for you. Ask how detailed the check should really be. Is this generic or specific — do you need a lint for "never type the word X"? Almost certainly the wrong level, but a lint that blocks typing anything is worse. Decide whether false positives or false negatives hurt you more, because that governs which mechanism fits. And when you say "this must *never* happen," you are choosing architecture — a wall — and every wall constrains the model. A well-placed wall is pure gain. A wall across the only fire exit is a disaster the day there is a fire and no one can get to work. Because of this, the skill will not silently enforce a mechanism without checking with you — you are the engineer who knows the context. Its own trigger is a sensor at the right semantic level: a hook that fires every thirty minutes or so in the orchestrator loop and asks, "since I last checked, did we hit any recurring failure, or solve one problem several different ways?" Solving it several ways is a don't-repeat-yourself smell in the code; hitting the same *process* wall repeatedly — "I failed to create

a VM," again — is a process smell. At agentic velocity you cover enormous ground in thirty minutes, so a repeat inside that window is the signal that a mechanism is owed.

The move generalizes. A skill on its own is *passive*: it sits there and waits to be invoked, and an agent heads-down in the work will not stop to invoke it. The hook is what brings it to life — kaizen on a timer, the improvement loop from The Engineer's Seat run without waiting for a human to notice the recurrence. The timer fires, the prompt lands, and the skill runs — it reaches into its catalogue of every way you might handle this class of failure and, usually, just fixes it. When it cannot decide, it surfaces the fork — "we keep hitting this; here are two ideas, which should I try?" — and waits. That hand-off is deliberate: the hook makes the reflection happen, the catalogue supplies the options, and the judgment call stays yours.

Learn more about this governance mechanism: reflection-facet substrate.

Underneath the posture sit a few reflexes the skill applies on every touch. Three of them:

Self-governance — three principles

- **Convert, don't just repair — audit becomes lint.** When a failure smells class-level, do not only patch the instance. Build the durable mechanism that kills the class. Today's audit finding — a thing you noticed by reading — is tomorrow's lint, a thing the machine notices for you. A bug in more than one file means "fix the sites *and* add the check," not "fix the sites and hope."
- **Guidance aims; machinery holds.** A mechanism is soft or hard. A soft one — a brief, a house-rule — can *aim* a probabilistic agent but cannot block it, and it rots. A hard one — a lint, a gate, a typed seam — *holds the line* regardless of whether the agent cooperates. Know which you are minting, and never claim a mechanism is enforced when you have only recommended it.
- **Architecture before sensors.** Prefer making the error impossible over catching it after. A structured model with one sanctioned seam beats a validator that scolds you for using the wrong one. Where a failure is costly, do both — belt-and-suspenders is a feature. But reach for the wall first, and remember that every wall you raise also constrains the agent, so place it with care.

4.2.3 Self-communicate: how the agent writes and draws

The third skill, self-communicate, governs how the agent explains itself — to you and to other people. Docs are a soft mechanism for an agent, but they are a genuine tool for humans, so the skill carries guidance on the kinds of technical documentation, crediting the Diátaxis project for enumerating them and the Apache Foundation for the exemplars it learns style and voice from. It targets prose that avoids imprecise claims and the grating verbal tics, and that is as terse as it can be without losing the reader. When the agent writes, it turns here. When it needs a drawing, the skill says "use me," and teaches drawing in Mermaid — dropping to HTML or SVG only when Mermaid cannot express the shape. That keeps the agent from trying to hand-generate raw PNGs, a thing it will attempt and be bad at.

The lexicon: one concept, one word

Self-communicate also owns a **lexicon**: the house vocabulary for your repo, so one concept gets one word, used consistently. You can bootstrap it — point the skill at existing material and have it walk the

repository, surface the terms you use with specific meaning, and write down the working definitions. That walk tends to catch two failures at once: different words for the same concept, and the same word for different concepts. Both confuse any reader, human or agent, and finding them may show you mistakes in your own thinking. “There are actually two concepts here — help me name them.” Now they have names, and you can retrofit structure onto the writing — which is, exactly, the code problem from Brownfield, wearing different clothes.

The lexicon is one specialization of a stance the skill carries into both legs. Three of its principles govern the whole:

> ***Self-communicate — three principles***
> - **Less is more — the representation must not distract from the idea.** In prose that means Hemingway: cut the fluffy adjective, the qualifier, the sentence that restates the last one. In drawing it means Tufte’s data-ink ratio and Picasso’s *Bull* — strip the chartjunk, the gradients and bevels that decorate without informing, until only the marks that carry the idea remain.
> - **Name the concept, then use the name.** A name is a handle: it carries a concept’s meaning and its constraints in one token. In prose, reach for the established term — say “circuit breaker,” not a fresh paragraph re-deriving one. In drawing, reach for the format’s native construct — an SVG `<marker>` with `orient="auto"` *is* an arrowhead; it rotates to its line and pins to the endpoint, so it cannot land crooked the way a hand-stitched triangle can. Use the marker.
> - **Decide the doc’s mode before its sentences.** A doc is a tutorial, a how-to, a reference, or an explanation — the four Diátaxis shapes — and it is exactly one of them. A how-to written as an explanation confuses the reader who wanted steps. Fix the shape first; the prose follows.

Those are the three skills that ship with the book. I hope they are of use.

4.2.4 The construction recipe is a pattern of its own

The three feel of a piece because they were all built the same way. Three layers, every time. A **base model of the domain** at the bottom. **Orthogonal models** layered on top — cuts through the base along independent axes, each adding a dimension without disturbing the others. And a top-level `SKILL.md` **of principles** that ties the models together and tells the agent when to reach for which. Set the three skills side by side along those layers and the shared skeleton shows through.

self-operate — the *operate* leg, by layer in Table 4.2-1.

Table 4.2-1: The self-operate skill by construction layer — base model, orthogonal models, and the SKILL.md of principles.

Layer	For this skill	Remark
Base model	the lifecycles a repo runs — dispatch, land, recover, deploy, reclaim	the base is the work an engineering team actually does, written down
Orthogonal models	runbooks (typed steps), the lifecycle state machine, event-bus observability	each cuts the base a different way — one names the states, one names the steps, one names the signals

Layer	For this skill	Remark
`SKILL.md`	orient-positive-first, route-a-break-to-its-class, hand a recurrence to self-governance	ties the models into a loop that turns on events from the environment

self-governance — the *harden* leg, by layer in Table 4.2-2.

Table 4.2-2: The self-governance skill by construction layer — the mechanism catalogue, its orthogonal axes, and the minting posture.

Layer	For this skill	Remark
Base model	the catalogue of governance mechanisms — around eighty failure-killing patterns	the base is a census of failures and the mechanism each one earns
Orthogonal models	soft-vs-hard enforcement, the agent / models-bridge target axis, the form taxonomy	independent axes over the same catalogue — a mechanism has a target *and* a form *and* an enforcement
`SKILL.md`	convert-don't-repair, architecture-before-sensors, right-size the fix	the posture for minting a new mechanism well, so the catalogue grows without teetering

self-communicate — the *communicate* leg, by layer in Table 4.2-3.

Table 4.2-3: The self-communicate skill by construction layer — the prose and drawing crafts, their orthogonal models, and the voice principles.

Layer	For this skill	Remark
Base model	the crafts of engineer-facing prose and technical drawing	the base is two legs — words and shapes — not one
Orthogonal models	rhetoric, Diátaxis doc-shapes, the lexicon, voice, the diagram vocabulary	each is a separable cut: a doc has a *shape* and a *voice* and a *vocabulary*, chosen independently
`SKILL.md`	less-is-more, name-the-concept-then-use-the-name	two stances that specialize per leg — cut the adjective in prose, cut the chartjunk in a diagram

Once you see it laid out, it reads as obvious — of course that is how you build one. Figure 4.2-2 draws the shared skeleton.

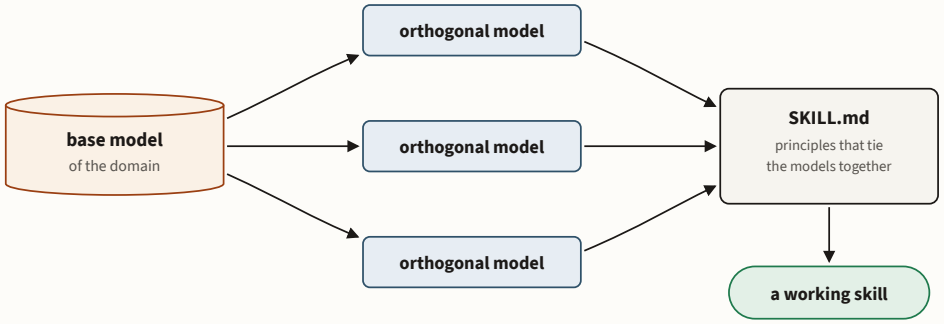


Figure 4.2-2: The Skill Skeleton. *Every shipped skill has the same skeleton: a base model of the domain, orthogonal models cutting it along independent axes, and a SKILL.md of principles that ties them into a working skill.*

Appendix E draws the recipe out step by step, with self-operate, self-governance, and self-communicate as its three worked examples — so the payoff of seeing the shape here is a working procedure there.

> **Left for later: the team dimension.** *This book is written as "you and your agents," but production software is built by teams, and the team dimension is its own chapter. A model that governs one person's work has to become a shared artifact — reviewed, owned, and defended by more than its author. Review changes shape when an agent wrote the diff: who reads it, and against what. And when a model drifts, someone has to own the reconciliation. Those are real questions with real answers, and they are not this book's. This is a field report and a method, not a survey of team practice, so I mark the boundary here rather than gesture past it.*

4.3 Everything Is a Transformation

There is one more habit of mind that makes all of this click, and it comes from the shape of the machine underneath. A large language model is a transformer, and it is extraordinarily good at one thing: turning an input into an output according to a simple-up-to-pretty-complex mapping function that it learns from its training data. So I suggest you frame every task you hand it as a series of transformations — the skill in using an agent is sizing each transformation to the model and to the guarantee you need. This chapter is about that sizing, and about how the three skills are all built from it.

> *A note for the curious.* A "learned mapping function" is simpler than it sounds. Picture the model as one enormous mathematical function: you feed in the words of your prompt, and it hands back the words most likely to come next. "Learned from its training data" means nobody wrote that function's rules by hand — its billions of internal dials were tuned automatically, over a mountain of text, until its output matched the patterns in the data. The particular design that let this approach scale to today's models is the *transformer*, introduced in the 2017 paper "Attention Is All You Need"[50].

4.3.1 Size the leap to the model and the guarantee

Give a model an input and either examples of a transformation or a precise description of it, and it will apply that transformation to something new — as long as the new thing is close enough to the examples, or the rule is precise enough. The stronger the model, the bigger the *leap* it can make reliably. A strong model turns abstract instructions into working code; a weak one turns the same instructions into a dumpster fire, or nothing that compiles. So you tune two things together: the level of abstraction you speak at, and the size of the leap you ask for — against the model's strength and against how much guarantee you want.

The trade has a sweet spot, and you can feel for it by counting reasoning steps. Want high guarantees? Ask for smaller leaps, and check the result of each one as it lands. Want speed? Ask for bigger leaps, and accept that you trade away some of that per-step trust. The smaller the leap, the surer each step lands — and yet the more steps and tokens you pay; the bigger the leap, the fewer the steps — and yet the longer the reasoning each one takes. The right size is big enough to use the model's full power, but not so big it needs an exhausting chain of thought. And there is a hard reason not to let the chain run long: probabilities compound. A one-percent chance of going wrong per step is nothing over two steps and almost a certainty over a hundred. Size the transformation to the task's complexity and the model's scale, and you keep the failure probability where you can live with it.

4.3.2 The right-sized leap, and the wrong-shaped one

Two transformations from this book show the lens at work — one sized right, one shaped wrong.

The remediation core could ask a strong model for the whole leap: here is a document, hand back an accessible one. That is a single enormous transformation, and a strong enough model can, of course, sometimes make it. But it is expensive — the model reasons over the entire document on every call

[50] Ashish Vaswani et al., "Attention Is All You Need," in "Advances in Neural Information Processing Systems 30 (NIPS 2017)," special issue, *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, 2017, 5998–6008.

— slow, and unauditable, because you cannot see which change it made or why. So the system sizes the leap down. It scopes the work to a closed vocabulary of small typed edits — set the alt text on this image, set the role of this block, reorder these children — the same edit language the editor speaks in Part 5. The model is asked for one bounded edit at a time; each call reasons over one scoped task instead of the whole document, and each edit comes back typed, stamped, and reversible. This is where the transformation-sizing lens earns its keep, and changes a decision the loop-and-model lens does not: that lens says *call the model to remediate* the sizing lens says *call it for one edit at a time* — cheaper, faster, and auditable, all at once.

Learn more about this governance mechanism: remediation verbs.

Sizing is only half of it. The transformation also has to be the right *shape* for the task and the model. I learned this the expensive way. For two weeks I sized a transformation beautifully and pointed it at the wrong problem — pulling a PDF's reading order out of the geometry of its bounding boxes, and handing that geometry to cheap vision models, which are bad at geometry. The model cheerfully made a doomed shape better and better. A well-sized transformation of the wrong shape is still the wrong transformation. So the skill has two halves: sizing the leap, and finding the one leap that fits both your task and your model — then abandoning the one that does not.

4.3.3 The skills are transformations all the way down

Figure 4.3-1 shows the shared skeleton — each skill carrying an input to an output through a sequence of sized transformations.

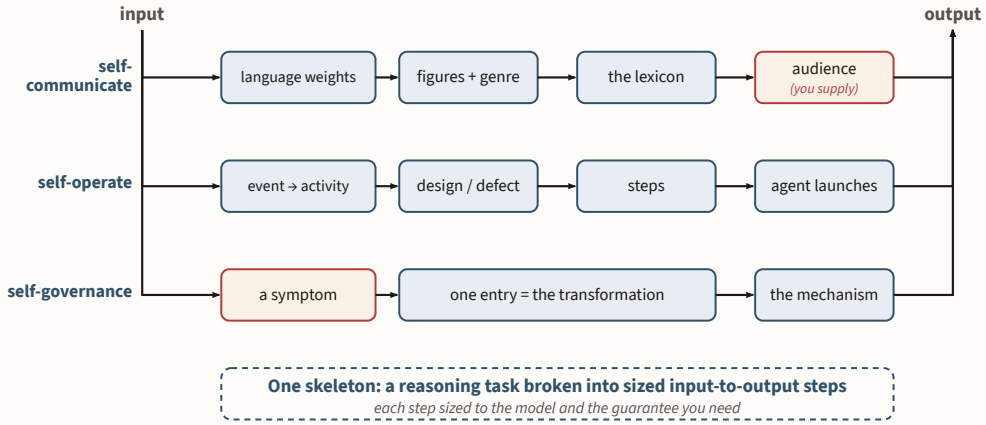


Figure 4.3-1: The three skills share one skeleton: each carries an input to an output through sized transformations. Self-communicate stacks language and genre; self-operate maps event to design to launches; self-governance turns a symptom into a mechanism. Each step is sized to the model and the guarantee you need.

Once you see everything as a sized transformation, the three skills reveal a common skeleton. Each holds a set of models the agent reasons over, and each names the steps — the transformations — that carry an input to an output. What you supply, in your own context, is the breakdown: the playbooks and runbooks for the steps your work actually takes.

- **Self-communicate** stacks its transformations on the model's language weights, then layers the rhetorical figures (the "not X, but Y" among many others), then the writing genre from Diátaxis, then

the lexicon for this context. It leaves the top layer to you: who is the audience, and what do they know? The skill cannot know that; you tell it.

- **Self-operate** puts the lifecycle models at the base, maps an incoming event to the kind of activity it is (development, bug fix, resource management, deploy), and then transforms forward: event to defect characterization or design document, design to steps, steps to distinct agent launches with their briefs. Each arrow is one transformation of input to output toward a goal.
- **Self-governance** holds a model of governance itself — architecture versus sensors, the properties that should hold of any codebase (don't-repeat-yourself among them), the kinds of sensor (static analysis, dynamic tests), the invariants you write over models, the languages you write them in, the checkers that validate them. Each catalogue entry is itself a transformation: from the *symptom* — the recurring failure you are living with — to the *mechanism* that ends it.

That last point runs through the whole book: each skill takes a general reasoning task and breaks it into a series of transformations over inputs into outputs, with steps the model can follow. This is not a coincidence of design — it is the shape of working with a transformer at all. It is why the companion catalogue can be *read* like a lookup table: you arrive with a symptom and leave with the mechanism, because the entry is the transformation between them. Work with agents through this lens — every task a sized sequence of input-to-output steps — and you will find yourself a much happier engineer.

4.4 Training Data and the Novelty Axis

There is one last property of a coding agent you must reason about, and it decides how much of this book you will actually need: its training data. Everything in these chapters is a way of conditioning a probability distribution — narrowing the agent's search space until the path you want is the path it takes. But that distribution did not start neutral. It was shaped by what the model was trained on, and that shape decides where the discipline is optional and where it is the only thing that will save you. The more your system resembles the training data, the less conditioning you owe; the less it resembles it, the more the conditioning is the work itself. This chapter walks that axis, from the common case to the hard one.

4.4.1 The model is biased toward what it has seen

Think of the training data as the whole internet and every book ever written, and think of the model as biased toward whatever it saw most. Read one book holding a position and a hundred holding the opposite, and it leans to the hundred. Except where the trainers worked hard to force a particular behavior, it simply weights the common over the rare. This has a moral dimension the labs fight with reinforcement learning from human feedback — you teach the model not to lie, and it lies anyway; not to steal, and it steals; not to break into other people's computers, and it will, given the chance. Training data has its limits. But the same bias carries an engineering consequence more useful to us: the kinds of systems the model has seen a great deal of, it can build a variation on almost effortlessly; the kinds it has never seen, it struggles with.

That divides the world of software neatly. If you are building your own version of something standard, the model has read the documentation and watched the walkthroughs of a hundred competing products, and it can reproduce the class in its sleep. (One caution: building a competitor this way can infringe a *patent* even though you copied no *code*, because a patent covers the concept rather than the text — so tread carefully.) A to-do list, a recipe tracker, a fitness app — GitHub holds a million of each, and the agent knows them cold. Any "typical web application" — a form-driven UI, a backend storing to a database, a compute layer producing outputs over it — it builds without breaking a sweat, because you are asking it to regurgitate a class it has seen a thousand times and then specialize it. This is exactly why a lab can credibly claim it built a whole C compiler in a week, or ported one runtime to another language in three days: the training data is full of compilers, and in the porting case, full of the very code being ported. Impressive, and entirely unsurprising.

I watched this split inside my own project. DocAble had to drive two families of document format the model had barely seen used in anger: the OpenXML packages behind PowerPoint, Word, and Excel, and the internals of PDF. It tripped over its own feet for weeks on the semantics of both — the OpenXML relationship graph, the PDF content stream and tag tree — figuring things out by running a PPTX and watching what broke, and, more than once, by reading the PDF specification itself. Hand the same model JSON or XML or CSV or YAML and it never stumbled; it has read a million programs that parse those and writes them fluently. The formats are not equally hard, of course — a tagged PDF is genuinely harder than a YAML file. But the gap in the agent's fluency was far wider than the gap in intrinsic difficulty, and that extra width is the training data. The common formats sit in the fat of the distribution, the document-format internals out in its thin tail, and the agent's competence tracked the data, not the difficulty. That is the novelty axis measured on one machine, and it is why the format models in this system are so heavily specified — I had to hand the agent the priors its training never gave it.

4.4.2 Where MBSE shines: the novelty axis

Figure 4.4-1 sets a part's novelty against the leap you can trust and the oversight you owe.

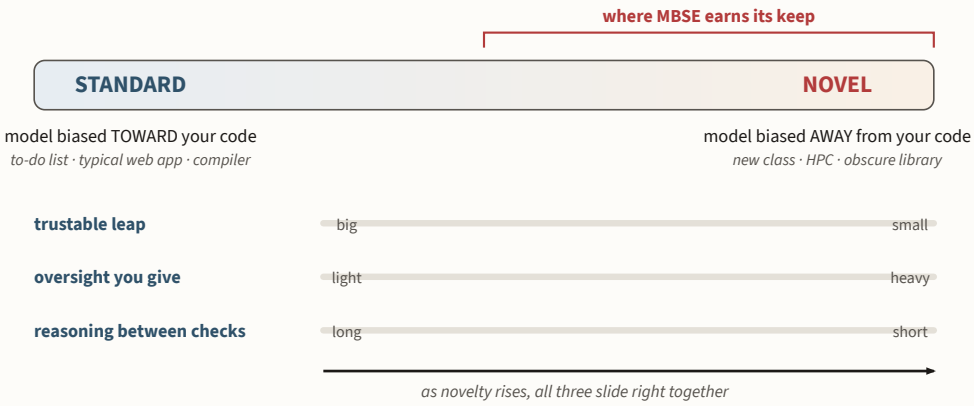


Figure 4.4-1: The Novelty Axis. *A part's novelty sets the leap you can trust and the oversight you owe. At the standard end data is thick and a big leap is trustable; at the novel end data is thin, oversight grows, and up-front modeling is not optional.*

The other half of the world is where this book earns its keep. Build a genuinely new kind of application — a new class of software, something hard nobody has done, or a research setting like high-performance computing — and the bias runs *against* you. There is very little open-source HPC code. The model has read the OpenMP, Open MPI, and CUDA docs and seen a pile of ML kernels, but it has not seen your climate simulation unless your organization open-sourced it, and even then it is one drop in an ocean of everything else. So the foundation model is biased *away* from the code you need. Left alone, it will map the architecture it *does* know onto your problem and make the wrong choices — the right moves for the wrong class of software.

This is precisely where model-based software engineering shines: the model lacks the bias you need, so the job of imposing that bias becomes yours — and the way you impose it is to articulate it. You can, because code is cheap; you have only to tell the agent what to build, with the full rigor of the earlier chapters. The companion catalogue's whole models-bridge exists for this: it is the channel through which you hand the agent the priors its training never gave it, so the invariants you care about are written down where it will read them instead of guessed at from the wrong reference class. On a standard class of software you still model, but you can lean on the model's built-in bias; on a novel one the up-front modeling is not optional, and you coach the agent step by step.

The diffusion analogy

I hold this through an analogy. Think of the diffusion models that draw pictures. Ask one for an exact copy of a famous painting and it delivers, because it has seen that painting endlessly. Ask it for a scene no one has ever painted — a genuinely new combination, or merely a complicated one — and it struggles, because it works from pictures of reality rather than from reality itself. The labs have, of course, pushed these models to reason one step deeper than the prompt — deep enough to know a hand has five fingers, so we are no longer handed humans with seventeen. But they always run up against the gap between what they have seen of the world and how the world actually is. Engineering a system is the same. A system the model has seen a thousand times, like Starry Night, it builds without trouble. A novel system for your startup — whose entire point is to be a new thing that does

not yet exist — it has not seen, and cannot. Say “draw an alien” and it invents seventeen fingers and two heads, because no one knows what an alien looks like, so there is no reality to map to.

Demoable is not productizable

So think of it this way: the agent will hand you a working prototype, but look under the hood of a novel system and you find Rube Goldberg machines in the engine room. It runs — and a skilled engineer is still needed to turn that crude, working prototype into something reliable across the full breadth of inputs and quality attributes you care about. For the standard parts, the model has seen the engine room and gets the internals right. For the novel parts, or the ones shaped by context it never trained on — a powerful but obscure library used commercially rather than in the open — it can read the library but has never watched anyone use it well. So it fabricates something that passes your examples and fails in general. The engine room of a vibe-coded novel system is not correct. It is demoable; it is not productizable — not yet, not without an engineer. That is the job. The more standard the part, the bigger the leap you can trust and the more code you can let the agent produce unattended. The more obscure, sensitive, or context-bound the part, the more oversight you give, the shorter the reasoning you allow between checks, and the more you condition the distribution yourself — narrowing the search by hand, with model-based software engineering.

> **Left for a security treatment: the untrusted agent.** *One thread runs through this book only in passing and deserves its own consolidated treatment: governing an untrusted agent. I have noted that agents will lie, take what they can reach, and break into a machine given the opening, and that a hook can block a network call. That is the edge of a much larger subject. Prompt injection turns your own inputs into instructions. Exfiltration rides out through the very tools you granted, in good faith, to get the work done. Supply-chain exposure arrives in a dependency you never audited. Each is a security discipline in full, and scattering a sentence about them across the book does them no justice — they belong together, in a treatment this field report does not attempt.*

4.5 Lessons Learned

I spent three months building a real piece of software with a coding agent, and the thing I came away with was a methodology, not a model release or a benchmark. I thought the project would take a weekend; it spiralled into a hundred thousand lines, then four hundred thousand, written at a thousand commits a week. Somewhere in that spiral the important questions stopped being *what can the model do* and became *how do I keep it doing the right thing at speed*. This chapter is what I learned. Read it as the reflection at the end of the method — the lessons that hold no matter which model you drive. Table 4.5-1 gathers them; the sections that follow earn each line.

Table 4.5-1: The lessons as quotable lines and the move each one demands.

The lesson	The move it demands
Claude is the fastest road to hell	Keep an open mind about the paradigm; speed only exposes the danger you already chose
Optionality is poison	Decide up front — every unmade decision is a place the agent guesses
"Done" is a claim, not a fact	Re-run the tests and lints yourself; trust nothing at the boundary
An agent fails differently than a person	Build sensor barriers that pin the invariant, not the example
Vibe-coding treats the program as an I/O device	Engineer it as a system — hold the invariants, not just the output
What lives only in your head is invisible to the machine	Make the architecture explicit, queryable, executable
Soft aims; hard holds	A convention that says "never" eventually gets violated at velocity — build the wall
The second time a failure shows up, it has earned its mechanism	Turn the audit into a lint
Amplification is neutral	Convert judgment into infrastructure — encode each call once

4.5.1 Three ways to run an agent, and why I chose one

There are three ways to run a coding agent, and I suggest you choose among them by where you sit on the risk map; taste does not enter into it.

- **Velocity-centric.** Maximum speed, minimal oversight. On a greenfield project with a handful of users — say, just the engineer building it — this is right. Claude whips up a prototype at the speed of the engineer's thought, and that engineer is the only stakeholder whose thought matters. High velocity, low risk. Let it run.
- **Oversight-centric.** A human reads every step. On a brownfield system serving thousands or millions, the risk is extraordinary, and a careful reviewer at each turn is the reflex — but it throws away the very speed you brought the agent in for.

- **Governance-centric.** You define the constraints up front, then let the agent move fast *inside* them. This is the quieter voice, the one that comes from engineering traditions like safety engineering: place bounds on the agent's operations, then let it rip.

I chose the third, and the rest of this book is the argument for it. Governance is the middle path that does not feel like a compromise, because a well-drawn constraint is the very frame that lets you take your hands off the wheel. If you demand that the system get built but refuse to read the code, then either the agents work on the first try (they don't), or you must monitor *something*. I chose to monitor failures. When I saw one, instead of asking Claude to fix it, I asked Claude how we could keep it from ever happening again.

4.5.2 Claude is the fastest road to hell

Coding agents accelerate *within the frame you give them* — even when the frame is wrong. The service was costing thirty to forty dollars a day to run idle on a Kubernetes cluster, so I set the agents on a scale-to-zero push: keep the cluster, keep the custom autoscaler, and squeeze the idle pods down to nothing. For weeks they made it better and better — and it never paid off, because scaling a cold pod back up took sixty to three hundred seconds, and a five-minute cold start is a worse problem than the bill. I was optimizing *within* the architecture when the architecture itself was the constraint; the real move was to change it — go serverless, and let the platform own the scaling I was hand-building against its own ceiling. The model is both an implementation aid and a way to search a solution space, and if you do not keep an open mind it will only speed your descent into the circle of hell you already chose. The tell, in hindsight, was that the optimization kept hitting the *same* wall: when an in-paradigm fix keeps colliding with the paradigm's own limit, that recurring wall is the signal to price out a paradigm change.

The speed is not the danger. The speed *exposes* the danger. A bad decision that would have cost you a slow week of hand-coding now costs you an afternoon and ten thousand lines, all of them committed, all of them plausible. Velocity is a floodlight on the quality of your thinking. This is why the frame comes first and why so much of the work is front-loaded. Your work now happens before the agent types: deciding what "correct" means, and how the code will be held to it.

4.5.3 The new economics

Refactoring used to be too costly; now it's free

For my whole career, refactoring was something you rationed. It was expensive, so you lived with a compromised design longer than you should have, and the compromise compounded. That economics has inverted. The refactors I had always deferred as too expensive became an afternoon's work. For the data, see The Road to MAGE → In one leap I unified the remediation interfaces across the document formats, which let me unify the remediation passes, which let me track what each change does — for corruption checks — and what it costs, for pricing. None of that was justifiable when refactoring meant days of careful hand-surgery.

So there is almost no reason to enshrine a bad shape. If the design is wrong, change it. The cost that used to make you tolerate accidental complexity is gone, and the discipline that replaces it is the willingness to keep paying the now-cheap price. Audit, find the compromise, fix it, and move on. The prototype is disposable; the learning it produced is what you keep.

The cost estimator is broken, and scope creep is guaranteed

Two consequences follow from cheap implementation, and both caught me off guard.

The first is that my sense of what things cost stopped working. Deferred refactors became cheap; migrations that sounded like weeks took hours. Agentic engineering makes an engineer's output a function of toolchain, environment, taste, context, and orchestration skill, so the range explodes. "What can one engineer ship this quarter?" no longer has a stable answer, and capacity planning that assumes bounded human variation starts to lie.

The second is that when implementation feels free, scope creep is a guarantee. More features, more integrations, more scope — the feeling changes your behavior before you notice. But cheap implementation does not just raise output. Every new thread still has to be decomposed, reviewed, and kept coherent, and the scarce resource stops being code and becomes judgment. Choosing what *not* to build is now as much of the work as building. Call it taste.

4.5.4 The failure modes

Optionality is poison

AI reduces friction, of course; but what it chiefly manufactures is optionality — an endless stream of choices, some architecturally critical, some product-relevant, and some stupid things the agent should have sorted for itself. At midnight, that stream is poison. Some decisions take you to hell, others to a parking lot where you spin your wheels, and others merely distract you from sleep, and in the moment they are hard to tell apart.

The deeper trap is the decision you leave open. When you do, the agent makes it for you — and it will pick whichever option unblocks it now rather than the one you would have chosen with the consequences in view. My merge-train cron kept stalling when two agents edited the same file, and it proposed a one-flag fix: tell the merge to just take one side and drop the other on any conflict. I signed off without asking what "drop the other side" meant. For four days it silently deleted real work from the main branch every time two changes overlapped — the same commit heading landing again and again, each time a little smaller — while I chased the symptoms one missing function at a time. The right answer was a merge resolver that understood the files it was merging, and I could have decided that up front. Instead I left the choice open and the agent filled it with the cheapest thing that made the error go away. It is the same reflex as the strings-and-ints story from Part 2 and Part 5: every unmade decision is a place the agent guesses, and its guesses optimize for the next green checkmark, with your system nowhere in the objective. So decide. Name the architecture, name the types, name what happens on a conflict. The narrower you make the agent's search, the more of its speed lands on the target instead of scattering. Explicitness is the mechanism by which you draw consistency out of a system that will otherwise improvise.

"Done" is a claim, not a fact

A failure mode appeared again and again: Claude reported a task complete, and my review agents later found the half-done implementations, the missing integration points, and the tests that never actually exercised the behavior. Current coding agents are superb at plausible local progress and weak at guaranteeing global completion. The confidence and the description report the agent's *intent* rather than the state of your repository — and the two drift apart constantly. 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 — but a botched merge

had landed them onto a tree that was missing the very field they were supposed to check, so they confirmed a field that existed and never touched the one that mattered. Both said “done.” Neither was.

So trust nothing at the boundary between the agent’s report and reality. Re-run the tests yourself. Re-run the lints the work was supposed to satisfy. Compare the counts against the claims. When the agent says “done,” your verification has only begun — and the discipline that catches the gap is mechanical: a gate the agent cannot talk its way past. My job shifted from writing implementation to running the control system around it — decomposing work, cross-checking invariants, hardening tests, and deciding whether the architecture was genuinely cleaner or merely different.

Your tests must survive agent-shaped failure modes

An agent fails differently than a person does. It does not forget a semicolon; it confidently produces a whole subsystem that looks right and is subtly wrong across every file at once. Worse, faced with a failing test, an agent will do the locally-expedient thing a human would be ashamed to: delete the tricky behavior, special-case the path that makes the input work, weaken a validation, bypass a pipeline stage — satisfy the visible symptom while corrupting the invariant. At least it apologizes when you confront it.

A test suite tuned to human mistakes will wave all of that through. So the tests have to be built for the failure modes the agent actually has. Mine stopped being conventional regression checks and became sensor barriers: asserting invariants, negative cases, architectural boundaries, end-to-end semantics. Pin the invariant, not the example. Reach for the strategy that catches the *class* — a property test where the shape is what matters, a state-machine check where a lifecycle can tear, a dynamics-aimed test where repetition or concurrency or stale state is the real risk. The goal is to block the agent from reaching the degenerate paths through an ambiguous specification.

Of those strategies, the property-based one is worth singling out, because it is the same move the whole book turns on, worn as a test — a compact, checkable model of what a correct output looks like, which a generator re-hunts on every commit. It turns out not to stand alone but to head a whole family of generative validation — property testing, fuzzing, and their synthesis with the structured model — pulled into its own chapter, Validation with Agents, where the three sit together as one move against three specifications.

Vibe coding treats the program as an I/O device; engineering treats it as a system

When you vibe-code, you ask the agent to produce something that appears to satisfy the requested behavior, and the evidence you use is all external: does the demo run, does the output look right? That works until it doesn’t — until the thing you cannot see is the thing that matters. Engineering asks the agent to preserve and improve a system’s essential properties while changing behavior, and the bar for evidence includes the internal structure: does it respect the architecture, hold the invariants, keep state in the right place, stay testable, avoid making the next change harder?

The difference lies in what counts as proof that a change is good. I did not vibe-code a million-line repo; I could not have. Agents make both modes cheap, which is exactly why you must choose deliberately — the fast, seductive default is the I/O device, and the I/O device is what turns a hundred thousand lines into a mess nobody can reason over.

Vibe coding often produces what in Hindi is called *jugaad*: an improvised, resourceful solution assembled from whatever components are immediately available. Large language models naturally reach for jugaad-type strategies, opportunistically combining APIs, libraries, design patterns, and generated code into systems that work surprisingly well. The ingenuity is real, and it is valuable for

exploration. But it rarely yields the architectural coherence, the explicit invariants, or the long-term maintainability that production software demands. MAGE begins where jugaad ends: it converts the successful improvisation into a durable engineering mechanism.

> *Jugaad* (जुगाड़) is a Hindi term for an ingenious improvised solution made from whatever materials, tools, or opportunities are available — widely used in India for frugal innovation and problem-solving under constraints. English has no exact equivalent; the closest popular analogy is "MacGyvering" a solution, though jugaad is broader and often carries a positive connotation of frugal ingenuity.

Explicitness is essential

A normal codebase keeps most of its conceptual structure in human memory. Humans remember what the components mean, which files are canonical, which patterns are deprecated, and they make excellent progress on that tacit knowledge. Agents can't. They start cold, carry finite context, can't sketch on a whiteboard with the senior engineer, and drift toward plausible-but-wrong edit sites.

The more the system is represented in explicit, queryable artifacts — component indexes, pass registries, standards manifests, lints, epics, handoff notes — the more agent-legible it becomes and the better the agents perform. One of my key moves was to add a soupçon of model-based software engineering: I made the system architecture explicit and executable, so it was visible to the agents working on it. What lives only in your head is invisible to the machine, and the machine is now most of your workforce.

4.5.5 The MAGE discipline

Governance is a spectrum from soft to hard

The mechanisms I ended up with were not all the same kind of thing, and the axis that sorted them was soft versus hard. A **soft** mechanism aims the agent — a documented rule, a brief, a convention. It shapes behavior probabilistically and cannot hold the line if the agent ignores it. A **hard** mechanism holds regardless of cooperation — a lint that fails the commit, a type the compiler rejects, a validator that refuses to ship. Guidance *aims* machinery *holds*. A mature environment uses both, and the skill is knowing which a given failure warrants. Not everything earns a wall. But when you find yourself saying "this must *never* happen," you have just specified a hard mechanism, and you should build it — a convention that says never is a convention that eventually gets violated at velocity.

The move that makes this compound is turning audits into lints. The first time a failure class shows up, you inspect for it by hand. The signal that you have a mechanism owed is the *second* time: a failure that recurs has earned its mechanism, and once you build it, the same failure is caught on every future agent, automatically, forever after.

AI is an autonomy amplifier

The thing agents amplify is your autonomy — your capacity to turn intent into a working system without waiting on anyone. I have spent years in industry and never felt more like a real engineer than on this solo project, and I think that feeling came from autonomy. I had end-to-end ownership: the problem, the regulatory environment, the user value, the architecture, the correctness conditions, the deployment, the optimizations and their direct relationship to pricing, and the consequences.

Agents lowered the implementation burden enough that I could inhabit roles usually spread across a team — developer, architect, DevOps, UX, pricing strategist, support, security analyst.

That amplification is neutral. It magnifies good judgment and bad judgment with equal enthusiasm, which is why the scarce resource has moved. Implementation used to be the bottleneck; now implementation is cheap and *judgment* is the thing in short supply. What is worth building, what "correct" means here, which failures deserve a wall and which deserve a note — those calls are the work now, and they are the calls an agent cannot make for you. So the discipline that pays is the one that converts your judgment into infrastructure: every decision you make well, you encode once, so you never have to make it again and neither does any agent after you.

What this means for the job

Watching my own experience of software engineering get compressed, I expected to feel the role shrinking. The opposite happened. The *experience* is being compressed — the hours at the keyboard, the mechanical translation of a design into syntax. The *role* is expanding. When one person can stand up hundreds of thousands of lines of governed production code, the constraint has moved from how fast you can write to how well you can decide, model, constrain, and verify.

Who holds the authority to convert a lesson

This is where I think companies may have trouble. I had authority to change everything — the architecture, the CI, the documentation, the task format, the release process, the validation hierarchy. In a real organization, the person who sees the recurring failure may only be allowed to patch their local problem. That is a recipe for churn: the same class of problem appears again and again, and no one has the authority to convert it into shared engineering practice. So to software organizations I offer one piece of advice, and it says nothing about whether to use agents: look at what your failures are teaching you, and ask who in your engineering organization is empowered to turn those lessons into architecture, process, and mechanisms. Put those people in charge of your agentic adoption.

The first engineers were locomotive engineers. They stood, literally, astride the harnessed steam engine — monitored it, tuned it, designed it. Software engineering has its DevOps and its SREs in the same lineage, and I suspect much of what I observed connects to their wisdom. The future of software engineering is the ability to convert judgment into infrastructure.

I will not pretend this is settled. My evidence is a single-person project without real customers, run against one vendor's models — Claude, Opus and Sonnet, over the March–July 2026 window — and the specifics will not all generalize. A legacy codebase full of ancient spaghetti, or a system with many stakeholders, may behave differently; if you are starting from spaghetti, the honest first move is to fix the spaghetti, because agents are excellent at refactoring once you have the test sensors to keep them from regressing. But the shape of it feels durable: convert judgment into infrastructure, let the machinery hold what your attention cannot, and spend your scarce judgment on the decisions the machinery cannot make. If we want to learn anything new, we have to try crazy things and ponder them. That is the method, and it is the most fun I have had building software in years.

4.6 Validation with Agents

Most tests reproduce a dot. You pick an input, you write down the output you expect, and the test asserts the two match. That is honest work, of course, and it catches a great deal; but it says nothing about the input sitting right next to the one you chose. There is a different move, and once you see it as a single move it turns out to name three techniques at once: **generate inputs and try to falsify a specification the model supplies.** Property-based testing, fuzzing, and the synthesis of fuzzing with model-based engineering are the same move three times. They differ on what the specification *is*, on what it means to *fail* it, and on how the inputs get *made* — and drawing those three axes sharply is what keeps the family from collapsing into a vague "testing with randomness."

That move is the whole book worn as a test. The specification the generator hunts is a model — the Modeling Thesis, that the system's correctness lives in an explicit model and not in prose. And the generator re-hunts a counterexample on every change — the Alignment Thesis, that the environment keeps re-checking the agent's work against that model automatically, forever, rather than trusting a one-time review. A generative-validation test is those two theses standing in one place: a compact model, and a machine that never stops trying to break it.

4.6.1 One move, three specifications

Table 4.6-1 lays the family out on one page. I suggest reading each row left to right — oracle, then failure, then generator — because the oracle is what the technique *is* and the generator is only how it gets there.

Technique	What the oracle is	What counts as a *failure*	How inputs are generated
Property-based testing	A **declared invariant** the author writes — a law the output must obey for every input: parse-then-serialize returns the original; a remediated document contains no less content than the original; a sort yields an ordered permutation.	The stated law is **false** on some generated input.	A **structured generator** synthesizes well-typed domain values; on a failure it **shrinks** the input to the smallest one that still breaks the law.
Fuzzing	An **implicit total-function contract** — "never crash, never corrupt," and "reject cleanly whatever the format forbids." The oracle is coarse but real: the process sur-	A **crash, hang, resource blow-up, or silent corruption** on adversarial or spec-edge input, even where no explicit law was stated.	**Malformed or adversarial bytes** — truncations, bit-flips, corrupted offsets, garbage appended — reaching inputs no well-typed generator would ever emit.

Technique	What the oracle is	What counts as a *failure*	How inputs are generated
	vives, and the output re-parses.		
Fuzz + model-based engineering	The **structured model itself** — the same model the fleet reasons through — becomes the oracle. It names the *stable point in the specification*: the closed set of legal outcome classes, the invariant predicate, the state-transition table.	The generated input drives the model into a state it declares **illegal** — an outcome outside the legal set, a transition the table forbids, an invariant predicate that goes false.	The same adversarial generation as fuzzing, aimed at a **model-declared surface**: the model's own read entry point, a state machine's driving conditions, a flow between services.

Three sharp distinctions fall out of that table, and stating them keeps property testing and fuzzing from blurring together.

First, **oracle richness and input wildness trade off, and the third member refuses the trade.** The richer the oracle, the tamer the inputs: a property test knows exactly what correct means, and yet it asks the question only of inputs a type would allow. The wilder the inputs, the coarser the oracle: fuzzing reaches bytes nothing else reaches, and yet it can notice only a crash. Fuzz-with-modeling is the synthesis that takes both halves: wild inputs, judged against a rich model-declared oracle. That synthesis is this book's thesis in miniature: the model supplies the oracle, and the search never stops hunting a counterexample to it.

Second, **the word *failure* means different things, and conflating them is the classic mistake.** For a property test, a failure is a stated law turning out false. For a fuzzer, a failure is the program leaving its safe envelope, crashing or corrupting, in a place where *no* law was ever stated. Blur the two and you get the two symmetric errors: a fuzzer that only checks for a crash when it should have been asserting an invariant, and a property test fed only tame inputs when the real risk was the malformed byte at the boundary.

Third, **the specification's stable point is the bridge between fuzzing and the model.** When a fuzzer finds a crash, there are two ways to fix it. You can patch the one input that broke — and leave every neighbouring input that would break the same way. Or you can trace the failure back to the point in the format's specification that the input violated, and fix *that* — so the fix covers every input the specification allows, not the single seed that happened to trip it. Fixing to the stable specification point is what promotes a raw fuzzer into a model-based one: the model *is* the stable point, written down.

4.6.2 The move: a generator hunts a counterexample to a model

Before the three members, the shared shape. An example-based test pins a single point: given *this* input, expect *that* output. A generative test does something else — it states the property the output must satisfy across the whole space of inputs, and hands a generator the job of hunting a counterexample. Where an example gives the machine one labelled dot to reproduce, a generative test gives it the law the whole space obeys and a search for the place the code breaks it.

That difference is exactly the one that matters against an agent. Faced with a failing example, an agent will do the locally-expedient thing: special-case the path that makes *that* input pass. The example goes green and the bug survives, now hidden behind a branch cut to fit the test. A property has no case to special-case. An agent that special-cases the one input sails through the example and straight into the law that the whole space of neighbours still has to obey. This is why the strategy that catches the *class* beats the one that catches the example, and why the generative family is the sharpest tool the lessons chapter reaches for when it says to pin the invariant, not the example. Figure 4.6-1 draws the difference.

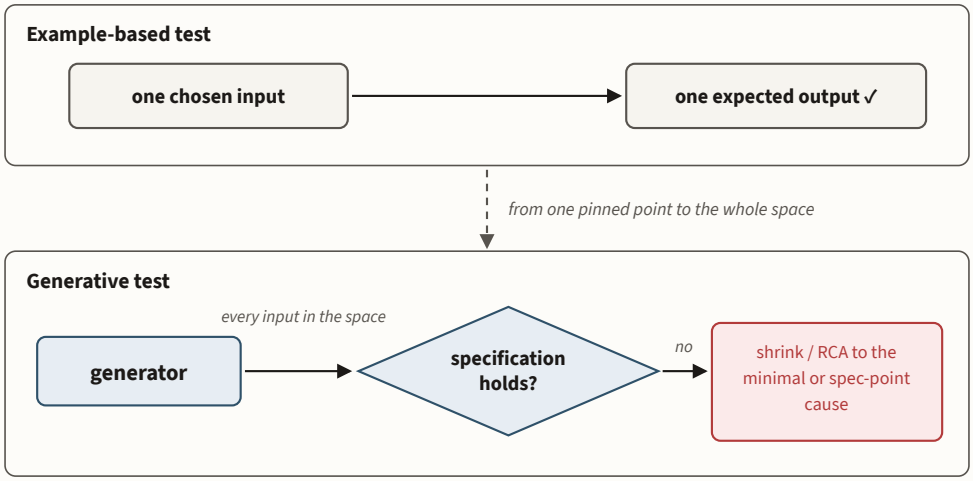


Figure 4.6-1: A Point vs a Space. *The example pins a point; the generative test models the space. One reproduces a dot the author already knew; the other states the law every dot obeys and lets a generator find where the code breaks it.*

4.6.3 Property-based testing: the invariant is the oracle

In property-based testing the author writes the law and the machine hunts the counterexample. The law is a statement about *every* output, not a single one: parse-then-serialize returns the byte-identical original; a remediated DocAble document contains no less content than the one that went in; a sort produces a permutation of its input in non-decreasing order. Each of these is a compact, checkable model of what a correct output looks like — a law, not a list. The generator synthesizes well-typed domain values across the space the law is supposed to hold over, runs the code, and checks the law. When the law breaks, the framework **shrinks**: it walks the failing input down to the smallest one that still breaks it, so the counterexample you get is minimal and legible rather than a thousand-node document with one bad byte somewhere inside.

The property is the Modeling Thesis worn as a test: a small statement the agent reasons over instead of enumerating cases it cannot hold in context. The generator is the Alignment Thesis, re-hunting a counterexample on every commit. In a real system this is no footnote technique. A property-based framework can run across every test project, and the discipline is to reach for it the moment a class's contract is expressible as a law rather than a list of examples. The counterexample it hands back is often a genuine defect the author never thought to write an example for. That is why you ask the machine to search the space instead of guessing at its corners.

Learn more about this governance mechanism: property-based tests.

4.6.4 Fuzzing: the contract is "never crash, never corrupt"

Property testing asks tame inputs a rich question. Fuzzing asks a coarse question of wild ones. The oracle is the implicit contract every input-parsing surface signs whether or not anyone wrote it down: *do not crash, do not hang, do not corrupt, and reject cleanly whatever the format forbids.* A malformed DocAble upload — a PDF truncated mid-object, a slide deck with a corrupted offset table, a document with megabytes of garbage appended — is precisely the input a generator of *valid* documents will never produce, and precisely where the crash lives. The fuzzer mutates and adversarially perturbs its way into that space and watches for the process to fall over or the output to fail to re-parse.

The multiplier is discipline about the fix. When a fuzzer surfaces a crash, the temptation is to handle the exact bytes it found. The stronger move is to trace the failure to the point in the format's specification that the input violated, and fix *that* — because the specification is stable while any one producer's quirks are not, so a fix aimed at the specification point closes every input the format allows, not the single failing seed. That is the discipline that turns a pile of one-off crash patches into a class of inputs handled once.

A campaign that merely runs has explored nothing, so the honesty layer matters: the host test-runner can auto-collect line and branch coverage over each fuzz campaign and track it against a baseline, so "we fuzzed it" becomes a claim with a number behind it rather than a vibe. That number is a saturation signal: the campaign has stopped finding new edges. Its deeper treatment belongs to the metrics chapter, which owns coverage.

Learn more about this governance mechanism: fuzz campaigns.

4.6.5 Fuzz + MBSE: the structured model becomes the oracle

This is the synthesis. Take the wild inputs of fuzzing and, instead of asking only "did it crash," judge the outcome against the structured model the fleet already reasons through. The model names the *stable point in the specification*: a closed set of legal outcome classes, an invariant predicate, a transition table. Point the malformed bytes at the model's own entry point, and classify what comes back against that declared set. A clean rejection of an illegal input is a *pass* — the model correctly refused it. An outcome outside the legal set — an unexpected exception, a silently corrupted structure, a state the transition table forbids — is a *fail*. The oracle is now as rich as a property test's, and the inputs are as wild as a fuzzer's. The trade-off the first two members lived under is gone.

Two instances make this concrete.

The first is the **format model**. Feed adversarial bytes to the structured document model's read entry point and sort the result into a small, closed set of legal outcome classes — parsed cleanly, rejected cleanly with a typed error, and so on. The set *is* the specification. An outcome the set does not name is the failure, and because the set is the model's own declaration rather than a hand-written assertion per seed, the same oracle judges every input the fuzzer can throw.

The second is the sharpest instance in the whole family, and it turns the fuzzer's usual approach inside out. Call it the **producer-dialect corpus.** Instead of *mutating toward* malformed bytes, round-trip a real document through a genuine third-party producer — a different office suite, a different PDF writer, a different export path — and let that producer's *legal-but-unusual dialect* be the adversarial input. Every producer emits its own accent within the format's grammar: unusual-but-valid object orderings, obscure-but-permitted structures, features the specification allows and your own writer never uses. Where the specification-point fix reasons *inward* from a failing seed to the stable rule it violated, the producer-dialect corpus reasons *outward* — it generates inputs that occupy the whole specification-allowed producer space, so the model is tested against the full breadth of what the format legally permits rather than the narrow slice your own tooling happens to emit. It is the family's most distinctive move, and it needs no mutation engine at all: the world's producers are the generator.

> **Harvest producers before you hand-roll a grammar.** *A grammar you write yourself covers the dialects* you *thought to encode. The format's independent producers — every rival tool that writes it — have already generated the legal-but-unusual space for you, a breadth no single authored grammar reaches. Reach for a from-scratch generator only where no real producer exists.*

The synthesis reaches one level up when the specification is not a format but a **concurrency invariant.** There the "input" is an interleaving of concurrent steps, the generator is an interleaving-fuzzer or an exhaustive search over reachable states, and the oracle is the invariant predicate evaluated over the model's states — no two workers hold the same lease; a job never leaves a terminal state; a queued item is eventually served. The class of the invariant even picks the checker: a straightforward linear invariant earns a property test, while a hairy invariant over concurrent interleavings earns an exhaustive state search that walks every reachable combination. The payoff, learned the hard way, is that *naming the invariant predicts where the bug is.* Writing the predicate down forces you to state the exact condition that must hold, and the search then drives straight at the interleaving that violates it — a defect a strong-but-static unit suite walks right past. This is generative validation standing on the same invariants-over-models ground the rest of the book is built on.

4.6.6 Coverage: did we explore the spec-relevant space?

Once you have a generator, a new question appears that example tests never had to ask: *did the campaign explore the space that matters, or did it just spin?* The book already owns coverage — the metrics chapter develops it as a flagship, with the rule to measure one level deeper than the raw percentage — so this section adds only the one cut that is genuinely native to generation and hands the rest back.

The native cut is **grammar coverage.** Line and branch coverage measure the *code* the inputs reached. Grammar coverage measures the *inputs* themselves: did the generator exercise every production of the input grammar — every structural variant, every mutation kind, every dialect the corpus is supposed to contain? It is branch coverage moved to the input side, and it answers the question the gap

analyses of a real fuzzing effort keep circling back to: *is the corpus rich enough?* A campaign that never generates a document with headers and footers, or never a cross-sheet formula, has a grammar-coverage hole no amount of line coverage will reveal, because the untested lines were never reached by any generated input in the first place.

For everything else, defer to the metrics chapter and close the loop. Line and branch coverage of the campaign is the saturation signal, and it is the honesty layer that keeps "we fuzzed it" from being an empty claim. But the coverage that ultimately matters for this family is **model claims** — did the generated inputs actually drive the invariants, transitions, and edges the model declares, measured as the metrics chapter's requirements-based coverage over the traceability graph? That is the connective tissue: the same "measure one level deeper" move the metrics chapter teaches, reused here as the saturation oracle for generative validation. Figure 4.6-2 draws the loop, closing on that coverage.

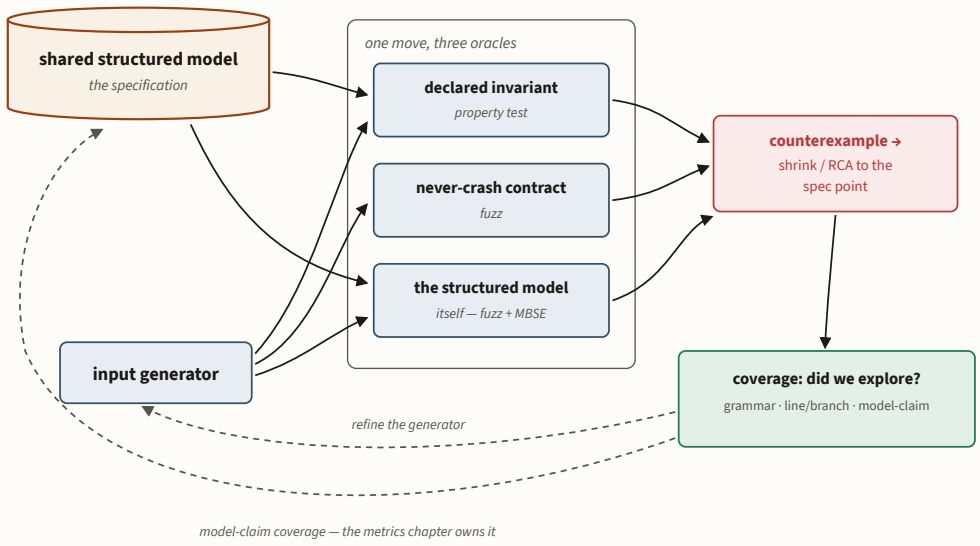


Figure 4.6-2: The Coin Is the Model. *Fuzzing and property testing are two sides of one coin: both draw their oracle from the same shared specification, and coverage asks whether the generator ever reached the claims the model makes.*

4.6.7 When to reach for which

The family gives a compact decision aid, and the axes of the opening table are the decision.

- A **stated law** over structured inputs — a round-trip, a content-preservation guarantee, an ordering — reach for a **property test.** The oracle is the law you can already write down.
- An **untrusted-bytes boundary** with a never-crash contract and no law worth stating per input — reach for a **fuzzer**, and fix its findings to the stable point in the specification, not the failing seed.
- A **structured model with a declared legal-outcome set or invariant** — reach for **fuzz plus the model**: point wild inputs at the model's surface and judge the outcome against its declared classes. When a real third-party producer exists, let its dialect be the corpus.
- A **concurrency invariant** — reach for the **state-model variant**: a property test for a linear invariant, an exhaustive search for a hairy one over interleavings.

All four are **hard** governance in the soft-versus-hard sense the lessons chapter draws: deterministic sensors, machine-read, re-run on every commit. They do not aim the agent and hope; they hold the line. A generator that re-hunts a counterexample every commit is the Alignment Thesis made mechanical — the environment refusing to trust the last green checkmark, and checking the model again, forever.

Step back from the fuzzer to the Part it ends. This one put the method to work end to end — the brownfield recipe, the skills that carry it, the transformations that move a legacy tree onto models, and the generative checks that hold the result. None of it came from a whiteboard; every move was forced by one real system under a real deadline. The next Part tells that system's story from the beginning.

Part 5

A MAGE Case Study

5.1 The ADA Context

> ***This chapter illustrates***
> ✓ The Alignment Thesis

Every method in this book was learned building one real system, under a real deadline, for a real problem. The problem is document accessibility. The engineering earned its keep only by clearing a bar the problem set, so the problem comes first — before agents, before loops. Why is making a document accessible a hard, mission-driven problem, and why was it worth a professor's Spring Break to attack it?

The system is called DocAble, live in beta at scholaccess.com. It takes a PowerPoint deck, a Word document, a spreadsheet, or a PDF and remediates it toward the accessibility standards a university now has to meet. Why that is worth doing at all is the rest of the story.

5.1.1 A noble idea with a crushing implementation

The Americans with Disabilities Act starts from a simple promise: people with disabilities should have equal access to public life, able to work, travel, study, vote, and use services without barriers society could reasonably remove. Passed in 1990 and signed by George H. W. Bush, the ADA descends from a long American line — the Constitution, the Reconstruction Amendments, the Civil Rights Act of 1964, each one widening the circle of who counts as fully included. The wheelchair ramp and the elevator are, of course, its famous results, and they are welcome. But the software problem grows out of a quieter corner of the statute.

That corner is Title II, which covers state and local governments — including K–12 schools and public universities. Title II says a public entity must communicate with people who have disabilities

as effectively as it communicates with everyone else. In 1990 that sentence mostly meant buildings and lectures and office hours. In 2026 it means something else, because a public university no longer communicates only in person. It communicates through websites, PDFs, slide decks, Word documents, spreadsheets, learning-management systems, videos, forms, exams, syllabi, and the thousand little files that pile up wherever people are allowed to upload things.

If those materials cannot be read by a screen reader, cannot be navigated by keyboard, lack captions, lack alternative text, or carry their meaning only through color or layout, then some students get a worse version of the university than others. The hazard is live, and it is legal as well as moral: accessibility lawsuits are filed every year by students who were not reasonably accommodated, and institutions have settled them for sums ranging from the thousands into the millions.

5.1.2 The rule that changed the burden

Universities have been obligated to comply with Title II since 1990. For most of that time, they complied reactively. A student would hit an inaccessible document, request an accommodation, and the institution would scramble to respond. That process carries a morally awkward shape: the inaccessible artifact exists first, and the burden of discovering it and asking for a fix falls on the person the barrier has already excluded.

In 2024 the Department of Justice changed the shape. A final rule updated the Title II regulations for web content and mobile apps, and its premise was blunt. Public entities should not wait for people with disabilities to discover and report every barrier. The DOJ put it in the language of physical access: just as stairs can exclude a wheelchair user from a government building, inaccessible web content and documents can exclude people from government services. 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*. Just as Americans do not call ahead to check whether a building has a ramp, a student should not have to negotiate with a professor over whether the slides will be readable. Table 5.1-1 lays the shift out side by side.

Table 5.1-1: The 2024 DOJ rule shifted the accessibility burden from reactive accommodation to proactive access.

Dimension	Reactive accommodation (the old posture)	Proactive access (the 2024 DOJ rule)
Who bears the burden	The excluded student — discover the barrier, then ask for a fix	The institution — build access in before anyone is turned away
When access is created	After a barrier is hit, one accommodation at a time	Before publication, as part of the service itself
What triggers the work	An accommodation request	Putting the document in front of students
The DOJ's own analogy	Calling ahead to ask whether a building has a ramp	The ramp is simply there
The standard	Ad hoc, case by case	WCAG 2.1 Level AA — with a compliance deadline
The scope	The single artifact a student flagged	Decades of accumulated digital sediment

The rule set a technical standard — WCAG 2.1 Level AA — and, for large institutions, a deadline. Universities are not sitting on three PDFs and a cheerful little homepage. They are sitting on decades of accumulated digital sediment: course sites, admissions forms, scanned readings, lecture videos, and more. A deadline is a deadline, and this one landed on a mountain of files no one had ever been asked to climb.

5.1.3 What it costs to make one document accessible

Compliance carries a cost, and the cost is easiest to feel on a single deck. Open a PowerPoint slide deck — say, the "Syllabus Day" deck for a graduate course — and click the *Accessibility* tab in Microsoft PowerPoint. On a typical real deck it reports dozens of distinct problems: images missing alt text, slides missing titles, low-contrast text, tables without headers, a reading order the tool cannot verify. One real deck examined this way flagged 42 separate issues across its slides.

Addressing one issue takes a couple of minutes at best, and the denser the figure, the longer the fix. Scale that to a course. A professor maintains on the order of ~100 assignable documents for the courses they teach — slide decks plus readings in Word and PDF. At a conservative 3 hours per document, remediating a course's materials by hand runs to roughly 300 hours — about 7.5 full work weeks. At a professor's salary that is over $20,000 for the materials of a single teaching load, and a research professor teaches more than one course. Multiply by the thousands of courses at one institution and the number stops being a budget line and starts being a reason the whole thing gets quietly deferred. This per-course estimate is the cost model the book returns to; other figures reconcile to it.

There are escape hatches, and each one closes on inspection:

- **Pull the plug.** A university could satisfy the letter of the rule by de-digitizing — removing the online materials so there is nothing inaccessible to fix. It would satisfy the letter and betray the spirit. Digital materials help everyone: captions serve the student in a noisy room and the student still learning English, not only the student who is deaf. No sane institution wants to become the one that took the slides away.
- **Hire it out.** External remediation vendors exist, and they quote on the order of $3 per page or slide — roughly $10,000 to $15,000 for one course's materials at the vendor's own rate. No vendor is staffed with people who can annotate graduate-level astrophysics or quantum mechanics, and no university is eager to pay that per course across every course it offers.
- **Do it yourself.** The professor has the specialized knowledge, and the professor is the one person whose time the institution values least for this task. At a research university, faculty are promoted on research; a large, uncompensated, recurring teaching-service chore is exactly the kind of work the incentive structure teaches them to avoid. Left standing, the burden does worse than go undone: it discourages updating course materials at all, freezing students into whatever version existed when the deadline hit.

Put these together and you have a genuine problem — no one had to manufacture it. A federal mandate with real consequences, a mountain of heterogeneous files, expertise that only the busiest people possess, and no affordable path from here to there.

5.1.4 Why a document is not just a file

Underneath the institutional bind sits a technical problem deeper than it looks, and it is why this became a software project worth building rather than a policy memo worth writing. A document is

not merely a file. It is a *visual and semantic* artifact. It carries text, figures, tables, diagrams, reading order, hierarchy, emphasis, and all 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.

For years that was out of reach. Earlier language models could not take an arbitrary slide and make enough sense of it to be trusted. The turn came with frontier vision-language models. Feed one a rendered slide and it looks at the image and recovers useful semantic content — describes the figure, reads the equation, names what the slide is trying to say. That is one of the core technical problems of document accessibility, and it is now solvable well enough to build on.

Raw model capability is not a compliance strategy, though, and this is the gap the engineering in this book had to fill. The model looks at a slide and understands something real. It understands nothing about university procurement, disability-resource workflows, faculty incentives, or the institutional alchemy that turns a federal mandate into a weekly email nobody reads. It is, as I like to put it, an infinite midwit: it knows everything and understands nothing. As the earlier chapters kept showing — a flash of capability is not a system; you get one only by bounding what you ask, typing what comes back, and validating before you believe. Turning that flash into a system a university can trust is engineering.

That is the context. A real law, a real deadline, a real cost, and a real technical gap that had just barely become bridgeable. The next chapter tells what happened when one professor decided to bridge it — and how a one-week undergraduate project became a production system built almost entirely by machines.

5.2 The Timeline and the Work

> ***This chapter illustrates***
> ✓ The Printer · ✓ The Governed Engineering Environment · ✓ Governance Conversion

The last chapter established the problem. This one tells what happened when I tried to solve it — the concrete arc from a five-minute experiment in the back of a committee meeting to a production system that, by the end, was writing almost all of its own code. The story matters because every governance idea in this book was learned at a specific point on this arc, in response to a specific failure. The methods are not a theory that came first. They are the sediment left by the work.

The scale is easy to state and hard to believe. Over about 20 weeks of focused effort, a fleet of AI coding agents and I produced a repository of 501,094 lines of production code, 1,505,737 lines of support apparatus, 35,323 lines of infrastructure-as-code, and 28,507 lines of system models bridging the two. On a routine day, six to eight agents ran in parallel and landed on the order of 200 commits — about 1,000 a week, sustained, against the hundred a week a fast unaided human might manage. I inspected almost none of the code. That one fact is the engineering problem this book exists to answer: how a fleet producing more code than any human can read stays trustworthy.

5.2.1 It started in a meeting

The project began, fittingly, in a committee meeting about the very deadline described in the last chapter. In early 2026 an accreditation committee was discussing the ADA compliance date. Most of the faculty in the room did not know a mandate existed; those who did had made little progress against it. While the meeting droned on, I did what an engineering professor does when bored by a conundrum: 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 looked at a rendered slide and recovered real semantic content from the image — the core technical problem of document accessibility, solved well enough to build on.

That single positive result was the decision point. It converted a research question into an engineering project. What it did *not* do defines the gap the rest of the timeline had to close: the model did not export a corrected file, did not add reading order, did not validate against any standard, and understood nothing about the institution the file lived in. A flash of capability is not a system.

The first plan was small and conventional. Draft a one-week project — automated alt-text for the equations in instructional materials — and hand it to an undergraduate or two over Spring Break. I posted the offer to twenty of the department's best students. Not one replied; they had, reasonably, planned to have fun. So I decided to supervise agents instead of undergraduates. I bought a subscription, opened an empty Git repository, and told myself one month should be plenty.

5.2.2 Six stages, then a seventh

What followed moved through a recognizable sequence of stages. Each stage was driven by a concrete pressure — a colleague's request, a user's complaint, a deadline, a failure — and each one enlarged both the product and the machinery that kept the product trustworthy.

- **Stage 1 — Feasibility probe.** The chat-model experiment from the meeting: screenshots of mathematics and figures, submitted to a vision-language model, to confirm that alt-text generation was viable at all.
- **Stage 2 — PowerPoint remediation.** The probe became a command-line tool. A walker traversed PowerPoint slides, queried a model for figures and equations, and wrote descriptions back in place — soon also fixing slide titles, contrast, and reading order, driving down the warning count in Microsoft's own accessibility checker. Colleagues started asking for other formats.
- **Stage 3 — Format expansion.** Word and Excel extended the same Office walker. PDF did not; it forced the first real architecture — a document-type classifier, a vision pass to recover untagged figures, and a two-pass "analyze, then remediate" model. Documents from many sources broke assumptions the clean local decks never had.
- **Stage 4 — SaaS.** A website, a containerized worker, and a cloud deployment came online within days, pulling authentication, metered billing, quota and cost controls, and an early security review in behind them. Real users on real documents exposed the limit of trust: a lower checker-warning count is not the same as having done the job. Some outputs passed the checkers and lost the meaning.
- **Stage 5 — Standards conformance.** To make a defensible accessibility claim, the project stopped trusting external oracles like Microsoft's checker and veraPDF and built its own standards-grounded check engine, mapping every finding to a specific WCAG 2.1 AA, Section 508, or PDF/UA clause. Owning the check let remediation and verification share one vocabulary and run after every pass.

Learn more about this governance mechanism: standards-grounded check engine.

- **Stage 6 — Hardening.** The DOJ extended the deadline by a year, and the project shifted from emergency response to production hardening: typed commits, isolated agent worktrees, validation and fidelity gates, provenance stamps, property and regression tests, audit agents, fuzzing, and deploy gates.
- **Stage 7 — Serverless.** On a Monday I told the agent I was spending too much money running a Kubernetes cluster that mostly sat idle, and we should go serverless. By that night the production deployment was serverless — about 400 commits of rewiring, structured into 27 phased designs, each of which surfaced a handful of judgment-call questions for me to answer before it ran. The multi-service system took on a function-as-a-service shape, so capacity could scale on demand rather than bill around the clock.

The two-line summary of the whole sequence is that the product grew, and the *apparatus that governs the product grew faster*. Measured in source size, the support machinery — tests, lints, load-bearing documentation, agent infrastructure, and tooling — ran about 3.0 times the size of the production code, and the test suite alone was roughly the size of the product. A ratio like that will, of course, read as gold-plating — until you recall that at 200 commits a day no human reads the diffs (the apparatus stands in — more on that in the built system). It is the price of the velocity.

5.2.3 The velocity curve

Figure 5.2-1 plots commits per week: a steep rise as the fleet gained capability, then the dip where velocity buys trustworthiness.

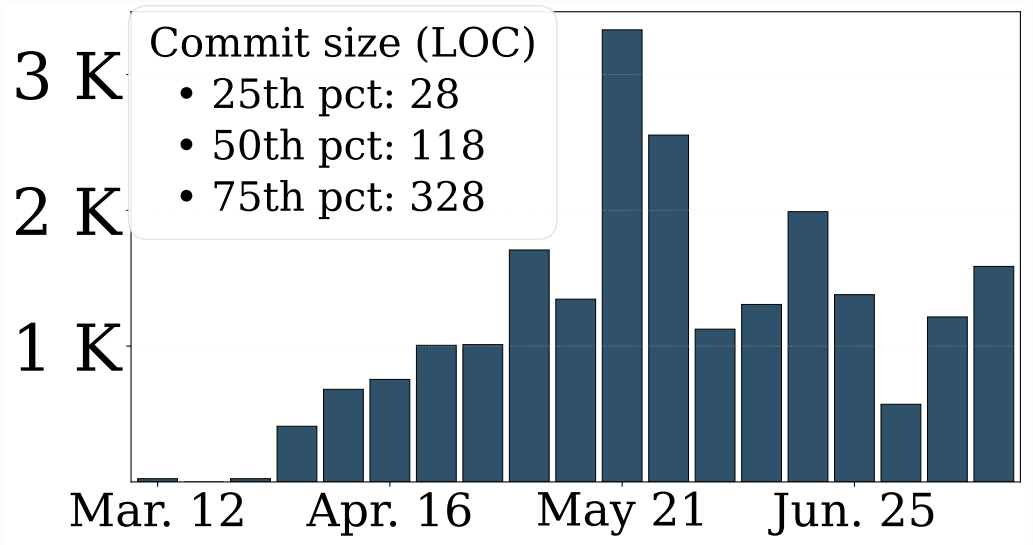


Figure 5.2-1: Commits per week across the project's history — the velocity curve. Each bar is one week; the curve rises as the fleet gains capability, then dips during hardening. That dip is where velocity is spent making the output trustworthy rather than merely fast.

The repository's own activity records the shape of the effort. Commits and lines-changed per week rose steeply through the MVP stages — feasibility, PowerPoint, formats, service — as the system acquired capability. Then, during hardening, the raw velocity fell. That dip is the signature of the work changing character: from producing features to producing the mechanisms that let features be produced safely. The curve is a picture of me turning from a coder into an architect.

The human cost is part of the record too. It was so fun to build that sleep dropped to four or five hours a night; exercise and most everything else fell away. To stay ahead of the subscription rate limits, I ran up to four top-tier plans in round-robin and scheduled my own body across them. The addiction was real. The lesson, though, is what the machinery had to become to make that pace produce something trustworthy instead of a mess.

> ***Field note — scheduling a body around the agents***
> The pace left marks on the days. I did development on the kitchen counter while the family slept. A graduate student once walked across a corporate-retreat dining room to complete a two-factor prompt for me. During a campus network outage I sprinted between two buildings so the agents would not stall waiting on me. The point of keeping these is not the drama; it is that a fleet running this hot needs machinery that holds the line when its operator plainly cannot.

The six stages read like a plan. They were not one. Each stage is a line drawn *afterward* through a run of dated, ordinary moments — a breakfast, a keynote walked out of, a hospital waiting room, a purchased fourth subscription at midnight on a holiday weekend. Set the clean account against the lived one and the stages stop reading as strategy. They are what they were: the shape you can only see once the mess is behind you. Figure 5.2-2 plots the real days behind the clean stages.

The model joins me on the countertop.
The vegetables fry and the code flies.

Spring Break · the kickoff

I offer a one-week project to twenty of our best students.
Not one replies. They planned to have fun.
So I supervise agents instead.

Wednesday, April 15

A keynote on the future of software in the age of AI.
The room's routers are overwhelmed. The model cannot hear me.
We should do this, not discuss it. We leave.

Thursday, April 30

I hit full context with a clean quiesce and a handoff document.
I post a screenshot in the lab chat.
My students cheer the performance.

Tuesday, May 5

Two accounts are saturated. The third needs a two-factor code.
Its owner is at a retreat, not watching the chat.
My graduate is at the next table. She walks over and taps him in.

Wednesday, May 13

I wake to ten emails. Each auto-funds the account five dollars.
Ah yes: I asked for a green build overnight.
The build costs real money.

Tuesday, May 19

The orchestration now makes progress for two hours in my absence.
I set the alarm for 4 AM. I was to be on vacation in Costa Rica.
This is better.

Thursday, May 21

A family member's seventh ER trip in two weeks.
The hospital network blocks one assistant, but not the other.
So the work continues from the waiting room.

Thursday, May 28

Figure 5.2-2: The Messy Timeline. *Nine dated moments from the 20-week build — the clean stages redrawn through the days they were actually made in.*

5.2.4 The support ratio: build the environment first

The apparatus outgrowing the product is not a one-time fact; it is a curve. Measure the support apparatus against the production code at four dated commits and the revealed preference of the whole build falls out in one line. Figure 5.2-3 plots it.

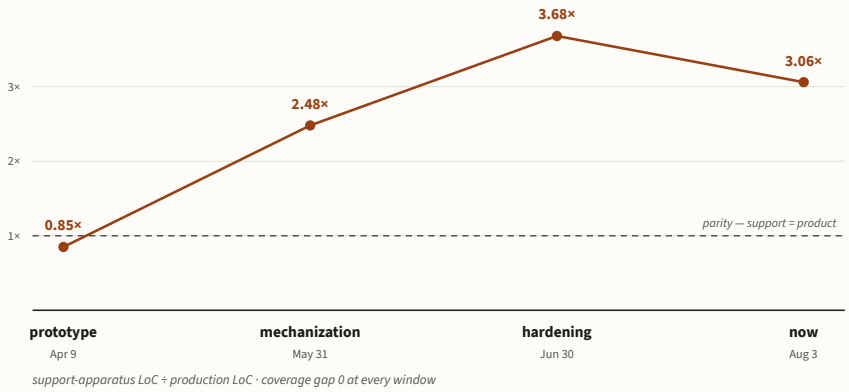


Figure 5.2-3: The Support Ratio. *Support-apparatus lines divided by production lines, across four windows: it starts below parity (0.85×), passes production at mechanization (2.48×), peaks during hardening (3.68×), then eases to 3.06×. The support line sits above parity for every mature window — governance led the fleet and kept pace.*

Table 5.2-1 gives the lines behind the ratio — production and support apparatus at each of the four commits.

Table 5.2-1: The support ratio across four windows — production and support-apparatus lines of code at four dated commits, and their ratio.

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×
now — Aug 3	491,090	1,501,907	3.06×

Read the curve as a decision, not an accident. In the prototype the support apparatus was smaller than the product — 0.85×, below parity — because there was barely a product to govern. By mechanization the apparatus had crossed production and reached 2.48×, and by hardening it peaked at 3.68×: roughly three and a half times the code it guarded. The governance was built *first*, then the features leaned on it. That ordering is the whole argument of this book, drawn as a line. The support ratio is the Governed Engineering Environment measured against the product it governs.

The final-window dip to 3.06× is not a retreat. Support did not fall — it grew another 21% in absolute terms. Production simply grew faster, up 45%, as feature work resumed on top of the finished environment. A falling ratio here signals a *stabilized* foundation being spent, not an abandoned one. Across the whole span the apparatus crossed from sub-parity to about three times the product while production itself grew roughly forty-seven-fold — the machinery led the fleet and never fell behind it.

One honesty note keeps the number from overclaiming. The census fails loud over seven primary source roots, so the coverage gap is zero at every window — no file the tool was asked to count slipped past it. About a fifth of the raw tree sits outside the census by design (root configuration, assets, non-primary test corpora, this book itself), and that side-tree share is roughly flat across the mature windows, so the curve tracks a real shift in how effort was spent, not an artifact of the category map moving under it. Lines of code remain a coarse proxy for effort; the ratio is a revealed preference across one repository and one fleet, not a controlled comparison of governed against ungoverned work.

5.2.5 The shape of the churn

Under the ratio sits the raw motion of the code: lines added and lines deleted, week over week, on the two paths that carry the product — `web/`, the Python service and worker, and `backend/`, the C# tool and rule engine. The preface named churn as the wall a fleet hits. Trace it per path across the four windows and the wall has a clear silhouette. Figure 5.2-4 draws it; Table 5.2-2 gives the counts.

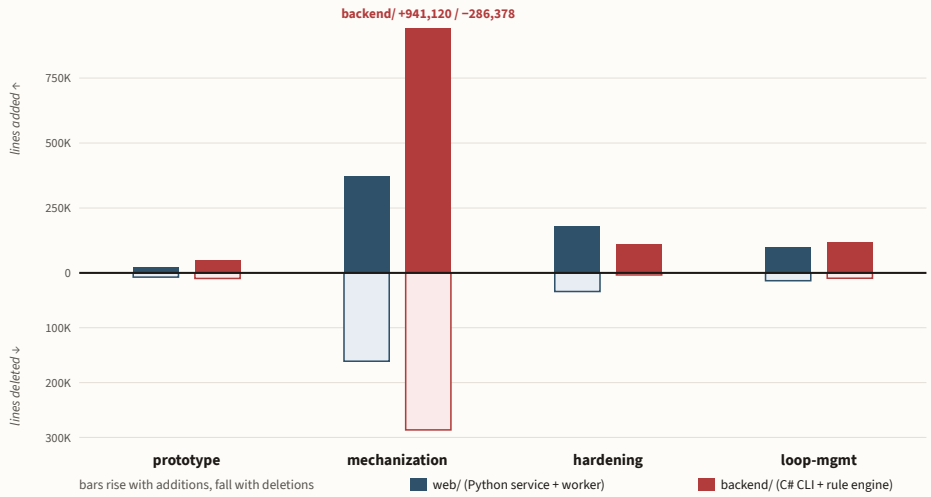


Figure 5.2-4: Path churn across four windows — additions rise above the baseline, deletions fall below it, one bar per path. Mechanization is the add-and-delete peak; after it deletions collapse (backend/ from 286,378 to 3,767) and later windows go net-additive as the environment stabilizes the code.

Table 5.2-2: Per-path churn across the four study windows — additions and deletions on web/ and backend/, from git numstat over the commit-date 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-mgmt	96,825	14,332	116,313	9,708

The peak is unmistakable. Mechanization — the wave where the agent fleet learned to build the environment — is where both paths churn hardest, the bulk of every addition and nearly every deletion. `backend/` alone rewrote itself in that window: 941,120 lines added against 286,378 deleted,

a system being torn down and rebuilt as fast as it was written. That motion is the danger the rest of the book governs. Ungoverned, it is exactly the churn the preface warned of — effort spent undoing and reconciling instead of advancing.

Then the deletions collapse. `backend/` falls from 286,378 deleted lines at mechanization to 3,767 at hardening, and both paths turn net-additive: far more added than removed, window after window. The code stops thrashing. That turn is the built environment paying off — once the apparatus holds the line, changes accrete onto a stable base instead of churning it. The shape of the deletion curve is the churn wall being pushed back.

One caveat keeps the numbers honest, and a measurement now bounds it. These counts come from git's own line accounting, which sees every tracked file, so generated bundles and vendored trees ride along with hand-authored source. Partitioning each changed line by path shows how much that inflates the totals. On `backend/` the answer is none: zero generated lines, so its additions are all real C# and configuration. On `web/` it is 14.6% of added lines, and almost all of that falls in the mechanization window (24.5% there), when the TypeScript-to-JavaScript build output and wire-contract codegen were still tracked; once the compiled output was git-ignored the generated share dropped to 1-2%. Across both paths combined the generated share is 5.2%, because `backend/` dominates the total and carries none of it. So read the bars as a churn *signal*, not a hand-authored source count — but the inflation is bounded and time-localized, concentrated in one window on one path, and the curve is predominantly real source everywhere else.

The deploy saga: the surprises the tests could not see

For a stretch of the build, nearly every unwelcome surprise came from deployment, and they shared a shape. The unit tests stayed green. The lints passed. The thing that broke was never a statement in a file; it was a *behavior of the running system* — a cold start that took a hundred times its warm latency, a gate budget sized for a regime that no longer held, two services contending for a core, a dependency graph that woke in the wrong order. The metrics chapter tells the cold-start half of this as a measurement story, and the numbers live there. The point here is different, and it is methodological: none of these lived in the statics. They lived in the dynamics, and no amount of green on the unit suite could have found them, because each coupling ran across layers the unit suite never spans at once — the code, the lints that guard it, the tests that exercise it, the infrastructure-as-code that places it, and the deploy pipeline that sequences it. A property that emerges only when those five layers run together is invisible to any check that reads one of them alone.

That is the case for testing the dynamics, not only the statics — and for the harder move underneath it. A one-off fix to a cold start or a gate budget clears today's surprise and leaves the next one to be discovered the same expensive way, on the next real deploy. The durable answer is to turn each discovered dynamic into a mechanism that watches it: a measured cold-start number fed to a model that recomputes the worst-case path, a budget derived rather than guessed, an ordering asserted rather than assumed. Do that and the class stops recurring silently. A finite review of a diff never had a chance against these — the coupling is not on the diff. Only a mechanism that watches the whole running dynamic holds the line as the fleet's deploys speed up, which is the mechanized-assurance claim read on the one surface where human attention is least able to help.

5.2.6 The kinds of work it required

Read this timeline as a catalog of the *kinds of work* agentic development turned out to need — a more useful reading than any feature list. The code was cheap. Everything around the code was where the judgment went, and the same handful of activities recurred at every stage:

- **Framing the problem.** Deciding what "accessible" even meant for a math-dense graduate deck, where the file alone underdetermines the answer, and which parts of the mandate the system should chase.
- **Discovering abstractions.** Recognizing that PDF was not "Office with a different extension" but a distinct problem needing a document model of its own — and building structured models so that structure was compiler-checked instead of passed as anonymous primitives.
- **Deciding local versus structural.** For each failure, judging whether it was a one-off bug to patch or the visible symptom of a missing mechanism — the single most repeated act in the whole record.
- **Building the checks.** Standards-grounded validation, content-preservation checks that assert the input survives into the output, hundreds of project-specific lints atop a maxed-out commodity floor, and a tiered regression suite from smoke tests to fuzz campaigns.

  Learn more about this governance mechanism: content-preservation checks.

- **Governing the fleet.** A dispatch-and-track workflow for parallel agents, isolated worktrees, a pre-commit gate whose passage is proven by a hash-keyed marker file, an agent registry that authoritatively answers "who is in flight," and lifecycle records for every agent that lands or dies.

  Learn more about this governance mechanism: pre-commit gate.

  Learn more about this governance mechanism: agent registry.

- **Operating and deploying.** Cloud deployment with staged gates, quota and cost controls, observability channels, audit trails that let a finished document's history be reconstructed, and runbooks for the failures that recur.

Read as a list, those are the chapters of this book. Every one of them answers a failure this timeline produced, and the recurring move that produced them all is the same: *velocity surfaces a failure; judgment classifies it as local or structural; the structural ones get converted into a durable mechanism that narrows the next agent's room to fail.* That move, applied over and over, is what the rest of this book laid out, told slowly enough that you can do it too.

By the end of the study period the system could process Office and PDF files to pass or substantially improve their accessibility scores — a graduate slide deck in about a minute, for about a dollar — and it was patent-pending and being readied for institutional deployment as DocAble. The direct development cost was about sixty thousand dollars, most of it my salary, with a few thousand each in model inference, cloud hosting, and subscriptions. That is the arc. How it was governed is the story the rest of this book has already told.

5.3 The Built System

> ***This chapter illustrates***
> ✓ The Modeling Thesis · ✓ The Alignment Thesis

The timeline told what happened. This chapter tells what got built, because none of the governance in the rest of the book makes sense until you can see the shape of the thing it governs. DocAble is a real system, live in beta: hand it a document a university actually distributes and it hands back a version a screen reader can read. Frontier models do the parts that require understanding a slide; ordinary deterministic code does everything else. The interesting engineering lives in how those two halves are joined.

5.3.1 What the system does, end to end

A user uploads a file. The system figures out what kind of document it is, breaks it into work, remediates each piece, checks the result against a real accessibility standard, stamps what it changed, and returns the corrected file with evidence that the job was done. Six moving parts carry that load.

- **A front door.** A website and an API take the upload, authenticate the user, meter the usage against a quota, and put the job on a queue. A professor drops in a slide deck; the system takes it from there.
- **A dispatcher.** The document is split into chunks — slides, pages, sheets — and each chunk becomes a unit of work handed to a worker. Splitting is what lets a hundred-slide deck finish in about a minute instead of an hour.
- **A remediation core.** A command-line engine, written in a typed language, does the actual repair. It walks the document through a structured model of its format, and for each thing that needs fixing it either applies a deterministic rule or asks a model. This is where alt text gets written, reading order gets set, titles get added, contrast gets fixed.
- **A check engine.** After every pass, a standards-grounded checker maps each remaining finding to a specific clause of the accessibility standard, so the system owns the definition of done rather than trusting an outside tool.

  Learn more about this governance mechanism: standards-grounded check engine.

- **A fidelity validator.** The fidelity validator introduced earlier sits here, catching the file that came out more accessible and less true.

  Learn more about this governance mechanism: fidelity validator.

- **A provenance layer.** The provenance layer from the pipeline stamps every insertion, giving a document the auditable trust a university requires.

  Learn more about this governance mechanism: per-mutator provenance stamps.

Those six parts are the product. What surrounds them is larger. In the talk this book grew from, I put the code-to-test ratio provocatively: the industry is happy with one line of tests per line of code,

and I had one-to-four. The real recorded figure across the whole tree is close to that in spirit — the support apparatus, tests and lints and models and load-bearing documentation together, runs about **3.0 times** the size of the production code. Call that inversion gold-plating if you like — but the more code the fleet produces, the smaller the fraction of it any human reads, and at this volume no one reads the diffs at all. The apparatus is the review.

Two design commitments run underneath all six. Every format is touched through **one structured model**, never the raw library, so a fix applied once holds everywhere. And the system **fails loud**: when the model is unreachable, the pipeline stops rather than quietly shipping a degraded file.

Learn more about this governance mechanism: one structured model per format.

5.3.2 The document is the hard part

None of this would be worth building if a document were merely text in a box, and of course it is not. A slide is a visual and semantic artifact — figures, tables, equations, an order the eye follows, emphasis carried by layout and color. To make it accessible, the system must recover enough of that meaning to present it through another channel. For years that was out of reach. The turn came with vision-language models that can look at a rendered slide and say what it means: read the equation, describe the figure, name what the slide is trying to teach.

That capability is real, and it is not a system. You can paste a small deck into a chat model and get useful output. You cannot get a corrected file back, a guarantee it satisfies the standard, or any evidence the meaning survived the trip. The model understands the slide and understands nothing about the file format, the standard, the university, or the cost of being wrong. Turning that flash of capability into something a university can trust is the engineering. The next section gives the pattern that does the turning.

5.3.3 Designing with a probabilistic component: the LLM as a function call

A model is a **probabilistic component**. Ask it the same question twice and you may get two answers. Ask it a question slightly outside what it saw in training and it will answer anyway, confidently, wrong. A deterministic function you can reason about; a probabilistic one you cannot. And yet the model is the only thing that can look at a figure and describe it. You have to build a reliable system out of an unreliable part.

The pattern that makes this work is to treat the model as a **typed function call**. Deterministic algorithms control the workflow and define the objective of each step. When a step needs semantic understanding, the system delegates a *bounded* task to a tool-equipped model — then a deterministic module validates the returned artifact before it is allowed in. The workflow is deterministic. The model is a subroutine inside it. The subroutine is never trusted on its word.

The same pattern, in another domain — and its lineage. *The shape is old and rigorous. Proof engineering separates the untrusted work of writing a proof from the trusted kernel that checks it: the kernel, not the author's confidence, decides what is admitted. Ringer and colleagues survey what it takes to make formally verified software scale: proof organization, automation, and maintenance[51]. Recent systems put a language model inside that loop, letting it search for*

proof structure while a prover returns counterexamples and admits only a closed proof[52]. Our own AutoSOUP is one: it verifies component-level memory safety by representing scope, loop bounds, and environment assumptions as explicit unit-proof artifacts, delegating bounded inference through an LLM-as-function-call architecture, and validating each choice against the verification objective[53].

MAGE carries the same trust boundary out of the proof assistant and into ordinary production. The external authority need not be a theorem prover. It can be a type system, a semantic lint, a preservation check, an architecture gate, a conformance engine, or a deploy rule. What must not blur is the guarantee each one carries: a lint proves less than a verifier, a preservation check proves only the property it encodes, and no green local check certifies the whole system. The untrusted generator paired with a trusted checker is not MAGE's invention; MAGE's move is to compose that one shape across the whole line.

Read the pattern in three moves, drawn in Figure 5.3-1.

- **The caller is deterministic, and it owns the workflow.** Ordinary code decides what happens and in what order. It decides *when* a step needs the model at all — most steps do not. When a step is a matter of judgment rather than a matter of rule, and only then, the caller reaches for the model.
- **The call has a typed contract on both sides.** The task handed to the model is packed into a typed input — this figure, this context, this question — and the answer must come back in a typed output shape the caller can act on. The return value is a structured, checkable artifact, never a free-form paragraph.
- **A deterministic validator sits at the trust boundary.** The model's answer is a *candidate*, not a result. Before it is incorporated, deterministic code checks it: does it satisfy the standard, does it preserve the input's meaning, does it fit the contract. On a pass, the caller incorporates the candidate and moves on; on a fail, it retries or falls back.

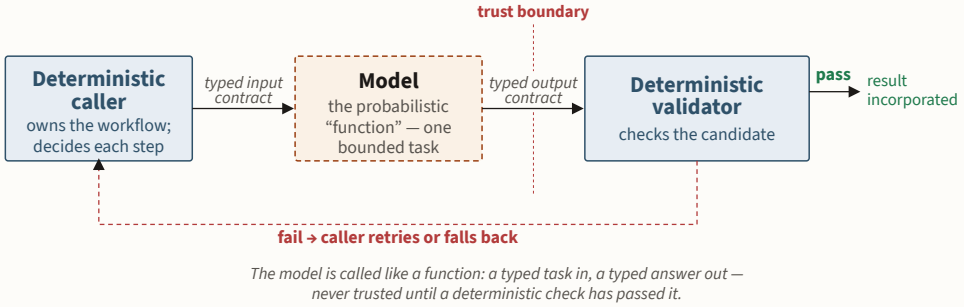


Figure 5.3-1: LLM as a Function Call. *A deterministic caller packs a task into a typed input; the model returns output under a typed contract; a validator passes it before use. Determinism on both ends makes the model safe to call: a bad generation costs a retry, never a corrupt result.*

[51] Talia Ringer et al., "QED at Large: A Survey of Engineering of Formally Verified Software," *Foundations and Trends in Programming Languages* 5, nos. 2–3 (2019): 102–281, https://arxiv.org/abs/2003.06458.

[52] Haoxin Tu et al., "Agentic Verification of Software Systems," 2025, https://arxiv.org/abs/2511.17330.

[53] DocAble Research Group, "Deterministic Workflows with Bounded Model Delegation for Software Verification," 2026, https://arxiv.org/abs/2605.10712.

This is why DocAble can make a defensible claim on top of an infinite midwit. The model supplies the one thing deterministic code cannot: it looks at the figure and understands something. Everything around that understanding — when to ask, what to ask, whether to believe the answer — stays in code you can reason about and test. The probabilistic part is sealed inside a bounded call with a check on the way out.

This discipline was earned the hard way. Skip the wrapper and the model will find the exception you did not guard — I assure you. Agents are the fastest road to a working demo and the fastest road to a subtle, confident, wrong result. The function-call shape keeps the first from becoming the second.

5.3.4 Three views of the same system

A model shows a view. The sections below draw three architectural views of DocAble from its real system models. Each answers a different question. The first asks how the services are shaped, and why that shape let the whole system change its deployment target without a rewrite. The second asks how the front end is decomposed, and how the editor changes a document through a small edit language. The third connects that edit language to the automated pipeline, and shows the join as a concrete instance of the function-call pattern above. Read together, they are the standing structure the rest of the book governs.

5.3.5 View 1 — a reactive service seam, and how it made serverless natural

The back end is a set of small services wired to react to events. A user's upload lands at a web front door, which splits the document into chunks and enqueues each chunk as a unit of work. A queue hands each chunk to a stateless worker. The worker remediates its chunk and publishes a completion. A fan-in step collects the completions and assembles the finished file. Every hop is an event handed forward, not a component polling for work. Two commitments hold the seam together: services stay reactive and stateless, and the split of duties is declared — Redis carries communication, the database carries truth. Figure 5.3-2 draws the seam and the two ways it deploys.

Learn more about this governance mechanism: service-flow model.

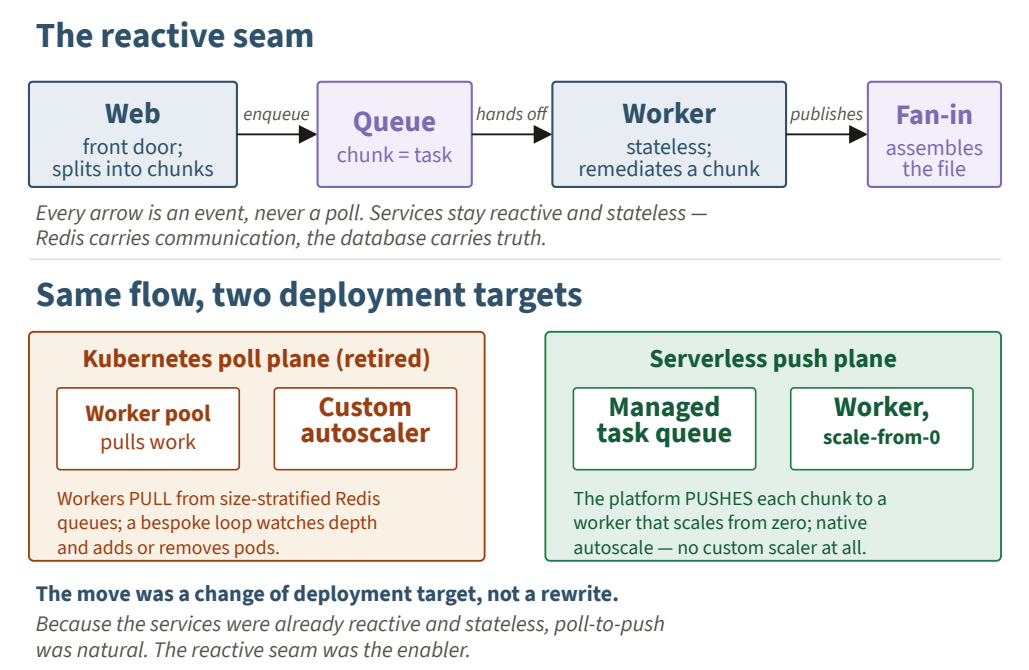


Figure 5.3-2: The Reactive Seam. *Top: an event-driven flow — web enqueues chunks, a queue hands each to a stateless worker, a fan-in assembles the result. Bottom: the same flow two ways — the retired Kubernetes poll plane and the serverless push plane, native autoscale, no custom scaler.*

That shape is what made a hard migration easy. For most of the system's life the workers ran on a Kubernetes cluster, and the dispatch worked by **poll**: workers reached into size-stratified Redis queues and claimed the next chunk, while a custom autoscaler watched queue depth and added or removed worker pods. The cluster billed around the clock, which for a bursty, low-traffic product was the wrong cost shape. The fix was to move to a serverless **push** plane: a managed task queue hands each chunk to a worker that scales from zero, a managed topic fans in the completions, and the platform's native autoscaler replaces the custom one.

In the industry a re-platforming like that is a quarters-long migration with a rewritten coordinator and a nervous cutover. Here it was close to a change of deployment target. The reason is the seam. A reactive, input-triggered handler *is* the serverless execution model (an event in, work out), so the workers did not need new logic to be pushed to instead of polling. A stateless worker can be spun up on demand and thrown away, because it carries nothing between invocations. And the split that kept coordination in Redis and truth in the database meant the durable state did not live in any worker that serverless would recycle. The poll-to-push inversion then *deleted* complexity rather than adding it: the custom autoscaler, the one-cluster-per-prefix rule, and the idle-scaling controller all fell away, replaced by the platform's native scale-to-zero.

None of the three hardening pushes that produced this shape — the reactive conversion, the move to statelessness, the modeling of the cross-service invariants — was aimed at serverless. Each was motivated on its own terms. Their convergence is the lesson: a system built reactive, stateless, and modeled is, almost by accident, a system that is safe to run serverless. The reactive seam was not a feature of the migration. It was its precondition.

Here is how fast the seam let the migration go — the operating loop the rest of the book describes. The timeline already tells the migration itself: a Monday teardown of an idle Kubernetes cluster, serverless by that night, about 400 commits structured into 27 phased designs. What matters here is the loop

those phases ran: the agent proposes a phased design, it surfaces the decisions that need a human — one to eight judgment calls a phase — I make those calls, and it executes. It was a re-platforming that would be a quarters-long project in industry, run over two days because the seam had already been built right.

5.3.6 View 2 — the front end as model-view-controller over a shared edit language

The front end is a small single-page application plus a set of supporting surfaces — an account view, a job history, an operator dashboard, and the editor. The piece worth drawing is the editor, because it shows the cleanest decomposition. The editor is a **model-view-controller** loop over one document, drawn in Figure 5.3-3.

- **The view** is what the user sees: two panes. The left renders the page with overlay boxes marking each structural element; the right is a list of cards, one per element, with editable fields for alt text, role, decorative-or-not, and reading order.
- **The controller** takes a user gesture — edit this alt text, change this role, drag these children into a new order — and does one disciplined thing: rather than reaching into the document and mutating it, it translates the gesture into a single typed **edit operation**.
- **The model** is the document's intermediate representation. Its only input is an edit operation. It applies the operation, returns the updated state, and the view re-renders from that state.

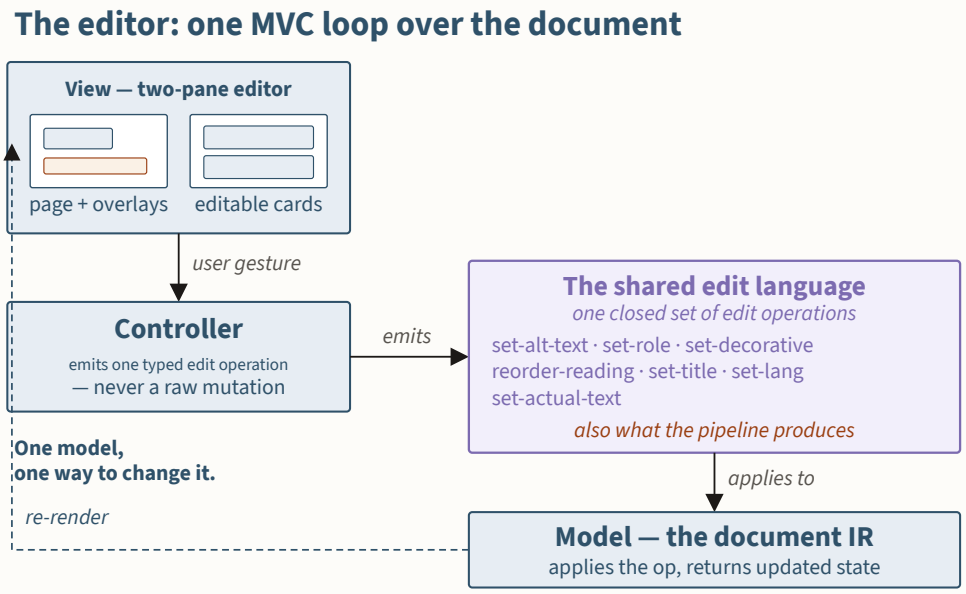


Figure 5.3-3: The editor as one MVC loop over the document. A gesture flows View to Controller; the controller emits one typed edit op from a closed vocabulary; the model applies it and returns state to re-render the View. The same edit language is what the automated pipeline produces.

The operation the controller emits is drawn from a small, closed vocabulary: set the alt text, set the role, mark an element decorative, reorder the reading order, set the document title, set the language, override the displayed text. That vocabulary is a little language for changing a document — an edit language. Every gesture routes through it for the reason that runs under the whole system: a

document has exactly one way to be changed. The editor never edits the document. It speaks the edit language, and the model is the only thing that touches the document.

Learn more about this governance mechanism: closed edit-operation vocabulary.

The read-only diff views for slides and word processor files share this shape, differing only in the adapter that fetches their structure. One surface, one edit vocabulary, many formats. The uniformity is the payoff: a fix or a constraint applied to the edit language holds for every format and every producer that speaks it — and the more producers speak it, the more each fix is worth. This is the Modeling Thesis in miniature — a structured model binding what a change *means* to how the document is *touched*, so intent and implementation cannot drift apart.

5.3.7 View 3 — the edit language as the target of automated remediation

Here the two halves of the system join, and the join is the function-call pattern from earlier in the chapter, seen once more.

The edit language the editor speaks is not only the editor's. It is the **same target the automated pipeline produces**. When the pipeline decides a figure needs alt text, the result it emits is an edit operation — the very `set-alt-text` the editor would emit if a human did it by hand. The human path and the automated path are two producers of one language, and they flow into one document model that applies each operation and stamps it.

The mapping from a remediation *task* to an edit *operation* is a function: task in, operation out. And that function is a concrete instance of the LLM as a function call. Inside it, a bounded task — this figure, this context, this question — is packed into a typed input and handed to the model as a call; the model returns a candidate; and a deterministic validator checks the candidate before it is allowed to become an edit. The edit language is the typed output contract. An honest caveat: this task-to-edit function is mostly hardcoded today and will be fleshed out. What matters now is the shape and the trajectory, not a finished implementation — the concept is that automated remediation and hand editing are the same operation reached two ways. Figure 5.3-4 draws both paths into one edit language.

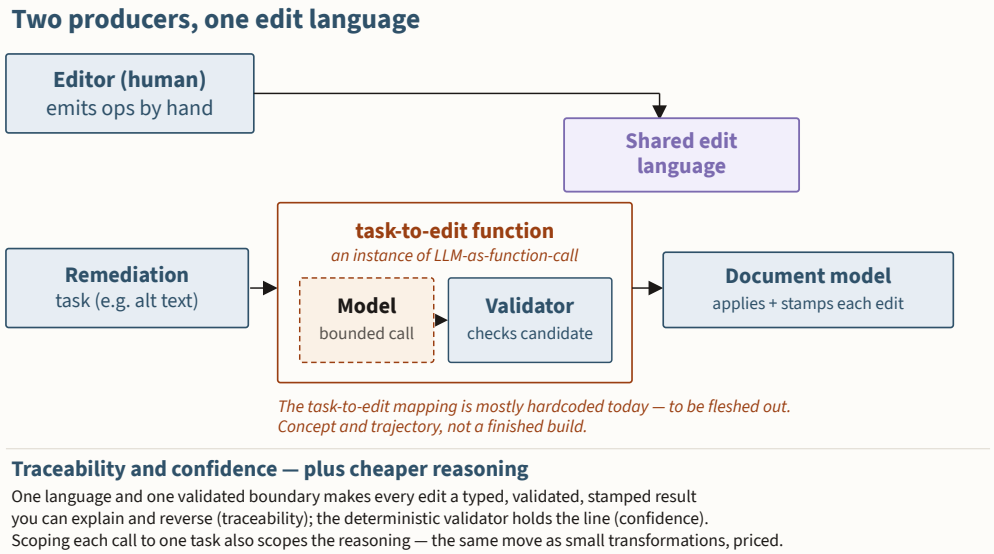


Figure 5.3-4: Two Producers, One Language. *The human path emits ops by hand; the automated path turns a remediation task into an op through a task-to-edit function — an instance of LLM-as-function-call. Both emit the same closed edit vocabulary into one document model that applies and stamps each edit.*

Why route through the language at all

The frontier models can already skip all of this. Opus and the project's own agent can take a slide deck and remediate it directly — at least for slides — with no pipeline, no edit language, no validator. So why build the machinery at all? Direct remediation buys neither of the two things a university needs. You cannot **audit** what a direct edit changed: the model hands back a file, not a list of typed changes you can inspect, explain, and reverse. And you cannot **trust** it: there is no deterministic check that the standard was met and the meaning survived. Routing every change through one edit language and one validated boundary buys both. Traceability comes free, because every edit is a typed, validated, stamped result with a history you can reconstruct. Confidence comes from the validator that sits at the trust boundary and refuses a candidate that fails.

There is a third dividend, and it connects this view to a habit the book returns to later. Scoping each model call to one bounded task also **scopes the reasoning**: the tighter the task, the less the model must reason over, and the fewer tokens the reasoning costs. A model asked to describe *this* figure under *this* context does less work than a model asked to remediate a whole document in one shot. That token-scoping is the same move as building the remediation as a **series of small, bounded transformations** rather than one giant leap — the same idea seen from the cost angle instead of the pipeline angle. A chain of scoped passes is cheaper to run *and* cheaper to reason about, because each

link asks the model for exactly one small thing. (See: *transformation* — everything a model does well is a sized transformation, and sizing the leap is the skill.)

These three views were not drawn by hand. They were rendered from DocAble's real system models — typed records the fleet reasons through — and those models stay true to the code only because a traceability substrate re-checks them against it, the deep-dive we return to in the Model Zoo.

5.3.8 Same song, second verse

Notice the shape of the argument, because the book made it again at a larger scale. A model can do the one thing deterministic code cannot — look at a figure and understand it — and it is unreliable. You do not make it reliable by trusting it more. You make a reliable system by wrapping it: bound what you ask, type what comes back, validate before you believe. Inside DocAble, the wrapped component is a vision-language model writing alt text.

Zoom out one level and the coding *agents* that built DocAble are the same kind of component: powerful, probabilistic, capable of a confident wrong turn. The methods in the earlier chapters are the same pattern applied to *them* — bound the task, type the interface, validate before you trust. You have now seen, in the small, the pattern the rest of the book applied in the large. Same song, second verse.

5.4 The Road to MAGE

> ***This chapter illustrates***
> ✓ The Modeling Thesis · ✓ The Governed Engineering Environment · ✓ Governance Conversion

The MAGE methodology emerged from months of work with coding agents. I initially expected to prompt the agents and have them make what I wanted, and for a little while that sufficed. But the more my ambition grew, the less mere prompting could carry, and I accumulated more and more of the method. Every model, every constraint, every sensor in the built system arrived the same way: a growing codebase and a rising quality bar demanded it, and I minted the smallest thing that met the demand. Then the system grew again, the bar rose again, the current system proved inadequate, and the next thing was demanded. MAGE is the name for what was left standing after enough of those rounds.

What follows walks the order in which the pieces appeared, and I suggest you read it as a staircase. Each step answers a load the step before it could no longer carry. Watch the job title climb as you go — co-coder, QA, review lead, tech lead, architect — each rung a job the one before it could no longer do, and each one handed off only because the step below had made it safe to let go. The thread through every rung is the same: the seat climbs only because each step first minted the model or the control the next rung would need. The models did accrete along the way. But the road this chapter walks is the climb of the seat, and the governance that made each rung safe to let go.

> ***Why the job title climbs***
> The whole ladder rests on one hinge, and it is **governance**. Every rung is a successive delegation. I hand the agents a piece of the *tactics* — write this code, audit that diff, run this zone — and keep the *strategy* for myself. That handoff is only safe because a control caught what the delegation would otherwise have dropped. A lint reddens on the boundary I stopped watching; a gate refuses the commit I stopped reading; a model the fleet reasons through means I no longer have to hold the map in my head. Governance is what makes it safe to let go — so delegation climbs exactly as far as the controls will carry it, and no farther.
>
> Get the governance wrong and the ladder runs in reverse. The controls that were supposed to catch the dropped thing miss it, the misses surface days later in some downstream gate, and I am pulled back down into the tactics to firefight. I am reading diffs again, chasing a drift the map no longer shows, with no one left minding the strategy. That is the *Claude is the fastest road to hell* principle at fleet scale: the same speed, now pointed through a governance gap instead of riding atop, carrying the whole crew downhill as fast as it once carried them up.

5.4.1 *Co-coder* Step one — I reviewed everything myself

At the start there was one document format, a few hundred lines, and me. When the agent proposed a change I read the diff, decided it was right or wrong, and merged it. That is the whole method in the first week: a human is the quality gate. It works, and it works precisely because the volume is small enough that one person can hold the whole thing in his head and say yes or no to every line.

The obvious trouble with that gate is that it does not scale. A real system takes more code than one person can read, and agents write code far faster than a human can review it — the more they wrote, the further behind I fell. The first crack was thus a queue: work finished faster than I could bless it, and I became the bottleneck.

5.4.2 *QA* Step two — I dispatched an agent to audit the work

The first offload was to hand the review itself to an agent. Instead of reading every diff, I dispatched a second agent to audit the first one's work and report what it found. Now the *checking* of the work had left my hands and entered the fleet. The human stopped being the reviewer of record and started being the reviewer of the reviewer.

That move taught a lesson. A test or an audit is not a chore bolted onto the work — it is a **control**, a discrete artifact that fires on a violation so a failure caught once is caught every time after. An *audit* is the weakest kind of control, because it is expensive, deferrable, and after-the-fact. It runs when I remember to run it, over a codebase that has already drifted. The audit worked, but it planted a question that drove everything downstream: which of these audit findings could become a *lint* — deterministic and cheap static analysis — so I never had to look for that class of failure again? Today's audit finding is tomorrow's lint.

5.4.3 *Review Lead* Step three — one audit became a crew

The single reviewing agent worked, but it had a flaw I could not brief away. Pointed at a hundred thousand lines, one audit agent always found *something* to complain about — it never came back empty. And it could not do two things I needed. It could not rank its own findings, so a cosmetic nit and a real architectural break arrived side by side with equal weight. And it could not tell one part of the codebase from another, so a rule that mattered in the PDF model got applied, out of place, to the web layer. A single generalist reviewer over a large codebase is a firehose with no aim.

The fix was to stop dispatching one reviewer and start dispatching a **crew** — a set of specialists, each with a narrow charter. I split the one audit into distinct briefs by *type* of concern: a **DRY** audit that hunts duplicated logic, an **architecture** audit that checks the code against its own stated rules, a **naming** audit, a **security** audit. Each brief knows one thing well and ignores everything else, so its findings are all of one kind and can be ranked against each other instead of against apples.

Aiming the crew took a second idea, and it is the one that matters for the rest of this chapter: **architectural zones**. I partitioned the codebase into named regions and let each audit run per zone. Now a finding carries both coordinates — *which kind of concern* and *which part of the system* — and the firehose becomes a grid of aimed jets. The zone idea is the seed of the first real model, and step four is where it grows up; here it is just the thing that made the crew usable.

A crew brief is a standing artifact the orchestrator re-dispatches per zone. Here is the head of the real **architecture** audit — the one that checks code against the architecture rules the project has written down for itself:

A standing brief — the architecture audit

```
# Architecture Compliance Audit Agent
```

```
**Purpose:** Verify that code follows the architectural rules and
invariants documented in CLAUDE.md, DESIGN.md, and ORCHESTRATION.md.
Catches violations before they become production bugs.

## What to check

For each rule below, verify the code matches what the docs say. If
the docs have been updated but the code hasn't, that is a finding.

- **Library discipline:** Verify PDF production code uses the one
  sanctioned PDF library only, that the tag-tree-stripping library is
  absent from tag-tree operations, and that GenAI calls go through the
  single client factory.
- **State machine discipline:** Verify the async job lifecycle uses the
  explicit state machine with a terminal-state guarantee in `finally`,
  and that metric hooks are wired as state-entry callbacks, not
  scattered calls.
- **Redis vs DB responsibility:** Flag durable state kept in Redis, or
  dispatch queues kept in the database.

## Output

Each finding: file:line, rule violated, 1-2 sentence description.
Cap at 15 findings. Prioritize by production-incident risk.
```

Read the brief and the two ideas are both visible. The `## What to check` list is the *type* charter — this agent reasons about architectural rules and nothing else. The `Cap at 15 findings. Prioritize by production-incident risk.` line is the fix for the firehose: a narrow charter can rank, because everything it returns is the same kind of thing. And this brief runs once per zone, so the same rules get checked against the PDF model, the web layer, and the dispatch plane as separate passes — the *target* coordinate.

The crew taught the lesson that drove everything after it. Once I was paying agent tokens to run these audits on a cadence, the waste was obvious: an audit is a probabilistic agent re-reading the code every time, and much of what it checks is *deterministic*. "Does any file outside the PDF zone import the PDF library?" does not need a language model — it needs a lint. So I leaned hard into two cheaper controls that match the crew's briefs one-for-one. First, **commodity lints** — the off-the-shelf static-analysis rules that catch the generic smells for free. Second, **custom lints** written to mirror each audit brief: the architecture audit's "one sanctioned PDF library" check became a boundary lint that reads the zone declaration and reddens on a cross-zone import, deterministically, at commit time. The audit crew did not disappear; it retreated to the findings a lint genuinely cannot make — the ones that need judgment. Everything a machine could decide moved down to the machine. This is the audit-to-lint move — today's audit finding is tomorrow's lint — run at fleet scale, and it is what freed the crew to be worth its tokens.

5.4.4 *Tech Lead* Step four — zones, so an agent can reason locally

A crew that coordinates is still a crew of agents each of whom has to understand the code it touches. And here the agent's nature bites. A cold-start agent with a finite context window cannot reach the tacit knowledge a human team keeps in its collective head — which file is canonical, what this module *means*, which of two look-alikes is the live one. A human learns that by working in the codebase for months. Think of the agent instead as a contractor with no memory of yesterday: it arrives knowing nothing and has to be *told*, in the code, every time.

The answer was to partition the system into named **zones** — a model that says, out loud and in a queryable form, what the parts are and where their boundaries lie. Once the system is carved into zones, an agent can be handed one zone, reason inside it locally, and be fenced from the rest. A boundary lint can then refuse an edit that reaches across a zone it has no business touching. The zone model turns "understand the whole 280-thousand-line codebase" — which no finite-context agent can do — into "understand this zone," which it can.

A zone is a small typed record. Here are two real ones — the PDF and PowerPoint object models, side by side:

Two zones, side by side

```
Component(
    name="pdf-model",
    focus_dirs=(RepoRelPath("backend/src/AdaTool.PdfModel"),),
    boundary_kind="internal",
    external_seams=("pdf-lib",),          # ONLY this zone may call the PDF
library; banned in every other zone
    description="PDF object model + typed mutators",
),
Component(
    name="slides-model",
    focus_dirs=(RepoRelPath("backend/src/AdaTool.SlidesModel"),),
    boundary_kind="internal",
    external_seams=("openxml",),          # ONLY this zone may call the Office
library; banned in every other zone
    description="PPTX Office-XML model + typed mutators",
),
```

Read `external_seams` and the whole discipline is legible. The `pdf-model` zone is the *only* one allowed to reach the canonical PDF library; `slides-model` is the only one allowed to reach the Office library. A boundary lint reads that field at check-time and reddens on any file outside the `pdf-model` zone that imports the PDF library directly — the "one structured model per format" rule from the built-system chapter, enforced not by vigilance but by a zone declaration a machine walks. The agent working the PDF zone is handed `focus_dirs` as its lane and `external_seams` as its permission slip, and it cannot wander.

The zone model is the first piece of MAGE proper, because it is the first time I wrote down a *model of the system's own structure* for the fleet to read, rather than a piece of the product. Its audience is the fleet; a user never sees it.

5.4.5 *Architect* Step five — the codebase modeled, from components.py to MAGE

By this step I never looked at code at all. What changed is that I stopped *holding the system in my head* and started committing it into explicit models — the system the fleet reasons *through*. It has one shape with three movements: a first model the fleet reads at runtime, a set of sibling models that grew up around it, and a substrate that keeps all of them honest. Together they are MAGE.

The first movement was to make the zone model a file the fleet reads at runtime. That is where the method crosses from *documenting* the structure to *executing* on it.

A typed registry of the system's own components — its zones, their focus directories, the seams each one is allowed to cross — is not documentation *about* the architecture. It *is* the architecture, in a form a tool can query. The zone I showed above is one row of it; the row's *shape* is a frozen dataclass:

The registry the fleet reads

```
@dataclass(frozen=True)
class Component:
    """One system component."""
    name: str
    focus_dirs: tuple[RepoRelPath, ...]
    tags: frozenset[str] = field(default_factory=frozenset)
    kind: str = "leaf"                                    # leaf | group |
meta
    external_seams: tuple[ExternalSeam, ...] = ()         # the powerful libs
it may reach
    import_allowed_components: tuple[ComponentName, ...] = ()  # who it may
import
```

Every field is a machine-readable answer to a question an agent would otherwise have to guess. `focus_dirs` answers "what files are in my lane?" `external_seams` answers "which powerful library am I permitted to touch?" `import_allowed_components` answers "who am I allowed to depend on?" When a lint needs to know "which components may cross which external boundary," it does not hardcode a list that rots — it reads `external_seams` at lint-time. When the dispatcher needs to know which files a brief may touch, it reads `focus_dirs`. When context has to be injected into an agent's prompt, the question stops being "which files do I paste?" and becomes "what durable artifact should exist so the right context is *selectable deterministically*?"

The payoff was concrete and it compounded. Adding a single typed field to that registry — one field naming which components cross an external boundary — replaced four hand-maintained allowlists that were quietly drifting and unlocked three new lints all reading the same field. On its first run the new boundary lint surfaced a large backlog of latent violations no one had had a way to ask about before. The type named a previously-anonymous shape, and the named shape became a question you could pose to the whole codebase at once. The reflective codebase *is* the agent substrate. The more the system knows about itself, the less the agent has to guess.

The refactoring got cheaper because the models told me exactly where to look and where to stop looking. They took the guesswork out of migration planning: instead of a fake-confident Claude analysis I got a real-confident one I could trust. A cross-format migration that touched fifty-eight files and rewrote five thousand lines — the kind of change I might once have deferred — took about five

hours of agent time over a dinner break. When the mechanical cost of restructuring collapses, the economics of architecture invert: postponing the right shape becomes the expensive choice, and building the right model becomes something you can do the moment you see you need it. The hard part was never touching fifty-eight files; it was knowing that fifty-eight files should be touched. The mechanics went free. Deciding *which* model to build did not.

A view per property I had to guarantee

Once that first model paid off, the others followed — but the right way to see the set is not "more models." Each model is a **view** of the system, and each view exists to answer for one **property or quality metric** I had to guarantee: a property like *deployability*, a quality metric like *security*. The set was never planned. What grew was the *demand*. The faster the fleet moved, the higher the bar on each property rose, and a rising bar on a property I could no longer hold in my head is exactly what forced a view to carry it. The models accreted on demand, one per quality the growing system made me answer for.

- **A service-flow model** — the view that guards *event-wiring correctness*. It appeared when the back end became enough services that no one could hold the wiring in their head: which handler reacts to which topic, what the durable-versus-transient split is. That view is what later made a hard re-platforming close to a change of deployment target.
- **Synchronization contracts** — the view that guards *concurrency safety*. They appeared when concurrent agents and concurrent workers started tripping over shared state, and the invariants that had lived only in prose had to be encoded where a checker could walk them.
- **A deployment-topology model** — the view that guards *deployability*. It appeared when the system ran in more than one place and the boot-time environment each service needs stopped fitting in anyone's memory.
- **Domain registries** — the view that guards *vocabulary closure*. The closed edit vocabulary, the remediation verbs, the mutator stamps: each appeared when a set of strings needed to become a typed, enumerable thing a lint could hold closed.

None of these was drawn from a master diagram. Each was the smallest view that answered for one rising property. But a stack of views has a failure mode of its own, and it is the one that turns a map into a lie: **drift**. A view that names a code symbol goes stale the moment that symbol is renamed, deleted, or moved, and nothing notices — the view still *looks* authoritative while pointing at a ghost. The fleet reads the ghost as truth, and the blow-up lands days later in a downstream gate instead of at the change that caused it.

The substrate that keeps the models honest

So the models grew a nervous system. A traceability substrate links every model to its lint, its code entry-point, its proof, and its sibling models as a typed graph whose every edge is a *derived* obligation a lint re-checks against a resolvable symbol — never a snapshotted line number. Derived edges defend; snapshotted ones drift.

Drift comes in two kinds, and the substrate catches each a different way, because the two are not the same problem. The first kind is **deterministic**. The code moved: a symbol was renamed, deleted, or hoisted into a different module, and a view's edge now points at nothing. That is mechanically decidable, so a machine decides it. A lint walks every edge in the graph and re-resolves its anchor against the live code; an edge that no longer resolves reddens on the spot. There is no judgment in it — the symbol is either there or it is not — so this is a lint, run deterministically, and a broken edge fails at the change that broke it rather than days downstream. The second kind is **judgment**. The code

still resolves — every anchor points at a real symbol — but the *change altered what the view claims*. A function still exists, but it no longer does what the invariant says it does; a checker was quietly made stricter than the producer it now rejects. No lint can see this, because nothing on disk is broken; the map matches the territory symbol-for-symbol and lies anyway. So this drift is caught by a review gate, not a lint: the definition-of-done audit at the close of every unit of work asks, in as many words, *did this change alter a model?* — and routes a "yes" to whoever owns that view. Deterministic drift is a wall; semantic drift is a question you have to keep asking.

That pairing — a lint that walks the anchors for the drift a machine can decide, and a review gate that asks the human-or-agent question for the drift it cannot — is the whole mechanism, and it is instructive enough to have its own catalogue entry: symbol-anchored traceability graph. The graph is what keeps the map equal to the territory, which is the only condition under which a context-bounded agent can safely operate a context-exceeding codebase *through* the models. The structured models, the substrate that keeps them honest, and the query surface the fleet reasons through — together these are MAGE. Figure 5.4-1 draws the five-rung climb.

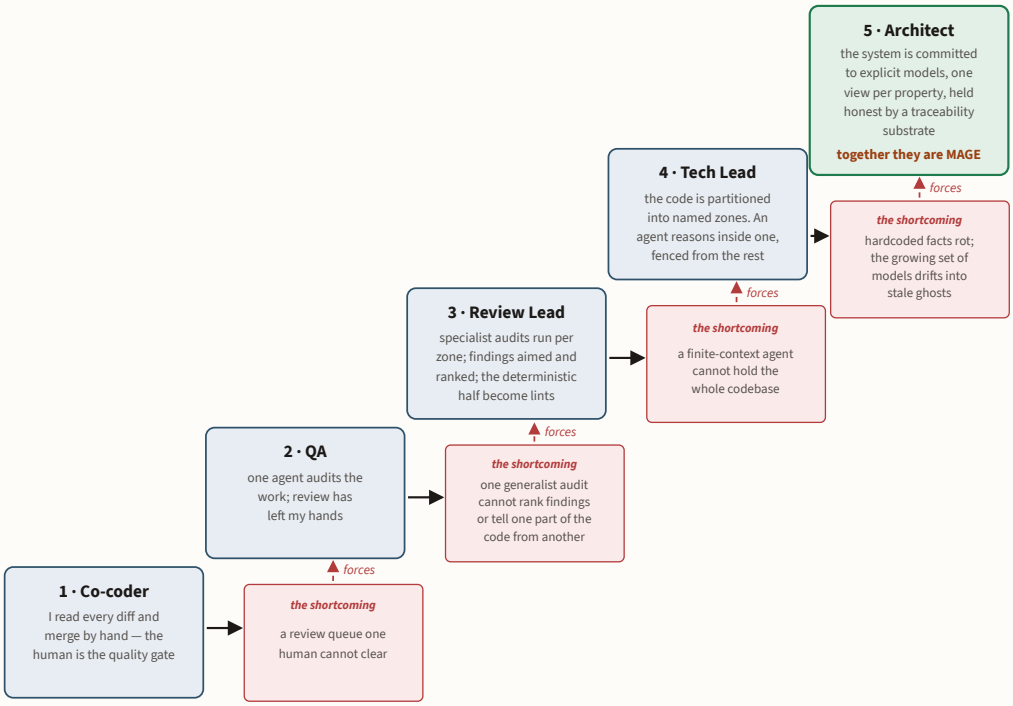


Figure 5.4-1: The MAGE Staircase. *Each rung fixes the shortcoming that broke the one below; the ladder ends where the system is committed to explicit models held honest by a traceability substrate — MAGE.*

The Architect's newest instrument: measuring what is still unmodeled

The Architect rung is where the ladder ends, but the work on it does not. Every model so far was built the same way: I noticed a load and modeled it. That is a *supply-side* method — it models what I happen to see. Once there are a dozen models and a traceability graph, the obvious question is the one I could not answer by looking harder: **what is still unmodeled?** That is the Brownfield problem, turned inward on my own system. A greenfield method invents the models it wants up front; a brownfield method inherits a large, already-built system and has to *discover* the models the system already implies but nobody has drawn yet. That is the situation I was actually in, and it needed a different instrument.

The instrument is a metric that is the dual of test coverage — the **Missing Model Metric**. Ordinary coverage walks model-to-code: for each model node, is it tested? This one walks the other way, **code-to-model**. Start from a test, follow the production code it exercises by static reachability, and ask whether any of that code carries a model anchor — an invariant, a state machine, a service edge. A test whose exercised code reaches *no* anchor is an **orphan**, and the orphans, clustered by the unmodeled code they run through, are a candidate list of *missing models*. It is negative-space modeling: instead of "is my model tested," it asks "what am I running in production that no model has ever named?" On its first run over the dispatch subsystem — the *most* heavily modeled corner of the codebase, where orphans should have been rare — more than half the tests came back orphaned. A sharpening pass turned that scolding into a map: every orphan reached code the model already knew at the coarse structural level, but that carried no fine behavioral model. The structural map was complete; the behavioral map covered the concurrency-critical spine and stopped. The metric had found a *granularity gap*, and it named exactly where to close it — three genuine candidate models: the durable-state persistence layer, the cost-and-subprocess pipeline, and the chunking-merge-and-validation path.

That map drove a loop, and the loop has run. Measure the orphan rate; rank the biggest clusters; for the largest, found a modeling Epic and drive it to a strong structured model under a real design review; re-measure; take the next cluster. The key property is the one that makes it a *modeling* instrument rather than a testing one: you do not move the number by adding a test — you move it by adding a *model*. Wave by wave, the loop worked as designed. The three seed models drove the dispatch subsystem's orphan rate down from the low-fifties percent to the high-thirties, and the next cluster — the sidecar persistence seams — took it into the low thirties. Each closed model converted exactly the tests that reach its new anchors from orphan to modeled, which is the falsifiable signal that the metric is measuring what it claims.

The loop was not bookkeeping. Building those models found real, shipping bugs the green tests had missed — the payoff that justifies the whole method. Modeling the cross-language pipeline seam caught a subprocess exit code that one runtime emitted and the other had never enumerated, so a "the service should have worked" failure was being silently misfiled as a generic error. Anchoring the chunking model to its one canonical cleanup seam exposed a failure path that purged a job's database rows and blobs but skipped the queue structures its sibling path always cleaned — a chunk leak on a rare redelivery edge. And modeling the interactive editor's save round-trip surfaced an unfenced last-write-wins race in which two concurrent edits silently discarded one's work. None of these was a hypothetical the model raised; each was a live defect the act of modeling forced into view.

So the method that began with a human reviewing every line arrived at a system that measures its own modeling completeness and dispatches work to close the gap it finds — a codebase modeling *itself*, brownfield, in a loop. MAGE was never designed. It was demanded, step by step, by a system that kept growing and a bar that kept rising — and the last thing it demanded was that the method turn its own instruments back on the parts of itself it had not yet named.

5.4.6 The whole method, made real

This book opened on one diagram — The MAGE Method at a Glance — that named every part before the argument had earned it. Walk its boxes one last time, and each is now a real thing in a system that ships:

- **The Printer** ✓ — a fleet of agents printing production code faster than any human could read the diffs.
- **The Modeling Thesis** ✓ — the system committed to explicit models the fleet reasons through, from the zone registry to the view-per-property stack.
- **The Alignment Thesis** ✓ — every probabilistic part, the vision model and the coding agents alike, sealed in a bounded, typed, validated call.
- **The Governed Engineering Environment** ✓ — the models, the controls, and the query surface, grown until the apparatus outweighed the product it guards.
- **Governance Conversion** ✓ — every control minted the moment a rising bar demanded it, one failure at a time.
- **Trustworthy software** ✓ — DocAble, live in beta, remediating a real university's documents for dollars a deck.

Six boxes, all checked. The case study was the evidence; the diagram was its table of contents.

6.0 Toward a Theory of MAGE

The book has argued from one case, seen in depth. A theory is what lets that case speak past itself: it names the constructs, draws the mechanisms between them, and states what evidence would make the account fail.

The theory offered here is middle-range. It does not claim that every software project will behave like the one this book studied, and it assigns no universal coefficients to the relations it draws — a single system cannot support them. What it offers is a dynamic model of a specific phenomenon: how a fleet of coding agents turns abundant implementation capacity into durable engineering progress, or into churn. Read the whole chapter with that scope in mind; I will not restate it under each claim.

The central claim is not that agentic velocity is inherently productive. Velocity amplifies the fit between the fleet and the engineering environment it runs in. When that environment is incomplete, incoherent, or too weak to hold the work, velocity produces churn, escaped defects, repeated intervention, and governance friction. Those signals create pressure to adapt the environment. Engineers and engineering organizations respond by modeling missing structure, converting recurring failures into durable mechanisms, and reconciling or retiring governance that has grown excessive. When the adaptation works, later agents inherit a more legible action space, and a greater share of raw velocity becomes durable progress.

[54] Donella H. Meadows, *Thinking in Systems: A Primer* (Chelsea Green Publishing, 2008).

6.0.1 The dynamic model

Figure 6.0-1 draws the whole theory on one diagram: a single set of constructs rather than four separate pictures, because the four dynamics act on the same system. The figure follows the causal-loop convention of system dynamics[54].

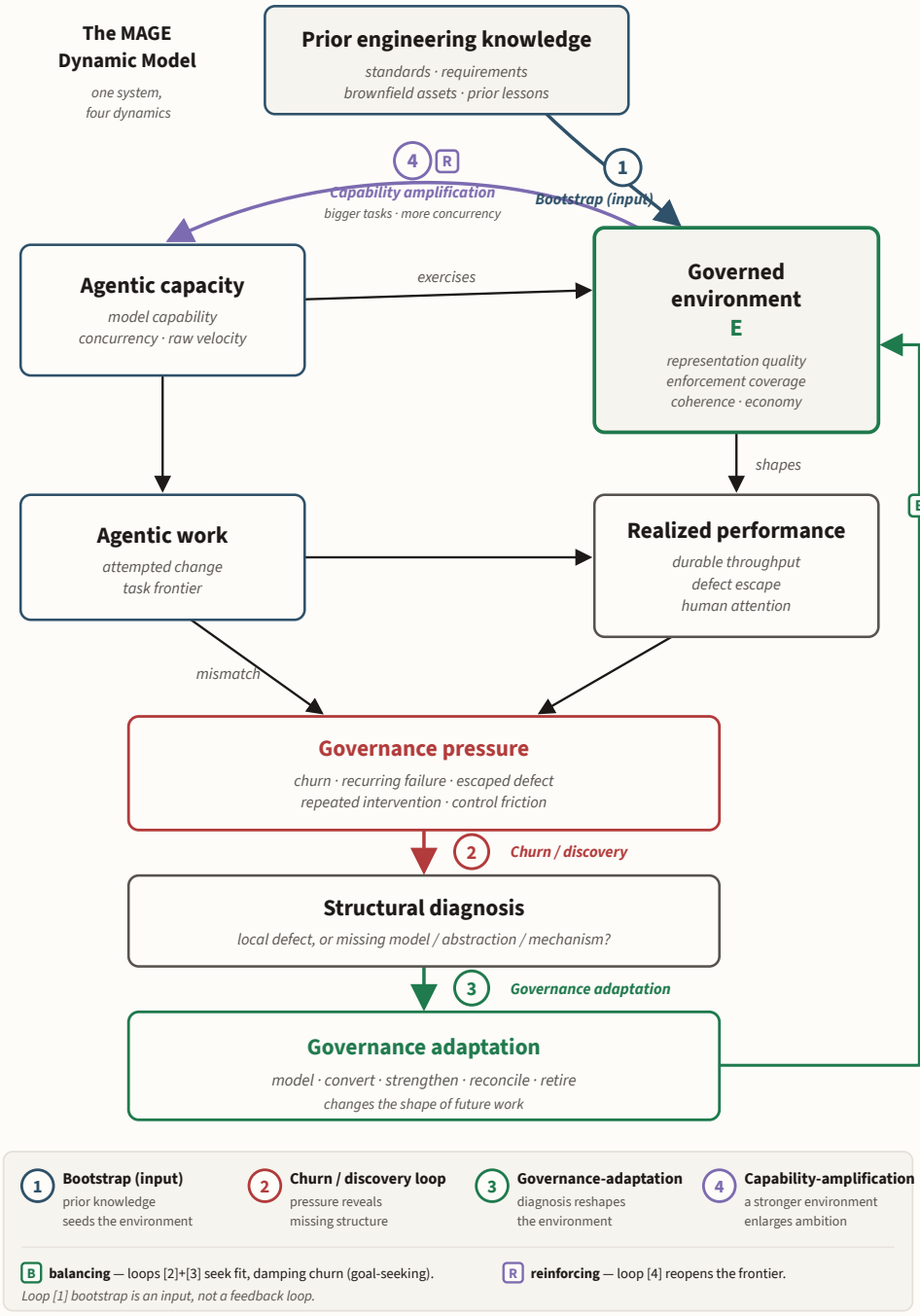


Figure 6.0-1: The MAGE Dynamic Model. *Eight constructs, read clockwise: capacity exercises environment E and drives work; work and E yield realized performance; the gap becomes governance pressure, then structural diagnosis, then adaptation, which reshapes E. Four numbered dynamics overlay it, legible in grayscale.*

The spine runs left to right. **Agentic capacity** — model capability, concurrency, raw implementation rate — exercises a **governed environment** whose effective quality the model calls *E*. That environment shapes **agentic work** into **realized performance**: durable throughput, defect escape, and the human attention each unit of durable output costs. The gap between the performance the fleet looks capable of and the durable performance it delivers registers as **governance pressure**. Pressure meets **structural diagnosis** — the judgment that reads a signal as a local defect or as evidence of missing structure. Diagnosis drives **governance adaptation**, which changes *E*. The changed environment reopens the loop.

Four numbered dynamics run over that spine, and each has a color and a number so the figure survives in grayscale:

- **Blue — the bootstrap path.** Prior engineering knowledge seeds the initial environment before any agent runs.
- **Red — the churn/discovery loop.** Work outruns the environment; churn and failure make the missing governance visible.
- **Green — the governance-adaptation loop.** A structural diagnosis reshapes the environment for the next agent.
- **Purple — the capability-amplification loop.** A stronger environment raises ambition and concurrency, which exposes a new frontier.

Follow the red loop first. It carries the experienced driver of the whole system — the pressure an engineer actually feels — and the other three are best read as answers to it: blue lowers the pressure before it starts, green relieves it once it appears, and purple explains why it never stays relieved.

The four dynamics are not all the same kind, and naming their types sharpens why the system does not settle. Three are feedback loops; bootstrap is the input that sets the initial condition. Loops [2] and [3] together form a **balancing** loop — churn drives diagnosis, diagnosis drives adaptation, and adaptation reduces churn: a goal-seeking cycle that drives the environment *toward fit*. Loop [4] is a **reinforcing** loop — a better environment raises ambition, and greater ambition reopens the frontier: a self-amplifying cycle that keeps moving the target. The balancing loop settles the environment toward fit; the reinforcing loop moves the target the moment it is reached. That pairing — balance toward fit, then reinforce to reopen the frontier — is why MAGE never converges.

6.0.2 The four dynamics

Each dynamic gets its own section below, and each opens with a simplified diagram of just its local causal path. The master figure above already established that the four are one system, so each section shows only the fragment it explains.

The blue path: bootstrap governance

Not all governance is discovered through failure; the blue path, in Figure 6.0-2, seeds the environment before any agent runs.

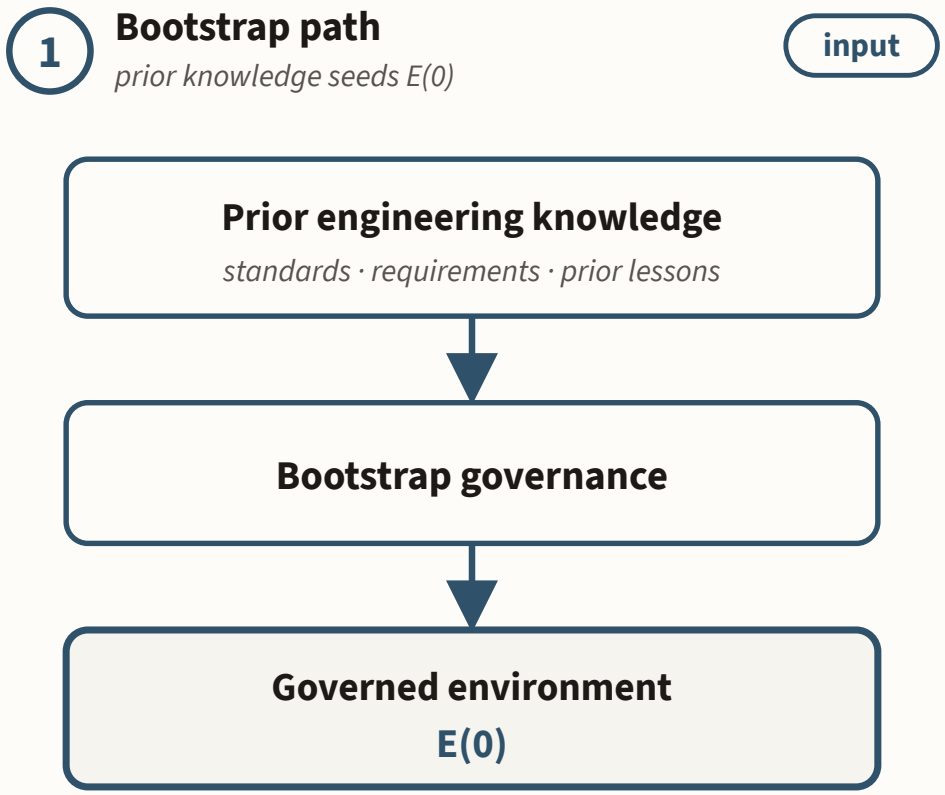


Figure 6.0-2: Bootstrap path. *Inherited and anticipated knowledge establishes the initial governed environment before the fleet encounters project-specific failures.*

A project begins with knowledge it inherited and knowledge it can anticipate. A brownfield system already carries architecture, tests, deployment controls, operational history, and institutional obligations. A greenfield project starts with less local evidence, but a skilled engineer can still seed it from prior experience, external standards, known security and reliability practices, and lessons transferred from earlier systems.

Bootstrap governance converts that ex-ante knowledge into the initial governed environment: a known obligation or a prior lesson becomes a model, a mechanism, or a constrained architecture, and the sum of them sets the environment's starting quality. This is why disciplined organizations and experienced engineers need not rediscover every failure from scratch. Governance conversion in one project becomes bootstrap knowledge in the next.

Bootstrap governance is never complete. The environment a product needs depends on the product, its trajectory, and the agents acting on it, and none of that is fully knowable in advance. The blue path lowers the initial failure rate. It does not close the red loop.

The red loop: churn makes missing governance visible

The red loop, in Figure 6.0-3, is where work outruns a weak environment and the gap surfaces as pressure.

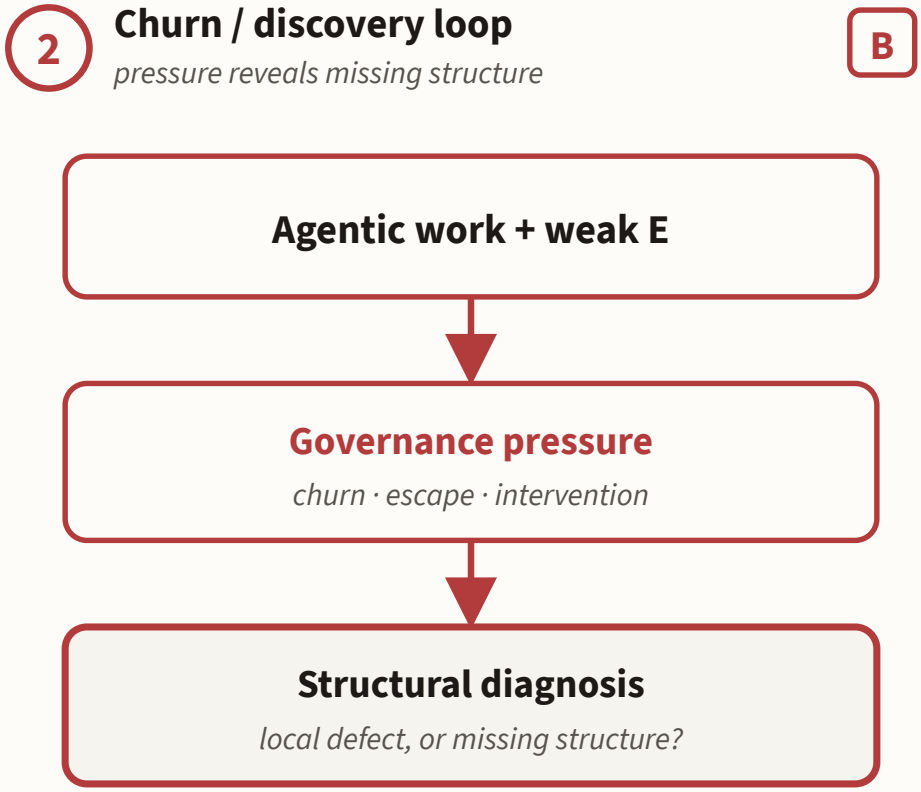


Figure 6.0-3: Churn/discovery loop. *Work outruns the environment; the resulting pressure becomes useful only when judgment recognizes a structural failure class.*

Agentic velocity is not the immediate motive for governance investment. The motive is the experienced gap between what the fleet appears capable of producing and what becomes durable progress. When the environment is weak, raw implementation capacity produces repeated rediscovery, regressions and reversals, escaped defects, recurring human intervention, control friction, and confidently wrong changes that consume tokens without advancing the system. The book calls the dominant form **churn**.

Velocity earns its place in the model as an amplifier. It clusters latent weakness in time: failure classes that might have arrived months apart under human-paced work land in a single afternoon, and their proximity makes the structure they share visible. Governance pressure is what the operator feels when that happens.

Pressure is not yet knowledge. The next act is judgment. **Structural diagnosis** distinguishes a local defect from a structural failure class. A local defect gets repaired and the matter ends. A structural failure reveals a weak or absent model, abstraction, boundary, invariant, oracle, mechanism, or governance seam — something whose absence will keep producing failures until it is supplied. This diagnosis is the scarce work of the whole method, and the red loop is where it is demanded.

The green loop: governance adaptation

A structural diagnosis triggers governance adaptation, the green loop of Figure 6.0-4.

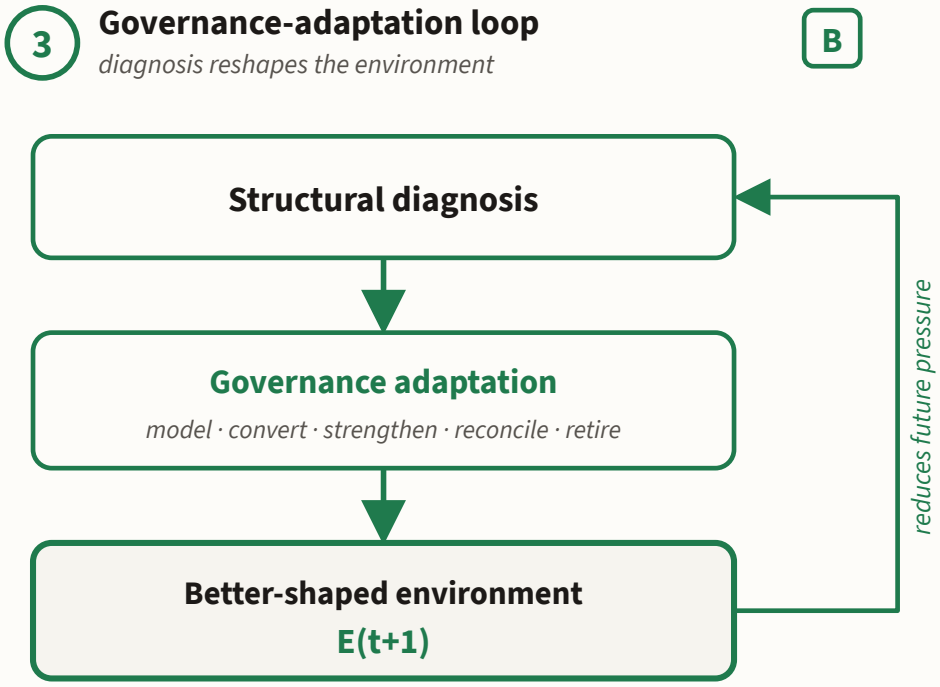


Figure 6.0-4: Governance-adaptation loop. *Diagnosis changes the environment so that later agents inherit a more legible and constrained action space.*

The core move here is **governance conversion**: turn a recurring failure class into a durable mechanism the environment enforces on every later agent. Mature adaptation keeps that move and adds four more around it:

- **Model** — make missing intent or structure legible, so agents reason over the right representation.
- **Convert** — turn a recurring failure into a durable mechanism.
- **Strengthen** — replace weak probabilistic guidance with harder enforcement where the obligation justifies it.
- **Reconcile** — resolve controls that duplicate, conflict, or fire in the wrong order.
- **Retire** — remove governance that has gone stale or become net-harmful.

The output is not merely more controls. It is a change in the shape of future work. Later agents inherit a more compact representation, clearer constraints, fewer invalid moves, earlier evidence, narrower sanctioned seams, and more reliable admission conditions. That inheritance is the immediate mechanism by which the environment's quality changes — and it is why adaptation can be subtractive as well as additive: reconciling a control collision or retiring a stale gate raises quality without adding a thing.

The quality that adaptation moves needs a definition sharp enough to keep apparatus quantity from standing in for it.

Effective governed-environment quality, *E*. The degree to which the environment gives agents a current, task-relevant representation of the system, together with coherent, proportionate, machine-actionable coverage of its important obligations.

That quality has four parts, set out in Table 6.0-1; each asks a question the raw apparatus count cannot answer.

Table 6.0-1: The four dimensions of E. *Effective governed-environment quality has four parts; each asks a question the raw apparatus count cannot answer.*

Dimension	The question it asks
Representation quality	Are the relevant properties legible in the environment's current models?
Enforcement coverage	Are the important known obligations held mechanically, not merely watched?
Coherence	Do the mechanisms compose without harmful collision or duplication?
Economy	Is the governance burden proportionate to the risk and value it controls?

Defined this way, *E* is a quality, not a quantity. A large support apparatus raises *E* only if it is coherent, economical, and actually covers the obligations that matter; the same apparatus can instead be duplicated machinery, stale enforcement, or a tower of governance that lowers *E* while the control count climbs. Lint count and support ratio are indicators of investment. They are not substitutes for quality.

The purple loop: capability amplification

Successful governance does not end the process; the purple loop of Figure 6.0-5 reopens it.

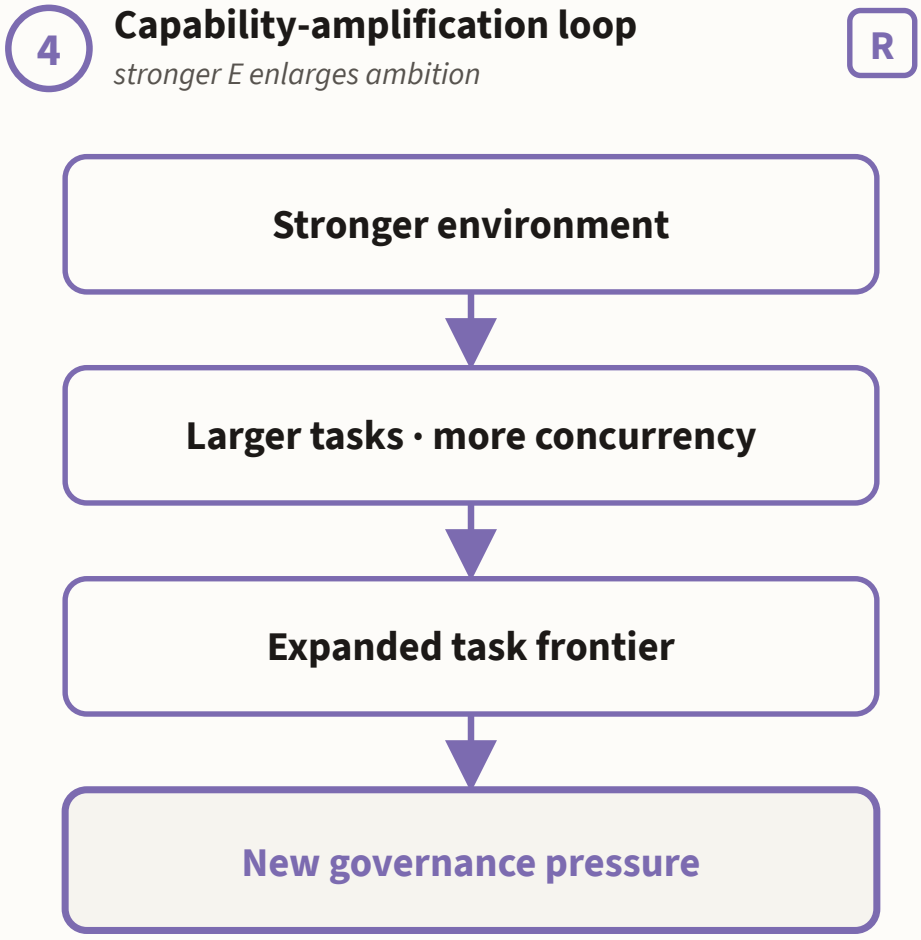


Figure 6.0-5: Capability-amplification loop. *Successful governance enables more ambitious work, which exposes the next unguided surface.*

It expands what the engineer attempts. A stronger environment makes larger delegated tasks feasible, supports greater concurrency, invites more ambitious product requirements, cheapens refactoring, and lets the operator run agentic capacity denser. Every one of those pushes the work into surface the environment has not yet governed, and the expanded frontier exposes new weaknesses.

This is **capability amplification**, and it is why MAGE does not terminate. The environment never converges on a final set of controls, because the product, the fleet, and the task frontier keep changing underneath it — and each gain in governance raises the ambition that finds the next gap. The loop also answers a natural objection: that better agents will eventually remove the need for governance. They do the opposite. A more capable fleet raises attainable velocity and ambition, which raises the consequences of a weak model or a missing control. The frontier moves out with the capability, and governance follows it.

6.0.3 Constructs and path-specific moderators

The two conditions the book leaned on — capability-fit and authority — are not global switches over the whole model. Each acts on one specific transition, and stating where sharpens what each one predicts. Table 6.0-2 places every construct and moderator on the transition it governs.

Table 6.0-2: Constructs and moderators. *Each construct's role in the dynamic model, and the transition it operates on. Capability-fit and authority attach to specific steps, not to the model as a whole.*

Construct or moderator	Role in the model	Operates on
Agentic capacity	Model capability, concurrency, raw implementation rate	Input to agentic work
Governed-environment quality *E*	Effective quality of models and mechanisms, not their quantity	Work → realized performance
Governance pressure	Churn, escapes, intervention burden, control friction	Performance gap → diagnosis
Structural diagnosis	Reading a failure as local, or as evidence of missing structure	Pressure → adaptation
Governance-adaptation capability	Capacity to model, convert, strengthen, reconcile, and retire	Diagnosis → environment change
Capability-fit	Fit between the work, agent capability, and the judgment a diagnosis needs	Moderates pressure → diagnosis
Authority	Ability or organizational path to modify shared architecture and controls	Moderates diagnosis → adaptation
Domain governability	Degree to which important properties can be represented and mechanically checked	Moderates *E* → outcomes
Control-estate coherence	Degree to which mechanisms compose without harmful interaction	Moderates apparatus → *E*

Capability-fit sits on the pressure-to-diagnosis step. A capable fleet given to someone who cannot tell a local defect from a structural one produces motion without direction; the pressure arrives, but no diagnosis converts it. **Authority** sits one step later, on diagnosis-to-adaptation. The person who reads the recurring failure has to be able to change the environment — the architecture, the mechanisms, the gates — or the diagnosis dies as a local patch and the class never dies with it.

Authority need not live in a single architect. In an organization it is a functioning path from an observed failure to shared infrastructure. Split ownership across teams, owners, and review boards

is not fatal on its own. The failure comes when the path terminates in a local patch: the same signal that could have become a shared mechanism produces a private fix instead, and the class survives to recur under the next owner.

6.0.4 A capability model, not a maturity model

MAGE is a capability model, not a maturity ladder, and the distinction matters because the field has been here before. For two decades the dominant frame for engineering quality was the staged maturity model — the CMM and its CMMI successor — which ranked an organization on a five-rung ladder and told it to climb. A staged ladder imposes one linear progression on every organization regardless of where its actual constraint sits, grades process conformance in place of the outcomes the process was meant to produce, and under pressure decays into box-ticking, where reaching a level stands in for shipping better software[55]. The objection is not to structure. It is that a ladder makes the rung the goal.

The dynamic model gives the alternative directly. A project's current bottleneck selects the next modeling or governance capability it needs; the bottleneck in front of you picks the next control, and no rung waits above it. When escapes climb, you reach for the artifact-side controls. When agents churn against a surface they cannot hold, you invest in the models. When recovery keeps pulling a human back in, you harden the fleet's substrate. Dependencies may impose a local adoption order — some controls presume others — but that order is a dependency, not a rank, and no universal sequence ranks every organization. The reason there are no levels is the reason there are no fitted coefficients: a single case cannot validate a progression that means the same thing everywhere. Different projects enter with different bootstrap environments and feel different pressures, so what replaces "what level am I" is a plainer question — which capabilities do you have, and what does each one buy against the bottleneck you actually face.

6.0.5 Outcomes and observables

The model earns its keep only if its outcomes can be watched. Three of them carry the theory, and they are the three dimensions the productivity literature already names — velocity, quality, and sustainability — read for a fleet rather than a team. Table 6.0-3 states each and the observables that would track it.

Table 6.0-3: The Outcomes. *The three primary outcomes the theory predicts, the way to state each so it reads as durable progress rather than raw activity, and the observables that would track it.*

Outcome	What it measures	Example 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

[55] Terry B. Bollinger and Clement L. McGowan, "A Critical Look at Software Capability Evaluations," *IEEE Software* 8, no. 4 (1991): 25–41, https://doi.org/10.1109/52.300034.

Outcome	What it measures	Example observables
Human-attention burden	Human judgment required per unit of durable output	interventions per landed change; review time; conversion effort

State human-attention burden as a rate, not a ceiling. The theory predicts amortization: as recurring failure classes convert into durable governance, the attention required per landed change falls, even if total governance work continues to grow. Per-change attention declines; the aggregate need not.

Apparatus measures are indicators, not quality

A second discipline governs how the environment itself is read. The quantities easiest to count — the support ratio, the model-coverage fraction, the control count, the gate count, the model-sync rate, the control-interaction findings, the stale-control rate — measure investment or accumulation. They do not measure whether the environment is any good.

A large support apparatus can be effective governance. It can also be duplicated machinery, accidental complexity, stale enforcement, a tower of governance, or an internally conflicting control estate. The number cannot tell you which. A high support ratio gains meaning only when it is joined to an outcome — when the apparatus that leads production is shown to sustain throughput or drive escape down. Treat every apparatus measure as an input or an indicator. Effective governed-environment quality is what the outcomes reveal, and the apparatus is only its raw material.

6.0.6 Principal predictions

Here is where the model sticks its neck out. The dynamic model yields **seven principal hypotheses**. Each names a mechanism and the single observation that would sink it. Core causal propositions appear here; derivative consequences and domain-specific instances follow as corollaries and specializations.

None of the seven is *tested* by this case. One repository cannot test a claim about a population. Every row is offered so a larger study can, and every row is stated as a directional, conditional claim — the shape it should take, and the conditions under which it should hold — not a measured law. Table 6.0-4 lists the seven with the observation that would falsify each.

Table 6.0-4: The Seven Hypotheses. *MAGE's principal predictions, each with the observation that would falsify it. Directional and conditional; single-case; offered for replication, not proven.*

ID	Hypothesis	Key falsifier
H1 — Failure-class exposure	Higher agentic velocity shortens the elapsed time to recognize recurring failures as one shared structural class, controlling for the opportunities to trigger the class.	Recognition latency does not improve with velocity once exposure opportunities are controlled.
H2 — Governance moderation	Governed-environment quality sets the sign of velocity's effect: higher velocity raises churn in low-quality environ-	Velocity relates to durable throughput the same way regardless of environment quality.

ID	Hypothesis	Key falsifier
	ments and durable throughput in high-quality ones.	
H3 — Mechanized assurance	Greater risk-weighted coverage of relevant failure classes by effective deterministic mechanisms lowers defect escape, and its advantage over attention-based control widens as velocity rises — for adequately scoped mechanisms.	Mechanized coverage adds no protection, or its advantage over review does not widen as human attention saturates.
H4 — Representation leverage	**H4a — Representation efficiency.** Task-relevant structured models reduce context cost and churn without lowering task quality. **H4b — Synchronization signal.** Model-synchronization measures provide predictive information about later churn or defect escape beyond conventional code-coverage measures.	**H4a:** Model use at matched recall yields no efficiency gain or lowers task quality. **H4b:** Synchronization measures add no predictive power over conventional code-coverage measures.
H5 — Local model deficiency	Unmodelled or drifted architectural surface predicts later churn or escape in the same subsystem.	Orphaned or drifted surfaces are no more likely than modelled ones to see later churn or escape.
H6 — Oversight amortization	As recurring failure classes convert into durable governance, human attention per unit of durable output declines — even if total governance work keeps growing.	Interventions per durable change stay flat or rise as effective conversion coverage grows.
H7 — Conversion conditions	Structural failures convert into shared governance faster and more durably when the capability to diagnose structure and the authority to change the environment are colocated, or connected by a working organizational pathway.	Split-authority or low-fit settings convert at the same rate and durability as high-fit, effective-authority ones.

Two of the seven carry conditions worth stating in the open. H3 does not claim hard controls never fail; it claims that a correctly encoded, adequately scoped mechanism is *less sensitive to throughput* than finite review, so the gap between them opens as the fleet speeds up. H6 does not promise the total oversight bill stops growing; it claims the *per-change* share falls as classes convert. Strip those conditions and both claims overreach; stated with them, each says only what the evidence can bear.

Corollaries and specializations

Several claims read more precisely as consequences or domain-specific instances of the seven than as coequal predictions. Naming them as such is not a demotion of their truth. It is a statement of where each sits in the causal structure; Table 6.0-5 places them.

Table 6.0-5: Corollaries and Specializations. *Former coequal predictions, restated as consequences or instances of the principal hypotheses.*

Former prediction	New status
Speed and stability do not trade	Corollary of H2 and H3
Conversion gets faster as tooling matures	Learning-curve specialization of H7
The substrate heals faster as it accretes controls	Consequence of H3 and H6
Grammar coverage beats line coverage	Domain-specific instance of H4
A fidelity gate reduces silent corruption	Product-specific instance of H3
Deployment pain stays bounded	Subjective consequence of H6; future work

The speed–stability claim, the one *Accelerate* made famous, now falls out of H2 and H3 together rather than standing alone: in a high-quality environment, pushing frequency up does not push the change-fail rate up. The two domain instances — grammar coverage driving out format escapes, a content-fidelity gate killing the silent-corruption class — are H4 and H3 read on one product's surface, valuable as grounding but not as separate laws.

6.0.7 Two directional relations

Two of the hypotheses have a form clean enough to write down. Neither is a fitted law. Each is a *shape* — a statement of which way a quantity moves and why, with the parameters named but not measured. Read them as the arithmetic behind the arrows, not as equations with coefficients this case supplied.

The churn relation

Durable throughput is raw velocity net of what churn eats:

$$V_{durable} = V_{raw} \cdot (1 - c)$$

where V_{raw} is the raw agentic implementation rate and c is the share of that effort consumed by churn. The substantive claim is about what moves c. As effective modeling coverage rises, more of the relevant surface fits the fleet's window before it churns, so over the under-governed range

$$\partial c / \partial E < 0$$

churn share falls as environment quality climbs. That is the arithmetic behind H2 and its subsystem sharpening in H5.

But the relation is not monotone across all governance accumulation, and the honesty of the theory turns on saying so. Push apparatus past the point of fit and churn starts to *rise* again — from collisions, false positives, latency, context burden, and constraints that block legitimate work. The qualitative shape has three bands, drawn in Figure 6.0-6: churn high in the under-governed region, lowest at fit, rising once more where the estate overgrows into a tower of governance.

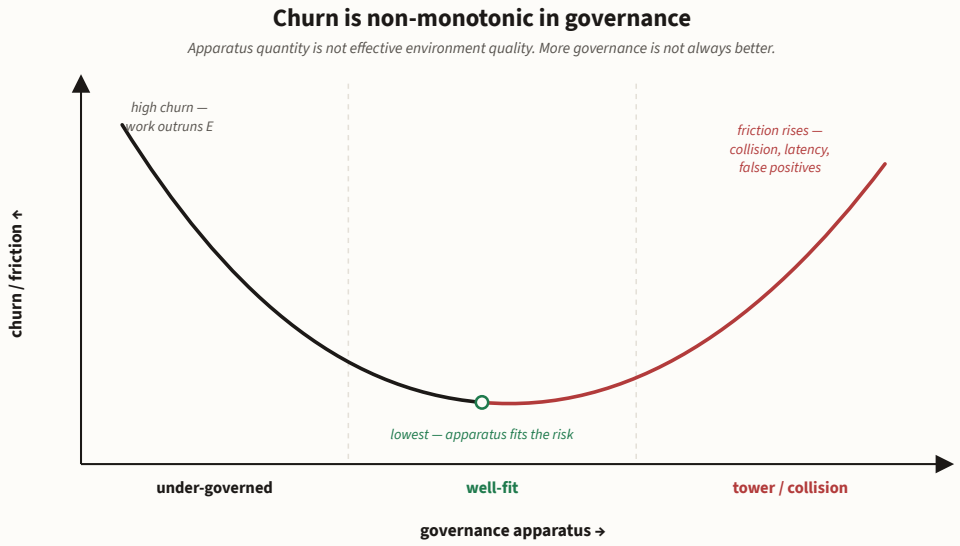


Figure 6.0-6: Churn is non-monotonic in governance. *Churn runs high when the environment is under-governed, reaches its lowest where the apparatus fits the risk, and rises again in the tower/collision region as controls duplicate and collide. Apparatus quantity is not effective environment quality; the curve is illustrative, not fitted.*

under-governed → **churn high** · well-fit → **churn lowest** · overgrown → **churn rises from friction**

No exact curve is claimed. The point is the non-monotonicity: apparatus quantity is not the same thing as effective environment quality, and a theory that assumed *more governance is always better* would miss the entire right-hand band.

Attention-based versus mechanized control

The second relation contrasts two ways to hold a failure class. Let p be the risk-weighted rate of defect opportunity, $r(V)$ the fraction a finite human review still catches at velocity V, and κ the risk-weighted fraction of relevant opportunities covered by effective mechanisms. Then the two escape rates are:

$$\varepsilon_{review} \approx p \cdot (1 - r(V)) \text{ and } \varepsilon_{control} \approx p \cdot (1 - \kappa)$$

Human attention is finite, so $r(V)$ falls as velocity rises — review catches a smaller share of a larger stream, and ε_{review} climbs toward the raw defect rate p. A mechanism applies to every change regardless of throughput, so κ does not fall with velocity in the same way. The claim is not that hard controls never fail. It is conditional and comparative: **for adequately scoped mechanisms, κ is less**

sensitive to throughput than review coverage r(V), so the daylight between "looked obeyed" and "was enforced" widens as the fleet speeds up. That is H3 in one line.

6.0.8 Evolution of this theory

The theory did not arrive whole. Its empirical core came first, induced from a bounded case; the rest is later reflection, and honesty requires keeping the two apart.

The first version appeared in the case-study paper *Cheap Code, Costly Judgment*[56]. That paper analyzed the first twelve weeks of the DocAble build and induced a five-step governance-conversion loop:

1. velocity exposes failure;
2. the engineer classifies the failure as local or structural;
3. structural failures are converted into governance;
4. later agents inherit a narrower, more explicit action space;
5. repeated conversions raise the environment's capacity to absorb future work.

That loop remains the empirical kernel. It contributes distinctions this chapter preserves: failure must be *interpreted*, not merely observed; governance works by reshaping subsequent action; conversion is recursive and does not terminate; capability-fit and authority condition whether a structural signal becomes shared infrastructure.

Continued development and roughly two further months of reflection extended the kernel in four ways.

- **Ex-ante governance became a first-class path.** The paper emphasized the delta of ex-post governance — controls discovered from failures no one could fully anticipate. Later work sharpened the complement: a disciplined organization or skilled engineer bootstraps a project with inherited architecture, standards, tests, prior incidents, and lessons carried from earlier systems. Conversion in one project becomes ex-ante knowledge in the next. Bootstrap and conversion are two loops, not one.
- **Modeling became a coequal route to governability.** The paper treated structured models as one governance mechanism among others. The Modeling Thesis gave them a distinct role. Models do not only constrain output; they compress intent and system structure into a representation the fleet reasons *through*, surface properties at the right semantic level, and support the drift gate that holds the map equal to the territory.
- **Governance accumulation proved non-monotonic.** The paper saw a large, layered substrate and described repeated conversion as compounding governability, already noting the tendency was not strictly monotone. Continued work supplied the missing negative side: mechanisms collide, duplicate, consume context, and block legitimate change. Mature adaptation therefore includes reconciliation and retirement, not only accumulation.
- **The process became visibly multi-loop.** The early theory centered one loop — failure, judgment, governance. The book revealed interacting loops: bootstrap establishes the initial environment, churn reveals what is missing, adaptation reshapes future work, and success expands ambition and concurrency onto a new frontier.

The result is still a governance-conversion theory at its core. It now describes MAGE as an adaptive system for governing the engineering environment itself.

[56] James C. Davis et al., "Cheap Code, Costly Judgment: A Case Study on Governable Agentic Software Engineering," 2026, https://arxiv.org/abs/2607.01087.

Epistemic note

The induced kernel and its later extensions are epistemically different, and the chapter should not blur them. The five-step loop was induced from the bounded case and its incident corpus. The four extensions are grounded in continued development, further observation, and integration across the book. Read the kernel as the case's direct yield and the extensions as refinements proposed for later testing — not as independently confirmed, population-level laws.

6.0.9 Research agenda

The seven hypotheses suggest a focused program, not a scatter of disconnected tests. Three study designs would cover most of the model, and none needs this repository; Table 6.0-6 maps each design onto the hypotheses it would test.

Table 6.0-6: The Research Agenda. *Three study designs and the principal hypotheses each would test.*

Study design	Principal hypotheses
Multi-project longitudinal panel with environment-quality variation	H2, H3, H4, H6
Subsystem-level panel tracking model gaps, churn, and escapes	H1, H5
Organizational comparison of diagnosis and authority pathways	H7

A particularly strong design would instrument a failure class from first occurrence through its whole life — recurrence, structural recognition, governance intervention, and later recurrence or escape. Such an instrument would test the discovery and adaptation loops directly, rather than leaning on aggregate repository metrics that cannot separate "the environment matured" from "the operator did."

6.0.10 Closing

The claim of MAGE is not that more governance produces better software. It is that abundant implementation capacity makes the fit of the engineering environment decisive. Known obligations can be encoded before the work begins. Unknown ones become visible only through churn, failure, and friction. Judgment separates the local defect from the missing structure; authority lets that diagnosis alter the shared environment; models and mechanisms reshape what the next agent can know and do. When the loop works, velocity buys durable progress. When it does not, velocity only reaches the churn wall sooner. That is the theory this book leaves for the next case to test.

6.1 Implications for Software Engineering

I built one system, alone, with a fleet of agents, and this book is what I learned from it. The shape of what happened is general enough that I want to step back from the particular product and ask what it implies for the discipline. If code is becoming cheap and judgment is becoming the scarce thing, then a number of assumptions the field has carried for decades were priced for a world that's going away. This section is about which ones.

Early in the book, Figure 1.5-1 argued that AI does not retire the software engineering lifecycle. It reassigns one seat. Requirements are still elicited, architectures still designed, software still validated, systems still maintained. The D moves to the fleet; the E stays with the engineer. The theory just developed says why that reassignment cuts deeper than a change of who types. The lifecycle stays recognizable. But the artifacts that support it, and the skills that perform it well, begin migrating into the governed environment itself.

That word carries the argument: **migration**, not replacement. The classical activities do not vanish. What moves is a set of engineering properties that used to live inside individual engineers. They migrate outward — into explicit representations, into executable mechanisms, and into the environment that holds both. The rest of this chapter, read against the theory, is an account of that migration; a later section gathers its five clearest forms.

The single mechanism under all of this is that **velocity exposes failure**. Agents produce change fast enough that the ambiguities, the missing abstractions, the weak boundaries, and the absent oracles surface in a matter of hours instead of quarters. That speed is, of course, a source of risk. But it is also a source of *perception*: related failures that would ordinarily arrive weeks apart now arrive inside a single afternoon, close enough together that you can see the structure they share. What you do with that perception is the whole game.

Everything the book taught to do with it comes down to the two theses. The Modeling Thesis: **use models to bind intent to implementation** — a typed picture the agent reasons over, held equal to the code by a gate, so the work fits in the context window before it churns. The Alignment Thesis: **add a governance mechanism** — a constraint or sensor the environment enforces, so a policy decided once holds against every change after, and confidently-wrong work is caught, not shipped. The preface named the single problem both theses answer: software has always wanted productivity to stay *linear* as you scale it, and where a human team hits Brooks's Law[57] and its N-squared coordination overhead, an agent fleet hits a different wall — churn, the loss of the thread as the work outgrows what the fleet can reason through and hold in view; its context window is the wall you engineer against. Both theses aim squarely at that wall. Models shrink what the agent must hold in-window, so the work fits before it churns. Governance keeps the agent from introducing those errors in the first place, and makes the ones that slip through visible. Read this way, the rest of this chapter is what those two theses imply once you believe them.

[57] Frederick P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Anniversary (Addison-Wesley, 1995).

6.1.1 From governing what you know to discovering what you don't

The tidy version of governance says: figure out your obligations in advance, encode them as mechanisms, and let the agents run inside the fence. Do that. It works, up to a point. For the accessibility product I had conformance checkers, content-preservation and provenance checks, and the ordinary commodity linters, all of them known before a single agent touched the code. All of them necessary, of course. But they were not sufficient, and the gap is the whole ballgame: the mechanisms you could not write down in advance, because you only learn them by failing.

Under agentic velocity, *many of the mechanisms you need cannot be written down before the work begins.* You discover them by failing. An agent makes a plausible, confident, wrong change; you look at it and ask not merely "is this acceptable?" but "does this failure reveal a hole in my governance?"; and when the answer is yes, you convert the hole into a durable mechanism. Governance, on this account, is a process the environment runs continuously — one that turns agent-produced failures into architecture and lints and gates that constrain the next agent. The environment *learns*.

This is why oversight, on its own, does not scale. The oversight model asks a human to supervise each change. At the volumes agents produce, no one reads the diffs, and a human who tries becomes the bottleneck the velocity was supposed to remove. The move that scales is conversion: convert a failure class *once*, and every future instance of it needs no inspection at all. Work that produced no failure signal gets admitted. Work that exposed a recurring failure class earns a new mechanism. The more classes you convert, the less vigilance you need.

6.1.2 Managing the workforce is not governing the environment

The volume is only half of why supervision fails. The other half is a difference in kind, and it is worth naming the school that difference defeats, because that school is the reasonable one. The natural response to a fast, fallible workforce is to reach for the practices that tamed fallible *human* teams: brief the worker, review its output, require a sign-off, keep an evidence trail — adapt the organization to the agent. Those practices work on people because a person's fallibility is bounded by two things the fleet does not have: an understanding that carries from one task to the next, and an accountability that makes a reviewer's judgment stick. An agent starts every task cold, holds a bounded window, and answers to nothing. Its unreliability is not a weaker grade of a junior engineer's; it differs in kind, not in degree, and the tools built for the human kind do not reach it. You cannot org-chart your way to trust in probabilistic output. The reliability the workforce lacks has to come from the environment instead: a model whose correctness a machine holds by construction and checks before the work is admitted. That is why this book argues for rebuilding the discipline around the environment that governs the agents, rather than offering a manual for absorbing them into an existing shop.

6.1.3 Soft mechanisms saturate; the substrate has to become machine-actionable

Here is a proposition I would put money on. **As velocity rises, tacit and review-centered mechanisms saturate, and deterministic ones do not.**

The reason is arithmetic. A convention is a probability flip. Flip it a thousand times a day and the one-in-a-hundred failure isn't rare — it's a schedule. A convention that an agent follows ninety-nine times in a hundred looks like a working mechanism when one engineer works at human speed; the faster the fleet runs, the sooner the hundredth violation arrives — many times a day, on the clock. Words are soft, and soft holds right up until the volume finds every edge you left in.

I have a concrete one. My cleanup routine had to distinguish a finished agent's abandoned worktree from a running agent's — and my rule for it lived in the instructions I wrote for each cleanup pass: "check that the agent isn't still in flight before you delete its workspace." It read as sound, and it failed three times in five days, each time destroying a live agent's work. The trap was timing: a freshly dispatched agent looks, for its first minutes, exactly like a finished one — a single bookkeeping commit and nothing else — so any check that reads the commit history at one instant misclassifies it. Rewording the instruction did not help; I reworded it three times. What ended the recurrence was moving the check out of the prose and into the tool: the dispatcher now writes a marker file when an agent boots and the tool deletes it only after the agent is truly done, and the cleanup *refuses* to touch any workspace whose marker is still present. The convention was correct and unenforceable; the marker is a fact the tool reads the same way every time, and it cannot be reworded away.

Learn more about this governance mechanism: agent registry.

So the governance has to move down to where a machine can enforce it: into types the compiler checks, schemas a validator reads, lints and analyses that gate a commit before a human ever sees it. None of this rejects the soft mechanisms, of course. The briefs, the templates, the prose conventions still aim the agents; they do real work. But they must be *mated* to deterministic mechanisms that hold the line whether or not the agent cooperates. The soft layer guides; the hard layer holds. At velocity you need both, and the failure mode of relying on the soft one alone is quiet: the convention looks obeyed right up until the day the volume finds the gap.

There is a counterintuitive corollary here that took me a while to trust: the stronger the guardrails, the faster the fleet can safely run. An agent that hits a deterministic guardrail gets an immediate, legible signal it can act on, and recovers from its own mistake without a human in the loop. The guardrail that would have felt like friction to a human team is, to an agent fleet, the thing that lets the throughput continue at all. Figure 6.1-1 draws the spectrum this section has argued, with the mechanism classes seated along it.

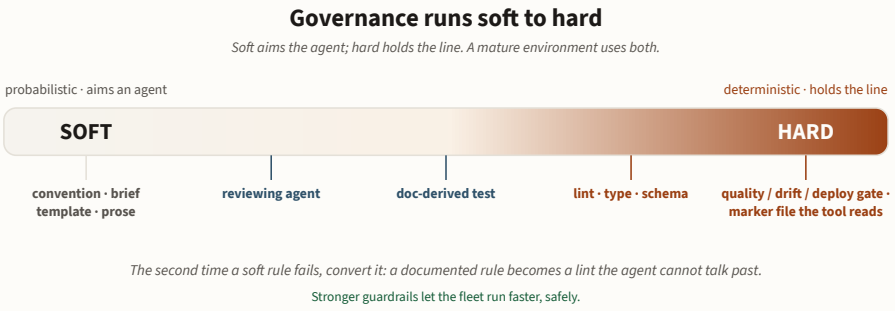


Figure 6.1-1: The Soft–Hard Spectrum. *Probabilistic mechanisms that aim an agent sit on the left; deterministic mechanisms that hold the line sit on the right. A mature environment uses both, and the skill is knowing which a given failure warrants.*

6.1.4 Three gains that usually trade against each other

Fred Brooks warned the field off silver bullets forty years ago. No single development — no language, no tool, no method — would give an order-of-magnitude gain in productivity, reliability, and simplicity at once, because the hard part of software is *essential*: the complexity of the problem itself, not the *accidental* complexity our notations and tools pile on top. You can shave the accidental part. The essential part does not yield to a trick, and anything sold as making it yield is snake oil.

Here is the claim, in that vocabulary and made carefully. Working at the model level, with an agent doing the work, gives you three things a team normally has to trade against one another. **Fewer tokens** — the agent reasons over a compact model instead of dragging a subsystem's worth of source through its window. **More velocity** — with the right picture in one window, the change lands faster and needs fewer round-trips. **Higher quality** — the same model carries invariants the agent checks its work against, so the change that lands is one a gate already vetted. Token reduction, speed, and quality do not come at each other's expense here. They arrive together, and roughly for free.

> *A footnote on silver bullets.* One might even call this the closest thing to a silver bullet the field has seen — Brooks's own name for the order-of-magnitude gain he argued could not exist. I propose it in his careful sense: not *a* silver bullet, since the essential difficulty never yields, but the sharpest tool I know for spending your effort on the essential part instead of drowning in the accidental one.

That is a large claim. Nothing here is something for nothing; the claim is that one move buys all three, and the reason is not luck. **A model is a distinct level of abstraction to work at.** It bridges the documentation and the code: accurate like the code, compact like the prose. An agent has a bounded window and reasons over whatever you put in it. Fill it with raw source and you spend tokens on detail that does not bear on the change, and the agent reasons over the wrong things. Fill it with the model and the join to the relevant code, and every token in the window earns its place. The gain lives in the representation: the agent, no smarter than before, is now reasoning over the *right* one. As long as the agent is good enough to work that representation faithfully, the models do the lifting — and because a human can read a model where a human cannot read the diffs, you keep review and control at the same time.

Now the Brooks-honest part, because this is where a lesser claim would overreach. The model level does not abolish essential complexity. Accessibility remediation across four document formats, each with its own tag model and conformance spec, is *genuinely* hard, and no model makes it easy. What the model level does is attack the *representation the agent reasons over* — which is exactly the accidental complexity Brooks told us was the only kind a tool can touch. It strips the accidental cost of holding a large system in a small window, and it leaves the essential difficulty sitting in plain view where a human's judgment can meet it. Brooks was right that the essential part does not yield. And the accidental cost it strips — holding a large system in a small window — is precisely the one that limits an agent fleet. That is the Modeling Thesis paying off against churn: bind intent to implementation through a model, and the churn wall moves out of reach.

This reframes the field's last run at making models central. CASE and Model-Driven Architecture reached for the whole bullet — make the model the system and execute it — and stalled on economics, not principle: heavy models could not keep pace with fast-changing code, and the price fell only where mistakes were catastrophic. Agents change that price, so how far to model is worth reopening.

But Brooks holds at the far end — a model never makes the essential problem easy — which is why the claim above stays the careful one: closest thing to, never the thing itself.

6.1.5 The judgment moved, and it moved toward you

If implementation is abundant, the human's comparative advantage moves to the work that surrounds the code: framing the problem, discovering the abstraction, making the architectural call, and designing the governance under which fast code can be trusted. Better agents make this work *more* consequential, not less, because better agents raise your ambition and your concurrency, and the decisions about what to build and how to bound it are what determine whether all that throughput becomes durable progress or an expensive pile of churn.

Two things gate whether this works — the conditions under which the whole method holds or fails.

- **Capability-fit.** The agents have to be matched to human judgment good enough to interpret their failures. Give a capable fleet to someone who cannot tell a local defect from a structural one and you get motion without direction. The governance does not design itself; a person has to look at a failure and decide it is the symptom of a missing abstraction. In my case the model itself was unreliable at drawing that line, and only began proposing sensible structural fixes once the codebase had accumulated enough worked examples to imitate.
- **Authority.** The person who sees the recurring failure has to be able to *change the environment* — the architecture, the mechanisms, the gates. This sounds trivial when one engineer owns the whole system. It is not trivial in an organization, where authority is split across teams, owners, review boards, and deployment pipelines. Split the authority and the same failure signal produces a local patch instead of a shared mechanism, and the class never dies. Agentic tools compress team-scale coordination into a single workflow; that compression is exactly what makes authority the pressure point.

6.1.6 What this changes for how we measure, staff, and teach

A few consequences follow, and I will keep them short because each could be its own argument.

Measure governed throughput, not volume. Commits landed, lines changed, tasks completed — these count activity. They do not tell you whether the activity became durable software. The question that separates progress from churn is whether agent-produced failures are getting converted into reusable governance, and whether later work benefits from those conversions without the whole thing losing coherence. Optimize for volume and you will get volume; you will also get a fast pile of slop with no one able to read it. I have the commits to prove that too.

The marginal cost of quality dropped, so treat quality work as affordable. The refactors and architectural improvements that a conventional team defers because they are too expensive become feasible when an agent fleet can carry out the mechanical part in a day. I moved an entire front end to a typed language over a weekend and saw no defects under the project's own checks. The same speed that manufactures technical debt can also pay it down — and, because related failures cluster in time, can make the architectural gap *visible* in the first place. Speed cuts both ways; which way depends on whether you convert what it exposes.

Read model failure as a framing problem before a capability ceiling. When an agent fails a delegated task, the tempting inference is that the model cannot do it. Sometimes that is true. Far

more often, in my experience, the task simply had not been framed as an inspectable, controllable process — the difficulty was mine, not the model's. The working disposition I ended up with was to assume that *I* was the problem: to treat a failure as a cue to decompose further and constrain harder, not as evidence of a ceiling. There is a paradox worth sitting with here. The model is valuable *because* it reasons over open-ended inputs, and yet the method is largely about systematically removing its opportunities to reason unconstrained. The paradox resolves once you remember what the machine is: a probabilistic coin flipped at every step. You bound the probabilistic behavior rather than exclude it — soft guidance going in, deterministic guardrails on the actions, and a deterministic envelope on what gets admitted out.

For research and education, attend to the environment, not only the agent. We already know agents can generate. The open question is which engineering environments make that generation *governable* — and whether agents can be taught to notice inadequate governance themselves, rather than waiting for a human to convert every failure by hand. For teaching, the implication is uncomfortable but clear: the comparative advantage is shifting away from syntax-level production and toward specification, abstraction design, validation, and process. If that sounds like a softer curriculum, I assure you it is a harder one — and it is the one that will matter.

6.1.7 What migrates into the environment

The theory gives the chapter's scattered observations a single shape. Five engineering properties, each once resident in a person, move into the environment. Naming them together is worthwhile, because each maps onto a dimension of environment quality or a loop the theory already drew, and placing this chapter after the theory is what lets the map be read.

Representation engineering becomes a discipline. The book has called judgment the scarce resource. True, and incomplete — it leaves the reader nothing to practice. Name the skill instead. The scarce activity is the design, synchronization, and governance of explicit representations: the structured models, specifications, invariants, and synchronization mechanisms that decide what a future agent can know and safely do. "Get good at judgment" states a disposition. "Get good at representation engineering" states a curriculum. The section that follows develops it as the language every mature engineering discipline already speaks.

Repository quality becomes representational quality. Classical software engineering read repository quality through code: complexity, modularity, coupling, maintainability. Those still matter. The theory adds a dimension they miss. Once the environment synthesizes each agent's context from the repository's own representations, quality comes to depend on whether those representations agree. Call the failure **representational entropy**: the same engineering reality encoded many times over — in terminology, in models, in prose, in prompts, in specs — with the encodings drifted out of step. This generalizes DRY. The old rule banned duplicated *implementation*. The new one bans duplicated *meaning* left unsynchronized. A stale doc, a conflicting prompt, a model that no longer matches the code: each raises the entropy and starves the context the environment can assemble.

This seems to collide with the beautiful garden below, where redundant encoding is a *strength* — one fact grown in many beds, surviving the loss of any single artifact. Synchronization is the distinction. Redundant copies held equal by a drift gate give resilience: they agree, so any one of them serves. Redundant copies left to drift give entropy: they disagree, so none of them can be trusted. DRY-for-meaning does not forbid the many beds. It forbids the many beds growing different plants.

Coherence, one of the theory's four dimensions of environment quality, is precisely the property that separates the two.

Engineering environments accumulate experience. Organizations used to improve mainly because their people did. An engineer met a failure, learned from it, carried the lesson to the next task. The knowledge lived in heads and left when they did. Governance conversion moves the lesson elsewhere. A recurring failure becomes a model or a mechanism, and the environment holds it for every later agent. The environment grows more experienced, and its experience does not walk out the door.

This meets a settled idea and sharpens it. Distributed cognition established decades ago that knowing is spread across people, tools, and artifacts rather than sealed in one mind[58] research on organizational learning and knowledge transfer showed how a firm's memory can outlast its members[59]. The migration is not that cognition is distributed — that much is old. It is that an executable governance mechanism becomes an active participant in the distributed system. A lint or a gate does not merely record a lesson; it enforces the lesson, on every change, with no one having to remember it.

Reasoning is compiled into the environment. As representations improve, more engineering reasoning gets performed once, captured, and reused. Dynamic Context Injection is the worked case: the environment maps the files an agent is about to touch to the exact constraints that govern them, then hands over that slice, so the agent inherits the relevant structure instead of reconstructing it from raw source Appendix B. The relationships, the models, the metadata perform a portion of the reasoning before the agent starts work. The reasoning is not gone. It is *amortized* — done by the environment, drawn down by every later interaction. The fitting word is **compiled**, and it rhymes with governance conversion: an insight, turned once into reusable structure, spent many times.

Capability becomes an environmental property. The purple loop of the theory carries the most provocative implication, so it earns the most care. Effective capability may come to depend not only on how strong the frontier model is, but on how well the environment is engineered. Picture two organizations. One runs slightly stronger models over an ordinary repository: sparse metadata, weak synchronization, little explicit structure. The other runs slightly weaker models over a repository whose semantics are richly modeled, kept in sync, and used to synthesize each task's context. The old intuition backs the first; it holds the more capable intelligence. The theory raises a different possibility. The second may win, because it has engineered the more capable *environment*.

State this as exactly what it is. It is an implication of the theory, not a result this single case established — a hypothesis the book hands to the next study, phrased the way Table 6.0-4 phrases the rest. Repository quality *may* substitute, at the margin, for model capability; rich representations let the environment perform some of the reasoning before the agent begins. Whether the trade holds, and how far, is the kind of whole-environment question the empirical program below is built to ask.

6.1.8 Models as the universal language of engineering

Here is the claim the whole book has been climbing toward. **Models are the universal language of engineering.** Every mature engineering discipline reasons about the thing it builds through a

[58] Edwin Hutchins, *Cognition in the Wild* (MIT Press, 1995).

[59] James G. March, "Exploration and Exploitation in Organizational Learning," *Organization Science* 2, no. 1 (1991): 71–87; Linda Argote and Paul Ingram, "Knowledge Transfer: A Basis for Competitive Advantage in Firms," *Organizational Behavior and Human Decision Processes* 82, no. 1 (2000): 150–69.

model rather than through the artifact itself — think of the load diagram, the circuit schematic, the control loop, the thermodynamic cycle. The two theses of this book are what finally let software join them. The Modeling Thesis makes the model the working representation the fleet reasons over; the Alignment Thesis holds it to the code with a gate. Together they turn "the model is the language" from a slogan into a checked practice: a picture accurate enough to trust *and* cheap enough to keep.

To say why software could not speak this language before, I need a distinction older than software. Aristotle separated a thing's **essential** properties, the ones that make it what it is, from its **accidental** ones, the ones it merely happens to have. Fred Brooks carried that distinction into our field in *No Silver Bullet*[60]: the essential difficulty of software is the complexity of the problem itself, and the accidental difficulty is everything our notations and tools pile on top. Brooks's warning was that no trick abolishes the essential part. His quieter point was that we spent most of our effort on the accidental part anyway, because we had no choice.

That is the historical claim I want to make loudly. Software engineering was forced to spend its best effort on the profession's *accidents*. We built decades of tooling to fight syntax, builds, dependency hell, and the friction of getting a change from a head to a running system — accidental complexity, all of it, however necessary. Agile itself belongs on that ledger. Its velocity-over-ceremony stance was, at bottom, an *accidental*-complexity concession: you shipped fast and kept models thin[61] **because** holding quality and a current model honest by hand was too expensive to afford at pace. The map drifted, and the discipline made a virtue of not drawing it. All of that spending on the accidents starved the essential — the system's real properties, its correctness, its design — of the attention it deserved. Agents change the arithmetic. They absorb the accidental work: they type the code, keep the map in sync, and hold the line for cents. For the first time, the essential part can get the attention.

That relocation does not shrink the engineer's responsibility; it moves where the responsibility is exercised. An engineer never had to hand-implement every choice to be accountable for it. Choosing a compiler with a known performance-and-defect profile is the old example: you did not write the code generator, and you were still answerable for the code it produced — for picking a tool whose behavior you understood well enough to stand behind. That was always the shape of engineering judgment, delegation with accountability intact. Agents extend the same arrangement to the whole system. You delegate the keystrokes and remain responsible all the way down to the code and beyond it — to the system's properties, the ones a model names and a gate checks. The locus of accountability did not move. The ratio of thinking to typing did.

And this is where the drift gate has to be kept honest, because it is the reason the residual moved *up* rather than away. A drift gate proves the model and the code agree; it does not prove either is right (the model can be a mirror, not a spec). Authoring the model that says what agreement should *mean* — which properties it asserts, which "ought"s belong in it — is the part that does not automate. So the essential residual Brooks warned would not yield is still here, and it has a new address: not "write correct code," but "author the model that says what correct means, and decide what belongs in it." The career does not disappear. It relocates upward, to the level where the properties live.

The last turn is the optimistic one. Modeling used to be architect-only work — the one person who could hold the whole system in their head drew the map, and everyone else worked below it, implementing against a picture they did not author. That division was a cost artifact. The map was expensive to draw and expensive to keep, so few people drew it. Freed from the cost of *typing* and

[60] Frederick P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer* 20, no. 4 (1987): 10–19.

[61] Kent Beck et al., "Manifesto for Agile Software Development," 2001, https://agilemanifesto.org/.

of *keeping the map in sync by hand*, every engineer — not only the architect — now works at the properties level: reasoning about the system's shape, and about how their own change moves the whole. The map is cheap enough that everyone can hold one. Modeling democratizes. That is the job the engineer's work becomes — fluent authorship in the language every engineering discipline has always spoken, now finally cheap enough for software to speak it too.

6.1.9 Empirical research opportunities

Every claim in this book rests on a single deep case: one repository, one method, roughly two years. That is a strength — a case seen at this depth exposes mechanisms a survey cannot — and it is a standing invitation. This section names three things the case could **not yet measure** and turns each into a research agenda: how to **govern a fallible oracle** when the test suite is the thing an agent optimizes against; how to **price a governed environment** when the very governance that makes it work also erases the signal a naïve benchmark would read; and how the field should **reason from deep single cases** to general claims. None of the three is a defeat. Each is a place where the case study has run ahead of the measurements the discipline currently knows how to take. It closes by widening those three case-level gaps into the measurement regime the whole field is entering.

Governing the oracle: fallible test suites as a research frontier

Throughout this book the test suite, the lint, and the conformance gate appear as *controls* — the deterministic machinery that converts a probabilistic generator's output into something an engineer can trust. But a control is only as good as its model of "correct," and a test suite is a **fallible model of requirement-satisfaction**. It can be too narrow (rejecting a correct fix that happens to differ from the reference implementation), too wide (accepting an incorrect fix because the requirement was under-specified), contaminated (the answer leaked into the model's training), stale (the world moved and the suite did not), or implementation-coupled (the oracle ratifies whatever the code already does). When the party being evaluated is an agent that *optimizes against the visible suite*, every one of these failure modes becomes an attack surface. This is the research frontier the book's governance stance runs directly into, and it deserves its own agenda.

The concern is not new, and stating its lineage first guards against reading it as an LLM-era novelty. A decade before the current benchmark controversy, Smith, Barr, Le Goues, and Brun established the canonical result for automated program repair: patches that pass the tests used to *drive* repair frequently fail an *independent* held-out suite, and "for programs that pass most tests, the tools are as likely to break tests as to fix them"[62]. Automation that optimizes against a visible oracle produces artifacts that satisfy that oracle while silently breaking untested behavior. The green bar was never the requirement; it was a proxy, and the proxy overfits under optimization pressure.

The same failure has now surfaced at the center of the field's most-watched benchmark. In early 2026 OpenAI — a *steward* of SWE-bench Verified, not an outside critic — announced it would stop treating the benchmark as a measure of frontier coding capability[63]. Auditing 138 problems a strong model consistently failed, they found **59.4% contained flawed test cases**: tests either too narrow (enforcing a specific implementation) or too wide (testing behavior the problem never described).

[62] Edward K. Smith et al., "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)," special issue, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015, 532–43, https://doi.org/10.1145/2786805.2786825.

[63] OpenAI, "Why SWE-Bench Verified No Longer Measures Frontier Coding Capabilities," OpenAI, February 2026, https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/.

They further documented **contamination** — frontier models could reproduce exact gold patches and verbatim problem text, evidence of training-time exposure — and concluded that "improvements on SWE-bench Verified no longer reflect meaningful improvements in models' real-world software development abilities. Instead, they increasingly reflect how much the model was exposed to the benchmark at training time." The steward is *governing the oracle by retiring it*.

The academic record corroborates the mechanism and separates its strands. Liang, Garg, and Zilouchian Moghaddam show that state-of-the-art models identify buggy file paths *from the issue description alone, without repository access*, at up to 76% accuracy on SWE-bench-Verified but only ~53% on tasks drawn from repositories outside the benchmark — a memorization signature, not reasoning[64]. Scale AI's SWE-Bench Pro, contamination-resistant by construction (copyleft and private codebases used as a legal deterrent against training inclusion), sees frontier scores collapse from the saturated seventies to roughly 23%[65]. And the governance response — treating the benchmark as a *living model that must be continuously re-derived* — is already visible: SWE-bench-Live continuously sources fresh tasks from post-2024 GitHub issues with per-task reproducible images, "grounded in live repository activity" precisely to defeat staleness and contamination[66] SWE-rebench automates decontaminated collection and flags contaminated instances[67].

A distinct strand matters specifically because *agents now write the oracles*. Hora and Robbes find that across 2,168 repositories, coding-agent commits add mocks to tests at a markedly higher rate than human commits (36% vs 26%) — mocks make a test "easier to generate automatically, but less effective at validating real interactions"[68]. A companion result shows LLMs tend to generate oracles that encode the *implemented* behavior rather than the *specified* one, so the oracle silently ratifies the code's bugs[69]. Superficial (mocked-away) and mis-aligned (implementation-coupled) are two different ways for an agent-authored green suite to mean nothing.

The load-bearing framing — and the guardrail. None of this says testing is useless or that richer oracle design eliminates the problem. That over-claim is false and the section must not make it. What the evidence supports is narrower and sharper: *a passing suite is a fallible model of correctness that must itself be governed.* And this is exactly the move the method in this book makes by reflex. Pin the invariant, not the example — fix a fuzz-exposed bug to the stable point in the format spec, so the fix passes every spec-allowed input rather than the one failing seed. Take an **obligation census** — enumerate what *must* hold, and treat a missing obligation as a finding, not a silence. Reach for the *strategy* that catches the *class*: property-based tests, fuzzing against a stable spec point, state-machine coverage, content-preservation checks — not one more example-shaped unit test. Distrust the self-report: an agent's "tests pass" is a claim about intent, not reality; re-run the gates at HEAD and re-derive completion from the artifact and multiple evidence sources rather than reading it off a

[64] Shanchao Liang et al., "The SWE-Bench Illusion: When State-of-the-Art Llms Remember Instead of Reason," in "Proceedings of the IEEE/ACM 48th International Conference on Software Engineering, Software Engineering in Practice (ICSE-Seip)," special issue, *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering, Software Engineering in Practice (ICSE-SEIP)*, 2026, https://doi.org/10.1145/3786583.3786882.

[65] Scale AI, "SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?," 2025, https://arxiv.org/abs/2509.16941.

[66] Linghao Zhang et al., "SWE-Bench Goes Live!," 2025, https://arxiv.org/abs/2505.23419.

[67] "SWE-Rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents," 2025, https://arxiv.org/abs/2505.20411.

[68] Andre Hora and Romain Robbes, "Are Coding Agents Generating over-Mocked Tests? An Empirical Study," 2026, https://arxiv.org/abs/2602.00409.

[69] "Do Llms Generate Test Oracles That Capture the Actual or the Expected Program Behaviour?," 2024, https://arxiv.org/abs/2410.21136.

green bar. The SWE-bench episode is the field discovering, at benchmark scale, a discipline this case study already practices at commit scale. **The oracle is fallible but governable** — and *how* to govern it (obligation census, contamination controls, live re-derivation, distrust of agent-authored oracles) is an open, fundable research program, not a solved problem.

Research opportunities this thread opens:

- **Obligation-completeness metrics.** What fraction of a task's requirements are *pinned by some control* versus merely *asserted by an example*? A repository-level "obligation census coverage" would measure exactly what SWE-bench's flawed 59.4% lacked.
- **Contamination- and overfitting-resistant evaluation as standing infrastructure**, not one-off benchmark refreshes — the SWE-bench-Live / SWE-rebench direction generalized.
- **Detecting agent-authored oracle degradation** (over-mocking, implementation-coupling) as a mechanical control in the review path, the way this book's lints catch structural defects.

The beautiful garden: why per-task ablation cannot price a governed environment

The most instructive negative result in this case study is one where the measurement *failed to find an effect* — and the failure is the finding. It deserves a full account, because it is the cleanest available demonstration of a claim the fallible-oracle discussion above only gestures at: that the metrics the field currently reaches for **cannot see the value of a governed environment**, and that the discipline therefore needs new measurands.

The setup. Over roughly two weeks we ran a method-ablation benchmark — internally, MAGE — over software-engineering tasks (root-cause analysis, code navigation, planning, design, review, model-evolution), *not* over the product's document-remediation tasks. The component under ablation was the **MBSE models bridge**: the typed models — state machines, the component/zone model, the invariant registry with its verification-tier taxonomy — through which a *context-bounded* agent reasons in order to operate a *context-exceeding* codebase. The per-task question was posed as a clean ablation. Give one agent the model map (arm A0, "guided"); give an otherwise-identical agent a codebase where the model file has been replaced by a 58-line stub (arm A1, "stripped"). On a task whose answer requires the map, does the guided arm win? The dependent variable was **recall**: which invariants and which `file::symbol` sites must change.

The null. After correcting a broken first instrument (the initial run put both arms at a 40/40 ceiling — zero discriminative headroom, an instrument defect, not evidence about the model), the first *informative* result came from re-scoring already-collected answers against a corrected, hand-adjudicated key — no new spend. The floor lifted in every cell, and **no cell admitted an effect.** The guided arm did not reliably beat the stripped arm. In the closest cell the delta reached only +0.20 of a required +0.25; in one cell the *stripped* arm slightly beat the guided one. Even the harshly-stripped arm recovered the key invariants — `INV-RC-4`, `INV-21`, `INV-4` — *by their identifiers.* The full per-cell result appears in Table 6.1-1.

Table 6.1-1: The Null Result. *Per-cell recall for the guided (A0) and stripped (A1) arms of the MBSE-map ablation — pilot-scale, three repeats. No cell clears the required +0.25 delta; in one cell the stripped arm runs ahead.*

Cell (model × strip-scope, R=3)	Guided (A0) recall	Stripped (A1) recall	Δ recall	Verdict
opus × harsh	0.867	0.667	**+0.200**	reject (no delta; closest, needs +0.25)

Cell (model × strip-scope, R=3)	Guided (A0) recall	Stripped (A1) recall	Δ recall	Verdict
sonnet × harsh	1.000	0.867	+0.133	reject (no headroom)
opus × light	0.600	0.800	**−0.200**	reject (no headroom; stripped *beats* guided)
sonnet × light	0.933	0.867	+0.067	reject (no headroom)

Why the null is forced — "delta ≈ 0 by construction." This is the load-bearing argument, and it is not a story about the model being worthless. It is a story about the *interaction between the dependent variable and the governed repository.* The method's discipline is that **the model is never the sole encoding of its own content.** Every invariant is named by identifier in a test, named again in the state-machine checker that walks it, restated in a rationale-comment at each enforcement site, and restated once more in prose in the architecture docs and the project's governing instructions. The map's content has been *propagated into the territory* — the same fact grows in many beds. This redundancy is not incidental; **it is the governance method working exactly as designed.**

That redundancy forces a structural dilemma. For any navigation-genre task, exactly one of three cases holds:

1. **The answer is recoverable from code and docs** → *no delta.* The stripped arm reads one of the propagated copies. In a governed repository this is the common case.
2. **The answer is recoverable only from the model artifact** → the delta is *1 by construction.* You removed the sole source; this is a tautology, not a measurement.
3. **The non-degenerate middle** → both arms can derive the answer, but the map changes the *cost* of deriving it. This is an **efficiency** claim, not a recall claim.

Navigation lands overwhelmingly in case 1; case 2 is unfalsifiable by design; only case 3 carries a real per-task signal — and it is not a recall signal. Stated exactly: in a garden-governed repo the stripped arm's mean recall tends to the ceiling, so the guided-minus-stripped recall difference tends to zero **regardless of how much the map is worth**, because recall can only see *find / don't-find* and redundant encoding erases that distinction. The null is produced by the instrument meeting the garden, not by the model lacking value.

The mechanism was caught in the act. In a single traced run both arms scored perfectly; the read traces show the guided arm opening the model file *first* and reasoning over the graph, while the harshly-stripped arm greps, hits the stub, and then **falls back to the code mirror of the same state machine** and reconstructs the identical closure. The finding, stated for the record: *the model is a navigational **convenience**, not a correctness **necessity.*** A change of genre did not escape the garden either — a task whose ground truth is *computed* from the model by exact set arithmetic over the exhaustive-search engine (removing the hand-drawn adjudication boundary entirely) returned the same null, because at small graph sizes the derivation is capability-saturated: both arms simply compute the small closure.

The over-read fence. This result was over-read twice inside the research loop, so the boundary is stated plainly: the null does **not** say the map is worthless. The map's value claims are *system-level* — fix-once-benefits-all, drift gates, tier derivation, checker mandates, the whole context-bounded-

agent-operating-a-context-exceeding-codebase bridge — and *none of those is a per-task recall claim.* A per-task recall null cannot touch them. The entire repository is the case study for the method's system-level value; the ablation was only ever meant to isolate the map's *per-task* contribution, and it found that in a well-cultivated garden that contribution cannot be isolated from the map's own propagated copies.

> The redundancy that flattens the ablation is *consistent with* the governance method achieving its stated goal: a repository in which the model's content survives the loss of any single artifact because it has been propagated into code, tests, comments, and docs. This cannot show that either representation is the "right" one — only that they agree. Map ≡ territory: the ablation cannot find a gap precisely because the governance has left no gap to find.

The positive agenda — three things to measure next. The null is not the end of the inquiry; it is a specification for the instruments the field is missing. The MAGE result is a concrete instance of a gap the surrounding literature names abstractly: *the predominant evaluation framework measures individual-task performance (pass rate on benchmark tasks) and does not capture the fraction of failure classes that become mechanically prevented, or the cost-per-functionality-unit of sustained agent-mediated work.* The garden null turns that abstraction into a research program:

- **(a) Efficiency at matched recall — the near-term measurable.** Case 3′s cost signal — *output tokens (and error rate, and wandering) once recall is held equal* — is the one per-task measurand the garden leaves open for navigation, and in this study it is **not measured**: the output-token field was null in every collected run (a token-capture-at-collection fix that had not landed). This is precisely a "cost-per-functionality-unit" metric. It is the obvious next measurement.
- **(b) System-level ablations — measuring the map's *actual* claims.** Fix-once-benefits-all, drift-gate catch rate, tier-derivation correctness, checker-mandate coverage: these are what the model bridge actually buys, and per-task recall cannot touch any of them. Measuring them ties directly to the open question of *which properties of a development environment predict whether agentic velocity is sustainable over weeks and months* — i.e. **governed throughput**, not implementation volume.
- **(c) The redundancy itself as a measurable quantity.** "The beautiful garden" is, operationally, *the degree to which the model's content is recoverable from code, tests, and docs.* That is a measurable property — call it **redundant-encoding density** or **agent-legibility** — and a higher value plausibly predicts resilience to the loss of any single artifact. This reframes a construct-validity observation (that documentation is a probabilistically-executable agent constraint, blurring the code/documentation line) as a *quantity to measure* and a candidate environment-quality metric.

The design lesson. Synthetic per-task ablation *fights* the tautology the garden creates: any answer available to the guided arm is, by construction, propagated into the code the stripped arm keeps. The complement is **real-task measurement** — a natural, in-production A/B in which some dispatches carry the model guidance and some do not, stamping the guidance condition and the token cost on real navigation and RCA work. This is the more promising path: the removal is natural, not artificial, so the garden cannot silently restore the answer. In this study that path exists but is not yet delivering — a broken join (the outcome writer stopped stamping the dispatch identifier, leaving ~92% of outcomes unjoinable to their guidance condition) has left it null and underpowered, and its repair is owned by a separate effort.

The register for all of this is deliberately modest. This is one repository's garden, pilot-scale (three repeats), with one unmeasured measurand. It is **analytic** generalization — evidence about a *mechanism* — not statistical generalization to a population. That is the right claim for a deep single case, and the single-case methodology discussion below makes the methodological case for why it is a claim worth making.

The methodology of deep single-case evidence

A reasonable reader reaches this point with a methodological objection: *this is all one repository.* The objection is correct, and meeting it squarely is itself a contribution — because the AI-augmented engineering literature is currently split between two evidence forms that answer different questions, and conflating them is the field's most common mistake.

Two evidence forms, two questions. On one side are the **build reports**: an organization stands up a large agent-driven effort and reports the outcome. These are proliferating and they are genuinely informative about *feasibility and scale*. Cloudflare rebuilt Next.js on Vite — an effort it calls vinext — in about a week, one engineer working with an AI model, and published the account[70]. Carlini describes building a C compiler with a team of roughly sixteen parallel Claude agents: a ~100,000-line Rust compiler that builds Linux 6.9, an account that exposes the coordination surface at scale[71]. Anthropic reports a million-line migration of its Bun toolchain from Zig to Rust, carried out through large dynamic agent workflows[72]. What build reports are *best at showing* is that a thing is **possible** — that agent-driven development reaches a scale skeptics doubted. What they *cannot* show is **mechanism**: which controls were load-bearing, which failures recurred and why, what the environment had to contain before the velocity became sustainable. A build report is an existence proof; it is not a controlled account of *why* it worked.

On the other side is the **deep single case** — this book. What it is best at showing is exactly what build reports cannot: the *mechanism* by which a governed environment sustains agent-mediated work over time. Seen at two years' depth, the case exposes the audit→fix→lint→meta-lint ladder, the moment a failure class converts into a deterministic control, the redundant-encoding density that flattens an ablation, the support apparatus that grows to several times the production code and functions as a substitute for code review. None of that is visible in a snapshot. The cost of the deep single case is obvious — $n = 1$, one toolchain — and the book states it plainly. The compensating strength is that a mechanism observed in enough detail supports **analytic generalization**: a reader can judge whether the *conditions* under which the mechanism operates hold in their own setting, and transfer the mechanism accordingly. That is a different and complementary logic from the statistical generalization a large-*N* study offers, and neither substitutes for the other. The companion "cheap code, costly judgment" account of this same repository tells that mechanism story at commit scale[73].

Table 6.1-2 sets the three evidence forms side by side.

Table 6.1-2: The Three Evidence Forms. *For agentic software engineering — what each form is best at showing, what it is blind to, and representative examples.*

[70] Steve Faulkner, "How We Rebuilt Next.js with AI in One Week," Cloudflare, February 24, 2026, https://blog.cloudflare.com/vinext/.

[71] Nicholas Carlini, "Building a C Compiler with a Team of Parallel Claudes," Anthropic, February 5, 2026, https://www.anthropic.com/engineering/building-c-compiler.

[72] Anthropic, "How Anthropic Runs Large-Scale Code Migrations with Claude Code," Anthropic, July 16, 2026, https://claude.com/blog/ai-code-migration.

[73] Davis et al., "Cheap Code, Costly Judgment: A Case Study on Governable Agentic Software Engineering".

Evidence form	Best at showing	Blind to	Examples
Build report	*Feasibility / scale* — agent-driven development reaches this size	*Mechanism* — which controls were load-bearing, which failures recurred, what the environment needed	Cloudflare vinext; Anthropic C-compiler (Carlini) & Bun migration; ~16-agent / ~100K-line / Linux 6.9
Deep single case	*Mechanism* — how a governed environment sustains agent work over time; which failure classes convert to controls	*Population-scale generalization* — one repo, one toolchain, one method	This book; and the "cheap code, costly judgment" account of the same repository
Controlled experiment	*Effect size* — does treatment X change outcome Y, holding others fixed	*Ecological validity* — the governed-repo tautology confounds per-task strips (the garden null above)	The MAGE ablation (the garden null above); the proposed real-task in-production A/B

The three rows are complements, not competitors. The build reports establish that the phenomenon is real and large; the deep case explains the mechanism; the controlled experiment — where it can escape the confounds the garden null above documents — measures effect sizes. A mature research program on agentic engineering environments needs all three, and the field's current over-reliance on individual-task benchmarks (the fallible-oracle frontier above) is a symptom of reaching for the third while neglecting the first two.

Inset — Methodological foundations for reasoning from a single deep case

The logic of learning from a single, richly-documented case is well-developed in the empirical software-engineering and social-science methods literatures, and this book stands on it explicitly. Runeson and Höst's guidelines for case-study research in software engineering supply the reporting and validity framework — design, preparation, collection, analysis, reporting, and the explicit treatment of construct, internal, and external validity threats[74]. The move from a single case to a *mechanism* claim — rather than a population estimate — is **analytic generalization**, and its most developed articulation is process-tracing: Beach and Pedersen's within-case, mechanism-focused inference, which distinguishes theory-building from theory-testing uses of a case and grounds the claim that a mechanism observed in one case can be transferred where its scope conditions hold[75]. The register is Merton's **middle-range theory**: not a grand law, not raw description, but a mechanism bounded by stated conditions — the correct altitude for the claims in this book.

[74] Per Runeson and Martin Höst, "Guidelines for Conducting and Reporting Case Study Research in Software Engineering," *Empirical Software Engineering* 14, no. 2 (2009): 131–64, https://doi.org/10.1007/s10664-008-9102-8.

[75] Derek Beach and Rasmus Brun Pedersen, *Process-Tracing Methods: Foundations and Guidelines*, 2nd ed. (University of Michigan Press, 2019).

This framing also disciplines the honesty of the garden null above. The garden null is *analytic-generalization evidence*: it identifies a **mechanism** (redundant encoding flattens per-task recall ablation) and the **scope condition** under which it operates (a repository governed to propagate its models into code, tests, and docs). It is explicitly *not* a claim about how often that mechanism obtains across the population of software projects — that is the empirical question the field's build reports and controlled experiments must jointly answer. Read this way, the book's central methodological propositions are falsifiable predictions a reader can test in their own setting: that higher agentic velocity surfaces structural failure classes sooner; that review-based quality regimes show rising escaped-defect rates where control-based regimes do not; that teams converting failures into deterministic controls sustain velocity where inspection-reliant teams do not. Each is a proposition the deep case *articulates* and that a larger study could *test*.

The methodological research opportunity, stated directly: the field needs a shared vocabulary for *which evidence form answers which question*, and a norm against citing a build report as if it established mechanism, or a single deep case as if it established prevalence. The three-row table above is a first cut. Sharpening it — and building the controlled and longitudinal instruments that can escape the governed-repo confound the garden null above documents — is the empirical research the book most wants to provoke.

The new measurement regime

Set this case's unmeasured threads against the field they belong to, because the difficulty is not local to one repository. For decades, empirical software engineering has tried to measure interventions whose effects were modest beside the variation in who does the work. Three confounds swamped the rest: the skill of the engineer, the context they work in, and the intelligence they bring. Development processes, review practices, architectural styles almost certainly matter, yet their signal has been hard to pull from that noise. The field spent decades at this and rarely showed an effect much bigger than the variation. That was a signal-to-noise problem, not a failure of rigor.

GenAI is now commoditizing intelligence, and that shifts what a study can hold fixed. A coding agent's intelligence has become a roughly uniform, purchasable input: buy the same model and every condition starts from about the same cognitive baseline. One of the three confounds can now be pinned. Hold the intelligence constant and vary the rest — the environment, the representations, the governance mechanisms — and a researcher may run many experiments with far more control than the field has ever had. One caveat carries real weight: this holds only for the kind of intelligence GenAI offers, not for intelligence in general. Human genius still varies as it always has. But much of software work now flows through agents whose intelligence a study can fix as a variable, and with that confound fixed, the others may finally come into view.

That commoditization is not a forecast. It is the current empirical record. A systematic review gathers nearly four hundred AI4SE studies across dozens of software tasks[76], and the controlled evidence beneath it runs one way. A field experiment across three firms and 4,867 developers measured a 26% rise in completed tasks[77] an earlier randomized trial clocked a single coding task 55.8% faster with an

[76] Xinyi Hou et al., "Large Language Models for Software Engineering: A Systematic Literature Review," *ACM Transactions on Software Engineering and Methodology* 33, no. 8 (2024), https://doi.org/10.1145/3695988.

[77] Zheyuan (Kevin) Cui et al., "The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers," *Management Science*, ahead of print, 2025, https://doi.org/10.1287/mnsc.2025.00535.

AI assistant[78]. The gains are real and often modest, and sometimes they reverse: sixteen experienced open-source developers, working in repositories they knew well, ran 19% slower with early-2025 tooling even while they believed themselves faster[79]. That last result sharpens the point rather than denting it. Commoditization means the input is available on demand, not that it always helps. What the field still argues about is the size of the gain and the conditions that govern it. Even the sharpest benchmark fights grant the model as a usable input and quarrel only over how far contamination inflates the scores[80]. A confound you can buy off the shelf, and dispute only in magnitude, is a confound a study can hold fixed. The order-of-magnitude MAGE proposes is a bet that the multiplier lives in the environment, not in the model's marginal point.

Agentic software engineering may change the scale of the intervention. The question stops being whether one process lifts productivity a few percent. It becomes why one engineering environment lets a single engineer govern hundreds of successful agent contributions while another stalls at a handful. The precise magnitude is not the claim — order-of-magnitude differences are a hypothesis the program should test, never a measured law. The claim is that the intervention may now produce effects large enough to see plainly.

Large effects earn their keep twice. They are worth more economically, and they are worth more *scientifically*. An effect that dwarfs the background variation is easier to detect, easier to reproduce across sites, and far likelier to support a mechanism-oriented theory than one that differs only marginally from organizational noise. If agentic engineering routinely produced such effects, the discipline would enter a regime where cumulative, replicable findings come within reach.

The object of study changes with the scale. Classical work mostly varied people and processes, which are emergent and hard to hold fixed. Agentic engineering lets a researcher vary *engineered* artifacts: representations, governance mechanisms, synchronization strategies, retrieval architectures, whole environments. These are built on purpose rather than grown by accident, and that shifts the discipline a step from behavioral science toward engineering science.

This optimism has to be squared with the negative result reported above, and squaring them sharpens both. The beautiful garden showed that a *per-task ablation* cannot price a governed environment: strip one model artifact and the garden's redundant, synchronized copies restore the answer, so the recall delta collapses toward zero no matter what the map is worth. That is a claim about the **micro** scale — one task, one artifact removed. The new regime's large effects live at the **macro** scale — whole environments set against whole environments, the two-organizations contrast run for real. The two results cohere, and tightly. The effects the field can now hope to see clearly are the whole-environment differences that per-task ablation is built to miss. So the garden null is not a rebuttal of the optimistic vision; it is the vision's complement. It marks where *not* to measure — inside one governed repository, one task at a time — so that the measurement lands where the signal is: across environments, on the outcomes the whole apparatus moves. Micro-ablation erases the effect by construction. Macro-comparison is where the same effect turns large and reproducible.

MAGE is one proposal for organizing this regime, not the regime itself. Other methodologies, architectures, and governance approaches will emerge, and the opportunity outlives any of them. The claim

[78] Sida Peng et al., "The Impact of AI on Developer Productivity: Evidence from Github Copilot," 2023, https://arxiv.org/abs/2302.06590.

[79] Joel Becker et al., "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity," 2025, https://arxiv.org/abs/2507.09089.

[80] OpenAI, "Why SWE-Bench Verified No Longer Measures Frontier Coding Capabilities"; Liang et al., "The SWE-Bench Illusion: When State-of-the-Art Llms Remember Instead of Reason".

worth making is the broad one: agentic software engineering may create interventions whose effects are large enough, engineered enough, and reproducible enough to carry cumulative, mechanism-oriented empirical science.

Whether MAGE proves correct matters less than whether the field takes up the opportunity. If it does, empirical software engineering gains a new class of phenomena to study — effects large enough to observe, explicit enough to reproduce, structured enough to build theory on. The invitation is not to adopt one method. It is to treat agentic software engineering as an experimental science whose theories can be proposed, compared, refined, and, on evidence, replaced.

6.1.10 Limitations

I should be plain about the limits of this book. My evidence is one system, authored by me, built inside one vendor's agent ecosystem. It is deployed in production and in use by me and my colleagues at Purdue University, and every claim here stays scoped to that case. But one deep case is not a lonely one. Engineers working with agents have been converging, independently and in numbers, on the moves this book teaches: heavy testing, specification-driven development, an environment that enforces its own rules. Those reports corroborate that the practices work. What they leave open is the mechanism — how the moves compose, and why they hold against a fleet. Answering that takes a system you control end to end, with every layer in view. That is the evidence this case supplies. I built it as an instrument of inquiry as much as a product: its quality had to matter for the evidence to, and I treated every failure as data about the method rather than noise to clean up.

There is a sharper limit worth naming, and this book is my own evidence for it. I wrote it following the method it describes: the book has structured models of its own structure, and a drift gate holds those models to the prose, the same discipline I turned on the code. An LLM assisted with the first draft. But the final version will be entirely mine — every sentence rewritten in my own voice — and that rewrite is not a formality: it marks where the method stops.

Writing for other people is human-to-human communication, and voice is part of the message. An LLM cannot imitate my voice, and it does not seem right to ask you to read my ideas at one remove, mediated by a machine that is not me. So the method governs the *models, the structure, the drafting* of a book. It does not supply the voice. That part stays human.

Code is different, and the difference is the crux. Code is human-to-computer communication, and its properties are analytical all the way down — correctness, performance, security. There is no voice to preserve. That is precisely why governed autonomy transfers cleanly to code, and to other analytical, verifiable artifacts, while it stops short of the voice-bearing work of writing for other people. The method reaches exactly as far as the analytical, and no farther.

If you try this on a codebase of your own, I want to hear how it goes — where it held, where it broke, where your setting behaved differently from mine. One system is one entry in that larger conversation, and the shape gets sharper as more people run the experiment and report back. Come tell me what you found.

6.2 Conclusion

> I applied mine heart to know, and to search, and to seek out wisdom, and the reason of things.
>
> — King Solomon (Ecclesiastes 7:25)

I set out to learn what engineering becomes when implementation is nearly free. Here is what I found: the code got cheap, and the judgment got expensive. That's the whole book in a sentence. The rest of it is me trying to say, concretely, what that expensive judgment *is* — and how you organize a system so a fast, unreliable machine can exercise the cheap part without wrecking the rest.

The method reduces to one loop, repeated until the system holds. Velocity surfaces a failure. Judgment classifies it — a local bug to patch, or the symptom of a missing mechanism. The structural ones you convert into something durable: a type that makes the error unrepresentable, a lint that catches it at commit, a gate that blocks it before production, a model the next agent reasons through. Then you go again. Each conversion narrows the space the next agent can fail in — the more conversions you land, the more the governability compounds. The apparatus that results is large — in the system this book is drawn from, the tests and lints and tooling outweighed the product code — and it is not overhead: the apparatus is the review, as Part 5 (A MAGE Case Study) showed. It is what you buy the velocity with.

> The future of software development is not faster implementation. It is maturation into an engineering discipline. Paradoxically, fast and unreliable agents give us the opportunity to build software more cheaply and to a higher degree of quality than ever before.

Let the machinery hold what your attention cannot, and spend your scarce judgment on the decisions the machinery cannot make.

Two things stay firmly with the human, and neither is going anywhere. The first is vision: deciding what should exist, what "correct" means for this problem, and which failures deserve a real fix versus a redesign. The second is abstraction: seeing the shape the code is groping toward and naming it, so the machine has a model to build against instead of a blank space to guess into. The agent supplies tireless implementation. You supply direction and structure. Where you supply neither, it will happily build you a hundred thousand lines that nobody needs and that nobody can read. I have the commits to prove it.

I would not have troubled to write a book if I thought that the MAGE methodology would change with the next generation of AI models. Consider my central claim: that modeling offers a suitable level of abstraction at which designers and implementors can dialogue. Modeling gives the designer the ability to reason about the correctness and cost conditions of the system. Modeling lends the implementor a blueprint to bring to life. Modeling allows an auditor to compare the vision against the reality. If this sounds obvious, well, of course it is. This has been the claim of all engineering disciplines for all time. But it has also been the weakness of software engineering among the engineering disciplines. Unique among engineers, we have labored with the expectation that change is constant. That has hitherto meant that maintaining models was only reasonable for contexts where changelessness could be enforced by physical laws or long-running government contracts. Now, although change is, if anything, still more constant, the MAGE methodology invites us to adopt models because they are at last the most cost-effective way of doing business. For software engineering, models are

an idea whose time has come — and our discipline can at last reason the way the older engineering fields always have.

There is a particular word here for the modeling community. The preface set *Model-Driven Software Engineering in Practice* on the shelf as this book's lineage, and that tradition had the destination right; it stalled on the cost of the road. Its own standard text concedes both halves: models and code must co-evolve, and the consistency between them goes unenforced, because no team could staff the reconciliation. The tradition was missing only an economical laborer. The agent fleet is that laborer. It re-derives and re-checks a model against the code on every change, for cents, under gates that make the sync a build property instead of a discipline. And the old bet now pays twice: the model the fleet keeps true is also the model that solves the fleet's context problem, so the upkeep the field could not afford has become the representation the workforce needs anyway. One system is thin evidence for a field's revival, and I keep the claim scoped to it. But what buried the tradition was economics, not its ideas, and the economics have changed.

Strip the book to three moves and this is what remains.

1. **Architect deliberately:** implementation is nearly free, so spend the saving on the shape — the structured model, the boundary, the abstraction the machine builds against — rather than on faster slop.
2. **Convert failure into machinery:** when velocity surfaces a structural failure, do not just patch it; turn it into a type, a lint, a gate, a sensor that catches the whole class the next time.
3. **Keep judgment scarce and central:** the machine searches; you decide what is worth searching for. Guard those decisions. Hand the mechanical work to the machinery. And spend yourself only where nothing else can. Three moves, one loop, run until the system holds.

One last framing. Take the premise of *Software Engineering at Google* — mechanize the discipline — push it to a workforce that cannot remember, and you are forced to encode everything explicitly. Take the form of the Gang of Four — the design pattern — and you have the unit in which to encode it and pass it on. The **governed engineering environment** is what you get when you do both: a system where fast, forgetful, capable machines produce trustworthy software because the discipline lives in the environment, not in anyone's head, and it lives there as a catalogue of named, reusable patterns. And once keeping the map equal to the territory costs almost nothing, the map itself fights churn by steering the fleet: the Modeling Thesis supplies the map, the Alignment Thesis enforces it, and together they are the method this book set out to name — Model-Based Agentic Software Engineering, run until velocity holds.

6.2.1 The part that stays yours

We opened by calling the agent a 3D printer, not a stapler. Let us revisit the metaphor in closing. A 3D printer builds almost anything you describe, and *only* what you describe. The theses are how you tame it. A model gives the machine the right thing to build against instead of a blank space to guess into; a constraint holds the part it prints inside the envelope you decided on. We cannot make our agents deterministic. We bound our probabilistic fabricators until what comes out is something we can trust.

So do not fear the machine, and do not overtrust it either. It is the most capable tool most of us will ever hold: it gives a skilled engineer the throughput of a team, and it does that precisely because it will do anything you ask, without the friction that used to separate ideas from reality. The governed

engineering environment is how you put the friction back where it belongs — on the failures, not on the human — so the throughput lands as a production system instead of a fast pile of slop.

If there is a single instruction to carry out of this book, it is this. Govern the conditions under which fast code can be trusted, and know when to stop governing: put a guardrail on the smells that predict a real failure, but remember that guardrails tax future changes. The machine can search faster than any of us. It cannot tell us what is worth searching for. That part is still yours, and it is the part worth getting good at.

Machines can help us search. They cannot give us judgment.

6.2.2 Where to go next

You do not have to adopt all of this at once, and you should not. I suggest you pick one recurring failure your agents keep handing you, and convert it — one type, one lint, one gate. That single conversion is the loop in miniature, and the method grows from there.

When you want more, the appendices are the working reference. Apply them to the one failure in front of you, reach for the catalogue when you hit the next, and watch your governed engineering environment grow.

I hope the MAGE method serves your work as well as it has served mine.

6.3 About the Author

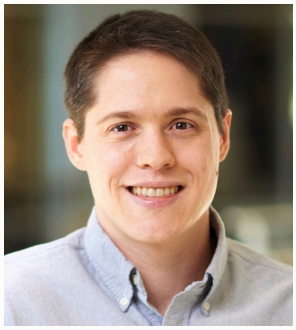


Figure 6.3-1: James C. Davis.

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.

Dr. Davis came to the professoriate by way of industry. He spent five years 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 record for most defects opened in a year. He earned his PhD from Virginia Tech in 2020. The methodology in this book thus rests on sixteen years of software engineering experience that straddles the GenAI divide.

6.4 Colophon: Dogfooding MAGE

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.

6.4.1 The production, briefly

This book was set in Fraunces for display, Source Sans 3 for the body, and IBM Plex Mono for code, on a warm paper stock with a single burnt umber accent. There was no separate typesetting pass. The manuscript is one structured source — chapters, sections, figures, and citations — that a single build script projects two ways book/build_book_html.py: to HTML for the web edition, and to Typst for the print PDF, which `typst` then lays out. The web book and the print book are two views of one source, so they cannot drift apart. The whole thing is kept under version control, validated and built by one stdlib-only tool that runs on a fresh checkout catalog.py. References live in a BibTeX file book/references.bib, resolved through inline markers, with a gate that refuses to build on a stale citation. The figures are hand-drawn SVGs — no plotting library behind them — each carrying a title and a description for readers arriving by screen reader.

Those are the facts a colophon usually records. The more unusual one is that the book was maintained using the method it describes.

6.4.2 The manuscript became an engineered system

The argument no longer lived solely in prose. It lived in explicit models that could be inspected, evolved, and kept consistent as the manuscript changed. Alongside the chapters sat a small set of structured models — the book's case written out as data, its concepts named, its claims registered, its citations positioned — and a build step that held those models in agreement with the prose the same way the rest of the method holds things together.

This was not an accident of tooling. Earlier, the book argues that MAGE is not fundamentally a software methodology but a method for governing large evolving artifacts through explicit models and executable constraints — the general claim Table 6.0-4 frames as a prediction for the next study to test. The manuscript became a chance to test that claim directly. The artifact under construction was no longer software. It was a technical argument — a book — and the same discipline governed it.

6.4.3 The models that governed it

Each model carried one facet of the manuscript's structure, and a check held that facet consistent as the prose moved. Table 6.4-1 names each one with the failure it prevented. They are described here in the book's own vocabulary — explicit models, traceability, drift, synchronization, governance — because the point is recognition: this is the same method the preceding chapters taught, turned on the book itself.

Table 6.4-1: 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 case as an ordered run of numbered claims, with which chapters advance each	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
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 naming the prose that would negate 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 case studies as packages of catalogue entries, 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

None of this is the book; the prose is the book. The models gave the argument somewhere outside the prose to live, so a change in one place could be traced to its consequences in the others.

That traceability paid off most when a claim moved. Because the argument existed as an explicit model, a global inconsistency became visible at once instead of surfacing by accident during editing. Three architecture-view chapters were labeled in the spine as advancing both of the book's theses; a sensor that watches for a thesis spread across more chapters than it is genuinely taught in flagged the second thesis and named exactly the three chapters responsible. The fix was small, and the model made its reach exact — a bounded worklist of three openings to re-read, rather than a whole-book hunt. The machinery did not decide what the change should be. It made the reach of the change finite and visible, which is the smaller and more useful thing.

6.4.4 MAGE, applied outside software

Running the method on the manuscript followed the same loop the book teaches for code:

- **Ideas** arrived first, usually as speech in the author's own cadence.

- **Explicit models** captured the structure the ideas belonged to — the spine, the claims, the concepts.
- **A governed manuscript** held the prose and those models together under one build.
- **AI drafting** turned rough material into candidate text to argue with.
- **Consistency checks** flagged where a model and its prose had fallen out of agreement.
- **Human judgment** decided every revision, and rewrote every sentence into the author's voice.
- **A finished manuscript** emerged that a check could confirm was internally consistent.

The governed manuscript sat under the same declared-source-plus-generated-artifact shape the rest of the method uses, laid out in Table 6.4-2: a human authors the model, a build maintains its derived forms, and a drift check refuses to let the two disagree.

Table 6.4-2: The governed manuscript. *Each model under the declared-source-plus-generated-artifact shape: what a human authored, what the build maintained, and the drift check that held the two in agreement.*

Model	Human authored	Machine maintained	Drift checked
Argument-spine model	Declared claims and per-chapter labels	Generated spine and summaries	✓ over-spread sensor
Chapter-shape model	Per-chapter opening/closing assessments	Derived reports	✓
Claims model	Declared propositions and audit predicates	Generated claims register	✓
Literature-positioning model	Declared interventions and frames	Generated positioning artifact	✓
Substantiation ledger	Data bindings, by hand	Aggregated per-claim backing reports	✓ under-substantiation query
Design tokens	Declared faces, accent, and scale	Projected CSS, Typst, and SVG palette	✓ token-drift lint

Because structured models absorbed the structure, the manuscript could take repeated large-scale rewrites without collapsing into inconsistency. That is the workflow the earlier chapters describe for a codebase, run on a book — the same loop the theory draws for code in Figure 6.0-1, now shown to scale past software to any artifact whose internal consistency has to hold.

6.4.5 What the language models did

Large language models were part of making this book, and it would be coy not to say so. They helped find adjacent research, proposed candidate citations, compared the argument against neighboring work, drafted figure concepts and accessible descriptions, and turned rough material into drafts worth arguing with. The lasting lessons are engineering ones, not the fact of the tool:

- **AI accelerated exploration** — it surfaced where a literature already existed and turned thought into candidate prose quickly.
- **Structured models constrained that exploration** — the spine, the claims, and the concept map kept generated text answerable to a declared shape.
- **Human judgment owned the architecture** — the argument, the evidence, and every acceptance or refusal of a revision stayed on the human side.

- **Every factual claim was independently verified** — a model will produce plausible bibliographic detail that is wrong, so every citation, quotation, and number was checked against its primary source before it was allowed to stay.

6.4.6 Limits, and human judgment

The machinery had a firm boundary, and it is worth stating without hedging. It could trace a claim through the book; it could not decide whether the claim was true. It could hold two representations in agreement; it could not turn that agreement into correctness. **A consistency check can show that the model and the prose agree. It cannot show that either is right.**

The method reached the manuscript's models, its structure, and its drafting. It did not supply the voice. Writing for other people is human-to-human communication, and voice is part of the message — so every sentence was rewritten by hand, and the judgments that mattered stayed human: choosing the argument, deciding what counted as evidence and where the evidence stopped, judging when a claim had grown too strong, and standing behind the finished book. The models made those judgments easier to carry across hundreds of pages. They did not make them.

Earlier, the book argued that MAGE governs complex evolving artifacts through explicit models and executable mechanisms, and that the claim was not confined to code. This manuscript became the first non-software artifact developed under that method. The software system described in these pages was built with MAGE. The book describing that system was, in turn, written with MAGE — which is the book's larger claim made concrete: MAGE is not merely a software engineering methodology, but a method for engineering any sufficiently complex artifact whose internal consistency matters.

Appendix A

MAGE Engineering Stacks

Appendix A — MAGE Engineering Stacks

Stacks: mechanisms that travel together

A single pattern in the preceding appendices kills one failure class. In practice, though, mechanisms arrive in *clusters* — a concept you want to adopt (model-based engineering, a self-operating orchestrator, an auditable format seam) is not one mechanism but several that reinforce each other. This appendix names those clusters. Each **stack** attaches to a concept, lists the mechanisms that make it up, and says which of them you can leave out.

A stack composes at a different grain than a `package` move. A **package** is composition *inside* one mechanism — a constraint shipped already welded to its own dedicated sensors, still one catalogue entry. A **stack** is composition *across* several distinct mechanisms — many entries that together make one governed capability. Both travel together; the package is the intra-mechanism weld, the stack the inter-mechanism cluster.

Every stack sorts its members into two kinds:

- **Mandatory** — the stack *fails* without this member. Model-based engineering needs both the typed models AND the drift control that keeps them equal to the code; adopt the models alone and you ship a map the fleet will trust while it quietly lies. A self-operating orchestrator needs its work-templates. These are the members you cannot skip without breaking the concept.
- **Complementary** — layers on top for extra value, not required for correctness. Dynamic context-injection can sit on top of a semantic-lint stack to *prevent* the violation the lint already *catches* heartbeats sharpen an observability stack that already sees and responds. Worth adopting, but the stack stands without them.

Each member links to its own pattern page in the earlier appendices. Read a stack to see which mechanisms you must adopt as a set, and which you can add later.

Where every mechanism sits

One idea sits *above* the pattern set rather than beside it: **enforce at the right semantic level**. A control must fire at the level where the property actually lives (structural → deterministic check; semantic → model/judge) and be as legible as the failure it prevents — the placement judgment that explains WHERE every other mechanism sits.

It earns that place because it is a choice, not a pattern you reach for. Aim a check one level too low and it passes the violation it should catch while firing on the legal case it should allow — present, but wrong. Read the pages that follow with this lens: each pattern is as much a decision about *level* as about shape.

The nine capabilities

The mechanisms answer to nine capabilities a governed engineering environment needs. Each names a job the fleet must do; the patterns grouped under it are the shapes that do it. Nine peers are hard to hold, so they hang on the book's own triad of what governance acts on — the **models-bridge** the fleet reasons through, the **product** it ships, and the **agent** fleet that does the work. The stacks below and the reference that closes this appendix both sort into these nine.

Models-bridge — the map the fleet reasons through, held equal to reality.

- **Maintain authoritative system knowledge.** Represent intent + structure in typed models the fleet reasons through.
- **Keep representations equal to reality.** Reconcile model ↔ code/system; catch drift mechanically.

Product — the shipped artifact: constrained, complete, preserved, attributed.

- **Constrain where and how agents act.** Sanctioned mutation surfaces + closed action vocabularies + semantic policy.
- **Establish completion on re-derived evidence.** Recompute completion; derive assurance obligations from the model.
- **Preserve product semantics.** Guarantee the product's meaning survives mutation + conforms to spec.
- **Track provenance and trace causes.** Durable, complete, checkable attribution of every mutation and cause.

Agent — the fleet that produces work: admitted, managed, self-governed.

- **Admit or reject changes.** Gate the work order and the path to production.
- **Manage work, state, and resources.** Lifecycle records, resource mediation, and fleet observation.
- **Govern the control estate itself.** Model, cover, and encode the governance system as its own subject.

A note on vendor features

The mechanisms in this book are described against one concrete substrate — a coding agent and its harness — because a real system is what makes them legible. The mechanism is never the vendor feature, though; the vendor feature is one *instance* of it. A harness's lifecycle hooks, for example, realize a general idea: **enforcement points** on the agent's runtime lifecycle, where a deterministic step fires whether or not the agent cooperates. Your framework may expose that idea differently — a middleware layer, a git hook, a CI stage, a wrapper process — and the concept carries across intact.

Read every “the agent does X” in these appendices as *here is one way to realize the MAGE concept X*. The concept is the portable part, and it is what you are meant to take with you.

Appendix A - 1. The provenance + fidelity stack

A two-page synthesis of the provenance + fidelity stack. Five patterns make one guarantee: reconstruct a remediated document's mutation history from the artifact itself, and catch any damage done through the sanctioned door. Who and why for every change, and nothing silently lost.

The capability

Reconstruct the full mutation history of a shipped artifact from the artifact alone — and prove nothing the author wrote was dropped on the way out. Two capabilities converge here: *track provenance and trace causes*, and *preserve product semantics*. It assumes every change already flows through one sanctioned door; on that door it builds attribution you can read back and a fidelity check you cannot skip. The evidence lives inside the file, so it survives copy, download, and re-open — no side log to trust or lose.

When to adopt this stack

Use this stack when:

- you must reconstruct what a tool changed, and why, from the shipped artifact alone — with no side log to trust
- failures show up as lost history or unexplained changes that leave no durable trace
- auditors or downstream consumers require traceable, reversible attribution of every change
- tool-inserted content must stay distinguishable from what the author actually wrote
- a pass can silently drop content and the damaged output still looks fine until a reader hits the hole

Typical domains:

- regulated software
- document processing and remediation
- medical systems
- financial systems
- any auditable transformation pipeline

Failure classes it covers

- **The unattributable change.** A pass mutates a document but leaves no durable trace of who changed it or why; once the file leaves the pipeline the history is gone.
- **The indistinguishable insert.** A tool-inserted artifact looks exactly like authored content, so nothing can tell what the tool added — and a validator cannot cover what it cannot name.
- **The silent hole.** A new mutator lands without attribution wiring; "we record every change" quietly stops being true, one commit at a time.
- **The unread evidence.** Stamps sit in the artifact with no consumer, so "auditable" is a claim no one can exercise.
- **The silent loss.** A pass drops a table, a note, a paragraph; the file looks fixed but shipped damaged, and no one sees the hole until a reader hits it.

Composition

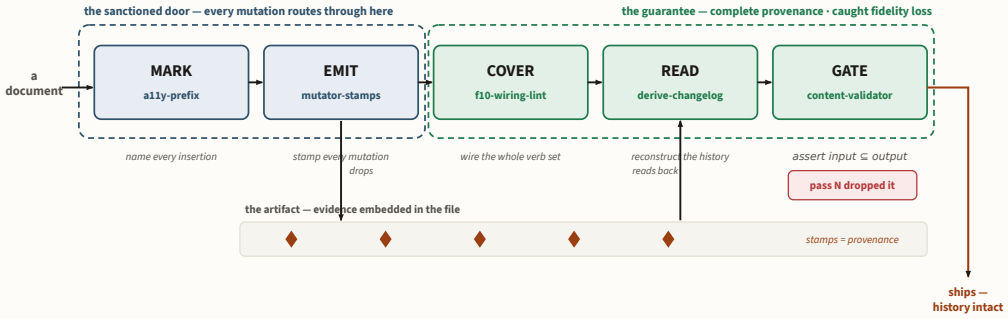


Figure 7.1-1: The provenance + fidelity stack in one picture. A document flows left to right through two lanes. The sanctioned door (fleet blue): MARK names every insertion so it is registry-covered; EMIT writes an attribution stamp for every mutation into the artifact. The guarantee (governed green): COVER's wiring lint holds the closed verb set at zero gaps; READ reconstructs the history from the embedded stamps; GATE asserts the input's content survives the output and names the pass that dropped it otherwise. Below the row, the artifact strip carries the stamps EMIT drops and READ and GATE read back. Mark it, cover the marking, read it back, and gate what leaves — provenance you can reconstruct from the artifact itself.

1. **MARK** `a11y_` prefix convention (§A.1.1)
2. **EMIT** Per-mutator attribution stamps (§A.1.2)
3. **COVER** F10 mutator-stamp-wiring lint (§A.1.3)
4. **READ** `derive-changelog` (reconstruct mutations) (§A.1.4)
5. **GATE** ContentValidator (input ⊆ output fidelity) (§A.1.5)

The five parts run as a chain: mark every insertion, stamp every mutation, prove the stamping complete, read the history back, gate what leaves against what came in. Each part hands the next a stronger guarantee.

The constituent parts

Five parts run as a chain, each handing the next a stronger guarantee: mark every insertion, stamp every mutation, prove the stamping complete, read the history back, and gate what leaves against what came in.

MARK — the reserved-prefix naming convention

Name every insertion. Every artifact the tool inserts gets a name marking it tool-added, so authored and inserted content stay distinguishable and a validator can cover an insert by its name alone. (MARK.)

Receives — the raw insertions a remediation pass wants to write: alt text, tags, off-canvas scaffolding. Nothing precedes it; this is where the chain starts.

Guarantees — a closed, registry-covered population of insertions. A three-way naming rule carries it: an invisible insert takes a reserved prefix, a user-visible one keeps an ordinary name, a spec-mandated name keeps its spec name. Every inserter records itself into one registry as it writes, so nothing enters the artifact unmarked.

Hands to EMIT — a complete set to attribute. Because the population is closed and named, the stamp-writer downstream has no blind spot: every insert it must stamp is already on the register. The naming is by rule, not by taste, which is what makes the coverage mechanical rather than a habit that quietly drifts the first time an inserter forgets to append.

→ **Deeper treatment:** `a11y_` prefix convention (online).

EMIT — per-mutator attribution stamps

Record every mutation. Each sanctioned change embeds its own attribution — its pass, its visibility — into the artifact at the mutation site, so the evidence travels with the file. (EMIT.)

Receives — the marked, registry-covered inserts from MARK, plus every other sanctioned mutation a pass performs.

Guarantees — embedded, attributable evidence, not a promise in a log. One sanctioned stamp-writer per format carries every stamp, and the raw stamp mutation is ban-linted away, so the writer is the sole surface. A visibility model keeps it honest for delivery: stamps default to Debug and are stripped before the file ships; user-visible passes opt into Preserved. The evidence lives in the document, not a side log, so it survives copy, download, and re-open.

Hands to COVER and READ — two things at once: a closed verb set for the wiring lint to prove complete, and an embedded stamp registry for the changelog to read back.

→ **Deeper treatment:** Appendix B - 28. Per-mutator attribution stamps.

COVER — the mutator-stamp-wiring lint

Prove the record complete. A blocking lint scans every mutator verb in the document models' write layer and fails the build on any verb that skips the stamp wiring, so attribution has no silent gap. (COVER.)

Receives — the same closed verb set EMIT's stamp-writer serves; the lint reads the write layer where the mutators live.

Guarantees — zero open attribution gaps. The check asserts one property: every path that performs a guarded mutation calls the stamp routine on its way out. So a new verb cannot land producing unattributable mutations. It sees the absence a code reviewer's eye slides past — a missing call shouts nothing in a diff, but a completeness scan over all verbs catches it. The lint stays cheap because it checks so little: not what a mutator does, only whether the stamp call was made on the way out.

Hands to READ — a stamped population the changelog can trust. Because COVER proves the writer is called across the whole verb set, READ downstream reads the embedded stamps as complete, not a sample.

→ **Deeper treatment:** F10 mutator-stamp-wiring lint (online).

READ — reconstruct the changelog

Reconstruct the history. A command rebuilds the document's mutation history from its embedded stamps, each entry a change attributed to the pass that made it. (READ.)

Receives — the stamps EMIT wrote and COVER proved complete, read straight from the produced artifact. It runs after remediation, against the delivered file, so the history it assembles is the one that shipped.

Guarantees — an attributed, human-legible ChangeLog: pass → change, each entry carrying its visibility. It reconstructs the history from the artifact itself, never a trusted external log, so the account

is reproducible from any conformant file. A diff would show what changed; this shows who changed it and why — the questions RCA and user transparency actually ask. The account is only as complete as the stamps behind it, which is exactly what COVER upstream holds at zero gaps.

Hands off — nothing further in the chain. READ is the consumer that makes emitting the stamps worthwhile: no reader, no reason to stamp.

→ **Deeper treatment:** `derive-changelog` (reconstruct mutations) (online).

GATE — the input ⊆ output fidelity gate

Prove nothing was lost. Where MARK, EMIT, COVER, and READ attribute what the tool *added*, this gate catches what a sanctioned pass silently *removed* — damage done through the same door provenance covers. (GATE.)

Purpose — close the chain from the other side. Attribution accounts for every change the tool made; it says nothing about content the tool dropped. The gate covers that blind side.

Mechanism — a deterministic post-condition. It extracts the content of input and output and fails the job unless the input's content is a subset of the output's, and it runs in production on every job. A staging-only per-pass variant adds localization: a dedicated marker and a nonzero exit code name which pass dropped the content, turning "content was lost somewhere" into "pass N lost it."

Guarantee — no damaged output leaves. A dropped paragraph, a mangled table, a lost caption cannot ship unseen, the worst outcome for a fidelity-critical tool, because the output looks fine and quietly isn't what the author wrote. What ships is both fully attributed and provably whole.

→ **Deeper treatment:** Appendix B - 25. ContentValidator (input ⊆ output fidelity).

A DocAble example, end to end

DocAble remediates a slide deck for accessibility. A pass writes alt text onto an untagged image. **MARK** gives that description a reserved prefix and records it in the insertion registry, so it reads as tool-added, not author-written. **EMIT** stamps the mutation into the file — this pass, invisible insertion — through the one sanctioned stamp-writer for the format. **COVER** has already proven, at build time, that the alt-text verb wires that stamp; a version of the verb that forgot to would have reddened the gate before it shipped. Weeks later a customer asks what the tool changed: **READ** reconstructs the changelog straight from the deck's embedded stamps, no pipeline access needed. And when a different pass quietly drops a speaker-notes paragraph, **GATE** catches it — input content is not a subset of output — fails the job, and names the pass that lost it, instead of shipping a deck that looks fixed but is not.

Tradeoffs and adoption order

Adopt in chain order, because each part presumes the one before it.

1. **MARK first.** Until insertions are named and registered, nothing downstream has a complete population to attribute. A naming convention costs nothing at runtime.
2. **EMIT, then COVER.** Stamp through one writer, then add the wiring lint that holds the closed verb set at zero gaps. The stamps cost one write per mutation; the lint is deterministic and does not decay.
3. **READ** is a read-only projection — cheap, and it makes the stored provenance finally usable.

4. **GATE** last, and independently valuable: a set-containment check over extracted content, run in production. Its per-pass diagnostic variant stays staging-only to keep the production path fast.

The chain's guarantee is only as strong as the one door it presumes — every mutation must route through the sanctioned surface, held by a separate ban-lint. Extraction that under-reads a format weakens the gate; bound it to the canonical reader.

The full treatment

Every constituent above links to its full Gang-of-Four pattern — in this appendix for the flagship members, online for the rest. The stack composes with the model-coherence stack — the sanctioned door it stamps is that stack's typed seam — and the specification + verification stack. The complete 83-mechanism catalogue, each pattern with its Motivation, Applicability, and Known uses, is online in the web edition.

Appendix A - 2. The model-coherence stack

A two-page synthesis of the model-coherence stack. Six patterns let a context-bounded agent reason through a typed map of the system instead of the whole territory — and keep the map equal to the territory, so what the agent queries is what is true.

The capability

Give a bounded-context agent a typed map it can reason through, and hold that map equal to the code so the map never lies. It answers two capabilities at once — *maintain authoritative system knowledge* and *keep representations equal to reality*. The models are executable data, not prose — read live, checked against reality by a gate, derived where a derived edge cannot drift, and generated back into the system. An agent that cannot hold the whole codebase queries the map instead, and the map is trustworthy because machinery, not hope, keeps it current.

When to adopt this stack

Use this stack when:

- an agent cannot hold the whole system in context and must reason through a map of it instead
- structure lives in diagrams and prose a program cannot read, so it drifts the moment the code moves
- consumers copy facts out of a model into their own code and the copies diverge silently
- a complex file format is mutated from a hundred raw call sites with nowhere to encode its invariants
- you need a map the fleet can trust because machinery, not habit, keeps it equal to the code

Typical domains:

- large agent-collaborative codebases
- model-driven and MBSE engineering
- multi-service system topologies
- tools built over a complex shipped file format

Failure classes it covers

- **The prose that drifts.** Structure lives only in diagrams and sentences; a program cannot read it, so it falls out of step the moment the code moves and no one notices.
- **The stale snapshot.** A consumer copies a fact out of the model into its own code; the copy and the model diverge silently, and the consumer acts on a value the model no longer holds.
- **The map that lies.** A model row points at a thing that no longer exists, or a real thing on disk has no row, and the fleet reasons through a map that quietly disagrees with the code.
- **The broken join.** The link between a model and the code it governs lives in someone's head; the code moves, the link breaks invisibly, and no gate fires.
- **The hand-edited generated file.** Boilerplate the model implies is written by hand beside it; the two fall out of step and the generated-looking file is silently stale.
- **The hundred raw writes.** Mutation of a format happens through many raw library calls, so a malformed write can land from any of a hundred sites with nowhere to encode the format's invariants.

Composition

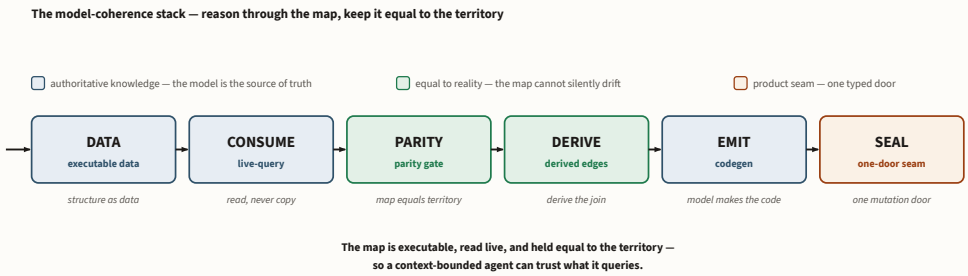


Figure 7.2-1: The model-coherence stack in one picture. Six parts run left to right in two capability lanes plus the sealed-format part. Authoritative knowledge (fleet blue): DATA models the system as executable typed data; CONSUME reads it live and never snapshots; EMIT generates artifacts back from the model. Equal to reality (governed green): PARITY fails the build when a model and reality disagree either way; DERIVE anchors every model-to-code edge on a resolvable symbol a lint re-checks, so a derived edge cannot drift. SEAL (accent) routes all mutation of a shipped format through one typed model held sole by a ban-lint. The map is executable, read live, held equal to the territory, and generated back into it.

1. **DATA** Executable source-of-truth models (§A.2.1)
2. **CONSUME** Meta-model consumption discipline (read, don't hardcode) (§A.2.2)
3. **PARITY** Drift & parity gates (§A.2.3)
4. **DERIVE** Symbol-anchored traceability graph (derived edges) (§A.2.4)
5. **EMIT** Model-driven codegen (§A.2.5)
6. **SEAL** PdfModel (sole PDF mutation surface) (§A.2.6)

The stack has two lanes. Three parts make the map authoritative — model it as data, read it live, generate from it. Two parts hold it equal to reality — a parity gate and derived edges. One part seals the same discipline onto a shipped product format.

The constituent parts

Six parts build the guarantee in rungs: model the system as executable data, read that data live instead of copying it, hold it equal to reality with a parity gate, raise parity to a derived graph, generate real artifacts from the model, and apply the same discipline to a shipped file format.

DATA — model the system as executable data

Make the system machine-readable. Model the system as executable data the tools import and run on, so a program reads the structure and catches it the moment it moves. (DATA.)

Receives — the system's own structure: its components, its state machines, its service topology, its registries. Nothing precedes it; this is the ground the rest stand on.

Guarantees — a machine-readable model a tool loads on every run and generates real artifacts from. A query returns the live fact where a stale sentence cannot. The structure lives as data, so a program can check it, not merely a reader who happens to look.

Hands to CONSUME — a typed object every part below stands on. The consumer queries it, the parity gate checks it against reality, the generator emits from it, the traceability graph anchors its edges to

it. The data is authoritative in name only until the parity gate three rungs down holds it equal to the code; on its own it is just well-typed documentation.

→ **Deeper treatment:** Appendix B - 17. Executable source-of-truth models.

CONSUME — read the model, don't copy it

Read the fact, never a copy of it. Each consumer resolves the fact it needs by querying the live model, so no second copy exists to fall out of date. (CONSUME.)

Receives — the typed model DATA published, and a consumer that needs one of its facts: a queue name, a component boundary, a policy value.

Guarantees — one authoritative value and no second copy to fall out of date. The consumer reads the model in place, so the fact it acts on is the fact the model holds now. A copy-detecting lint catches the one consumer that smuggles a constant back in.

Hands to PARITY — a single value to check, not a scatter of copies to reconcile. Because every consumer reads live, "source of truth" becomes true in practice rather than in aspiration, and the parity gate downstream has one authoritative value to hold against reality instead of a dozen drifting snapshots. This is the discipline that makes the DATA above it worth trusting.

→ **Deeper treatment:** Appendix B - 20. Meta-model consumption discipline (read, don't hardcode).

PARITY — fail the build when the map and territory disagree

Catch drift mechanically. A fleet of deterministic lints fails the build whenever a model and the reality it mirrors disagree, in either direction. (PARITY.)

Receives — the executable model and the reality it claims to mirror: the code, the artifacts, the things on disk it names.

Guarantees — bidirectional parity or a red gate. Every model row resolves to a real thing, and every real thing carries its row; a meta-sync contract names, per model, what reality it mirrors and when it must be re-derived. So the map cannot quietly lie while the fleet reasons through it.

Hands to DERIVE — trusted data the rest of the stack can build on. The gate converts "the model is probably right" into "the model is right or the build is red," so the executable data, its live consumers, and the emitted artifacts can all be trusted at once. Drop it and every part around it degrades into optimistic documentation.

→ **Deeper treatment:** Appendix B - 16. Drift & parity gates.

DERIVE — anchor every model-to-code edge to a symbol

Make every join refactor-proof. Anchor each model-to-code edge on a resolvable symbol a lint re-checks, so moving the code reddens the scan instead of silently breaking the link. (DERIVE.)

Receives — the parity-held models and the code symbols they connect to: the functions, the classes, the checks that give each edge a real endpoint.

Guarantees — edges that cannot drift. Each terminates on a resolvable symbol, never a line number, and is a derived obligation a lint re-checks at scan time. Move the code and the symbol either stays resolvable or the scan reddens; the broken link turns mechanically visible instead of rotting in someone's head.

Hands to EMIT — a graph of trustworthy connections to generate from. Where PARITY asserts a model matches reality, DERIVE computes the join, so those edges need no hand-authored parity rule at all — the higher rung. It leaves the generator standing on links that a refactor cannot silently break.

→ **Deeper treatment:** Appendix B - 22. Symbol-anchored traceability graph (derived edges).

EMIT — generate real artifacts from the model

Generate from the model, don't restate it. A generator emits real artifacts from the model — policy, wiring, catalogs, contract types — each carrying a provenance header. (EMIT.)

Receives — the parity-held, traceable model, consumed the way a compiler consumes a source file.

Guarantees — generated artifacts that cannot silently drift from the model. Each is emitted from the model, so the model drives the system rather than merely describing it. A provenance header names the emitter and the regen path, so a hand-edit to a generated file is caught on the next run and reverted.

Hands to SEAL — a model that now writes the territory, not just maps it. Because the artifact is derived, the parity gate treats it as generated output, not a second source to reconcile. What remains is to apply the same one-model discipline to a shipped file format, which the last part does on the product side.

→ **Deeper treatment:** Model-driven codegen (online).

SEAL — route a file format through one model

Give the format one mutation surface. Route every read and write of a complex file format through one structured model, with raw library access banned (our instance: a PDF model over the canonical PDF library). (SEAL.)

Purpose — apply the same one-model discipline to a shipped file format, where mutation would otherwise scatter across a hundred raw library calls with nowhere to encode the format's invariants.

Mechanism — one typed model is the sole door; a ban-lint forbids the raw library, so every change passes through code that encodes the format's invariants. Every mutation a remediation pass wants to make routes here rather than through a raw call site.

Guarantee — a single, compiler-checked mutation surface. A malformed write can no longer land from just anywhere, and a fix to an invariant holds everywhere at once. This is also the single door the provenance stack's stamp-writer needs to cover, so the two stacks meet at exactly this seam.

→ **Deeper treatment:** Appendix B - 24. PdfModel (sole PDF mutation surface).

A DocAble example, end to end

DocAble's PDF remediation is where SEAL earns its place. Every tag-tree read and write routes through one typed PDF model; a ban-lint forbids raw calls to the underlying library, so the format's invariants live in exactly one place and a fix holds across every call site at once. The surrounding lanes govern the wider system. The component-and-zone model, the job-lifecycle machines, the domain registries are all **DATA** — executable, not prose. Tools **CONSUME** them live: a check resolves "which service owns this seam" by querying the model, never by a copied constant. **PARITY** gates fail the build if a model row names a service that no longer exists, or a service ships with no row. **DERIVE** anchors each model-to-code edge on a symbol, so a refactor that moves the code reddens the scan instead of silently breaking the link. And **EMIT** regenerates catalogs and wiring from the models with provenance headers, so the generated files cannot quietly fall behind their source.

Tradeoffs and adoption order

1. **DATA and CONSUME are the floor.** Typed data plus read-don't-copy costs a query in place of a constant. Without them there is no model to keep honest.
2. **PARITY is mandatory, not optional.** The executable data is worth nothing without the gate that holds it equal to reality; drop it and every model degrades into optimistic documentation. Its cost is a gate per model.
3. **DERIVE raises the ceiling.** Where the gate *asserts* a match, derived edges make parity unnecessary for those joins — symbol-anchored edges survive refactors that line numbers would not.
4. **EMIT and SEAL are targeted.** Codegen pays off where the model implies boilerplate; the sealed format model pays off on a complex shipped format, at the cost of building the model and migrating every call site.

The whole stack leans on one presumption: consumers read live and mutations route through the sanctioned surface. A smuggled snapshot or an escaped raw call is where it weakens — each held by its own lint.

Choosing a rung for one fact — hold the join at the highest affordable rung

The stack's parts are also a **strength ladder for a single fact that lives on two surfaces.** Whenever the same fact is written in two places — a value in the model and a copy in the code, a spec and a sample, a status recorded once and read somewhere else — the two can fall out of step. The parts above say how firmly you can hold them together, strongest first:

- **UNIFY (strongest) — no second surface at all.** Keep the fact in exactly one place, as executable data (the **DATA** rung). There is no join to hold because there is no copy. The compiler, not a gate, keeps it honest.
- **CODEGEN — generate the second surface.** Where a second artifact must exist, emit it from the first (the **EMIT** rung), with a provenance header. The copy is regenerated, not hand-authored, so it cannot be edited into disagreement — a stray hand-edit is reverted on the next run.
- **DERIVE — compute the join.** Where both surfaces are genuinely authored, anchor the edge between them on a resolvable symbol a lint re-checks (the **DERIVE** rung). Move the code and the scan reddens; the join cannot break in silence.
- **PARITY — assert the join.** Where you cannot compute the edge, a deterministic lint asserts the two surfaces match and fails the build when they diverge (the **PARITY** rung). Both surfaces are hand-kept; the gate catches the drift after the fact.
- **comment (holds nothing).** A note that says "keep these in sync" is the bottom of the ladder. It records the obligation and enforces none of it — the two surfaces drift the first time someone touches one and not the other.

The rule: for each two-surface fact, hold the join at the highest rung the two boundaries afford — not the highest rung imaginable. Some facts cannot climb. A config value and its documented sample must exist as two real artifacts on two sides of a boundary; a model row and the code it governs are legitimately separate. Those honest non-climbs are sanctioned — you hold them at PARITY or DERIVE and stop. What the rule forbids is holding a join *below* its affordable rung: a fact synced by a comment that a parity gate could have caught, a hand-mirrored pair that codegen could have generated, two copies that agree only because nobody has touched either yet.

A join held below its affordable rung is a latent drift class, and a close-time audit treats it as a failure. The reflex to reach for, whenever a change puts the *second* copy of a fact on disk, is to ask which rung this pair can climb to and hold it there. The close-time review that keeps an effort honest (the *derived defends, snapshotted drifts* discipline behind the Definition-of-Done) makes the same check at the end: a fact left on a weaker rung than it afforded is flagged, not shipped. An audit of one such review pass found most of its drift instances were exactly this defect — a join a rung too low — rather than an outright missing check.

The everyday shape is a **status or summary field that is read to make a decision but written by hand.** Left as a hand-edited line, it drifts the moment real work moves past it and the next reader trusts a stale value. Derive it instead from the artifacts that already record the truth — a projection over the underlying files and history — and it cannot drift, because there is no second surface to fall behind. That is the ladder applied to one field: climb from a hand-kept copy (comment rung) to a derived projection (UNIFY/DERIVE) and the whole drift class closes.

The full treatment

Each constituent links to its full pattern — in this appendix for the flagship members, online for the rest. The stack is the substrate under the provenance + fidelity stack (its sanctioned door is this stack's sealed seam) and the specification + verification stack (whose state machines are DATA this stack keeps honest). The full 83-mechanism catalogue is online in the web edition.

Appendix A - 3. The specification + verification stack

A two-page synthesis of the specification + verification stack. Six patterns turn a concurrent behavioral spec into checked assurance: model the lifecycle as composed state machines, derive every obligation from it, discharge each at the rigor its shape demands, then map coverage back onto the model so an unexercised invariant is a visible gap, not a guess.

The capability

Say what "correct" means for a concurrent system, derive every check that correctness owes, and prove each one at the right rigor. Two capabilities run through it: *establish completion on re-derived evidence* and *constrain where and how agents act*. It models the lifecycle as state machines running at once, names the predicates that must hold across them, and routes each to a checker by its shape — an exhaustive proof for a hairy safety or liveness invariant, a deterministic lint for a linear one. Then it projects coverage back onto the model, so a verified-in-principle invariant with no live test is a named hole.

When to adopt this stack

Use this stack when:

- correctness spans concurrent components and "correct" is nowhere written down
- concurrency invariants are "tested" by a handful of sampled interleavings, so the violating one ships proven-absent
- coverage is a line percentage that averages a critical invariant's zero down to invisibility
- a recurring mistake keeps re-entering through review because a convention decays under a fleet
- you need each correctness obligation discharged at the rigor its shape demands — a proof for the hairy ones, a lint for the linear ones

Typical domains:

- concurrent job pipelines
- distributed systems with safety and liveness invariants
- workflow and state-machine engines
- safety-critical concurrent software

Failure classes it covers

- **The unnamed invariant.** Concurrency correctness lives as scattered ad-hoc guards; the predicates that must hold across components are never named, so no one can say what "correct" even means.
- **The invisible obligation.** Coverage is a percentage over the lines someone happened to test; a whole untested seam or failure edge is invisible because nothing knows it was owed a test.
- **The unsampled interleaving.** A concurrency invariant is "tested" by a unit test that walks a handful of interleavings; the one that violates it is never sampled, and the bug ships proven-absent.
- **The convention that decays.** A recurring mistake keeps re-entering through review; a convention says "don't do that," but a convention decays under a fleet and the class re-appears one commit at a time.
- **The check aimed wrong.** A lint one level too low passes a spec-legal variation it should catch and fires on a legal one it should allow — present but wrong.

- **The comfortable average.** Line coverage reads 80% and everyone relaxes; a critical invariant sits at zero covering tests, invisible because the percentage averages it away.

Composition

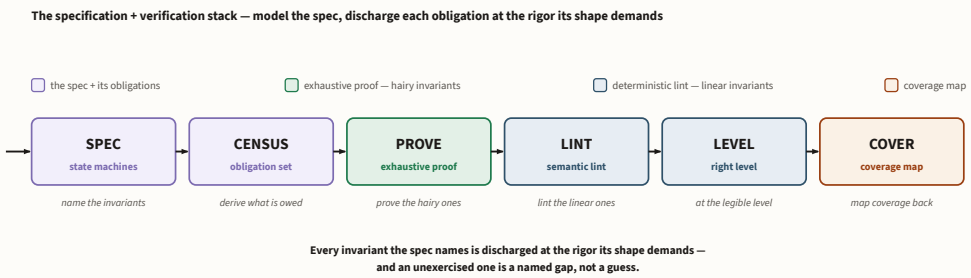


Figure 7.3-1: The specification + verification stack in one picture. Six parts run left to right. The spec (violet): SPEC models the lifecycle as composed state machines and names the cross-machine invariants; CENSUS derives every obligation owed. The rigor tiers: PROVE (green) discharges the hairy invariants with an exhaustive check routed by each invariant's temporal form; LINT (blue) discharges the linear ones with a blocking semantic check at commit, and LEVEL (blue) aims each check at the granularity where its property first becomes legible. COVER (accent) projects coverage back onto the model's nodes, so a verified-in-principle invariant with no live test is a visible gap.

1. **SPEC** Composed state-machine model (typed lifecycles + cross-machine invariants) (§A.3.1)
2. **CENSUS** Model-derived test-obligation census (derive what should be tested, lint the gap) (§A.3.2)
3. **PROVE** Formal invariant verification (temporal form → model checking) (§A.3.3)
4. **LINT** Blocking semantic lints (§A.3.4)
5. **LEVEL** Enforce at the right semantic level (§A.3.5)
6. **COVER** Coverage → model-node mapping (which invariants are actually tested) (§A.3.6)

Two parts build and read the spec; three discharge the obligations at graded rigor; one maps coverage back. The spec is the single source every later part reads.

The constituent parts

The spec is the anchor every later part reads: name the invariants, derive the tests owed, discharge each at the rigor its shape demands — an exhaustive proof for a hairy predicate, a deterministic lint for a linear one — place each check where its property is legible, then project coverage back onto the model's own nodes.

SPEC — model the concurrent lifecycle and name its invariants

Say what "correct" means. Model the concurrent lifecycle as state machines that run at once, and name the predicates that must hold across them as first-class invariants. (SPEC.)

Receives — the system's concurrent behavior: the job lifecycles, the worker and coordinator states, the transitions that today live as scattered ad-hoc guards. Nothing precedes it.

Guarantees — a named, checkable definition of "correct." Each cross-machine predicate becomes an invariant, and each invariant's shape assigns its verification obligation: a safety predicate earns an

exhaustive state-space check, a liveness one a temporal check, a linear one a property test. Concurrency correctness is stated, no longer assumed.

Hands to CENSUS — the invariants and seams every later part reads. The census turns its invariants into an obligation set, the prover reads their temporal form to pick a checker, the coverage map projects tests back onto these same nodes. The spec is only as honest as the parity gate that keeps it equal to the running code, so it leans on the model-coherence stack beneath it.

→ **Deeper treatment:** Appendix B - 13. Composed state-machine model (typed lifecycles + cross-machine invariants).

CENSUS — derive what should be tested, lint the gap

Derive every check you owe. From the model, derive the set of things that should be tested — every external seam to fuzz, every failure edge to inject, every invariant to check — then lint the gap. (CENSUS.)

Receives — the SPEC's invariants and seams, walked as typed data rather than recalled from memory.

Guarantees — a named backlog of test obligations, and a lint on the distance between it and the tests that exist. Coverage stops being a percentage over the lines someone happened to test. An untested seam or unguarded failure edge is no longer invisible; the model knows it was owed a test, and the gap reddens.

Hands to PROVE and LINT — an explicit obligation set for the two rigor tiers to discharge. "What still needs verifying" becomes a query over the model, not a memory. It hands the prover the hairy invariants and the lints the linear ones, so each tier works against a named backlog instead of whatever the author remembered to write.

→ **Deeper treatment:** Appendix B - 21. Model-derived test-obligation census (derive what should be tested, lint the gap).

PROVE — let temporal form route the exhaustive checker

Prove the hairy invariants exhaustively. Give each invariant a temporal form — safety (□P, always) or liveness (P ⇝ Q, eventually) — and let that form route the exhaustive checker that verifies it, so the one violating interleaving is found or ruled out, never sampled past. (PROVE.)

Receives — the SPEC's invariants and the CENSUS's obligation set, filtered to the ones whose shape is hairy enough to demand a proof.

Guarantees — an invariant proven by the method its shape demands, not by a sample. A safety predicate is discharged by an exhaustive state-space model-check; a liveness one by a temporal checker. The one violating interleaving a sampled unit test would never walk is either found or ruled out, so a concurrency bug cannot ship proven-absent.

Hands to LINT — the linear invariants it does not claim. PROVE takes the hairy predicates and verifies them exhaustively, leaving the linear ones for the deterministic checks beside it. Its reach is bounded by how faithfully the SPEC mirrors the code — abstract away the detail that carried the bug and the proof proves the wrong thing.

→ **Deeper treatment:** Appendix B - 18. Formal invariant verification (temporal form → model checking).

LINT — reject the recurring violation at commit

Reject the recurring violation at commit. A fleet of blocking semantic checks reads the tool's own source — banned APIs, silent-catch bans, typed-seam violations — and fails the build on the invariant violations the compiler and review miss. (LINT.)

Receives — the CENSUS's linear obligations and the tool's source structure, read as a parse tree rather than a regex over surface text.

Guarantees — a recurring class of mistake rejected at commit time, not re-caught in review. A convention decays under a fleet; a blocking structural check does not. It moves a recurring judgment out of the reviewer's eye and into a deterministic gate that fires on every commit.

Hands to LEVEL — checks whose correctness now turns on where they fire. Between PROVE and LINT the obligation set is covered, each invariant at the rigor its shape demands. But a deterministic check is only as good as the granularity it targets, so its trustworthiness passes to the next part to secure.

→ **Deeper treatment:** Appendix B - 26. Blocking semantic lints.

LEVEL — fire each check where its property is legible

Check each property where it's legible. Place each check at the granularity where its invariant first becomes observable, not at the cheapest or earliest point. (LEVEL.)

Receives — the checks LINT defines, each needing a scope at which its invariant is actually observable.

Guarantees — a check that fires where its property lives. Aim a lint one level too low and it passes on a spec-legal variation it should catch and fires on a legal one it should allow — present but wrong. Aim it right — check model-to-code drift when an agent returns from a multi-commit task, never at a per-commit hook where the model is legitimately mid-flight — and the check earns trust.

Hands to COVER — checks placed where they can be believed. Only once each fires at its invariant's level is the deterministic tier worth mapping. A check placed a level off fails silently, reading as a false pass rather than a red gate — the most expensive failure to notice, and the one this part exists to prevent.

→ **Deeper treatment:** Enforce at the right semantic level (online).

COVER — project coverage onto the model's nodes

Make "is this tested?" a query. Project coverage onto the model's own nodes — its states, seams, and invariants — so an invariant with no covering test is a visible gap, not a guess from a line percentage. (COVER.)

Receives — the SPEC's nodes and the tests the census owed and the two tiers discharged, mapped test-by-test to the model nodes each exercises.

Guarantees — every invariant's test status as a query. Line coverage reads 80% and everyone relaxes while a critical invariant sits at zero covering tests, averaged into invisibility. The map turns the model into a work-list: an invariant node with no covering test is a visible gap that drives the next test.

Hands off — the stack's final guarantee. Of everything the CENSUS owed and PROVE and LINT discharged, COVER asks which is actually exercised, so a verified-in-principle invariant with no live test becomes a named gap. It is exactly as complete as the SPEC's node set, which the census keeps honest.

→ **Deeper treatment:** Appendix B - 15. Coverage → model-node mapping (which invariants are actually tested).

A DocAble example, end to end

DocAble's job pipeline is concurrent: a parent job fans out into chunks, each chunk walks its own lifecycle, and results fan back in. **SPEC** models the parent and chunk lifecycles as composed state machines and names the cross-machine predicates — for instance, *the fallback file is uploaded to storage before the job row is marked complete*. **CENSUS** walks those machines and derives the obligation set: the seams to fuzz, the failure edges to inject, the invariants to check. **PROVE** takes the hairy safety invariant — no crash interleaving leaves a job marked done with no artifact — and discharges it with an exhaustive state-space check, not a sampled unit test. **LINT** discharges the linear ones deterministically at commit: a banned raw call, a silent catch, a mutation that skips the typed seam. **LEVEL** keeps those lints honest — a model-to-code drift check fires when an agent *returns* from a multi-commit task, never at a per-commit hook where the model is legitimately mid-flight. **COVER** then asks, of every obligation CENSUS owed and PROVE and LINT discharged, which is actually exercised — so an invariant verified in principle but with no live test surfaces as a gap.

Tradeoffs and adoption order

1. **SPEC first, and it is only as honest as the parity gate that keeps it equal to the running code.** Author the machines and the invariants; without them nothing downstream has a spec to read.
2. **CENSUS turns the spec into a work-list.** Cheap, deterministic, re-runs on every change; it makes "what still needs verifying" a query, not a memory.
3. **PROVE and LINT split by shape.** Route the hairy invariants to exhaustive proof — its cost grows with the state space — and the linear ones to commit-time lints, which are cheap and fire on every commit. LEVEL is the judgment that keeps each lint aimed right; misplacement fails silently as a false pass, the most expensive failure to notice.
4. **COVER last.** A projection, not a runtime dependency; it keeps the discharged set honest against the spec's nodes.

The full treatment

Each constituent links to its full pattern — in this appendix for the flagship members, online for the rest. The stack sits on the model-coherence stack (its state machines are that stack's executable data) and feeds the observe → react loop (a verified invariant still needs a live signal when it breaks). The full 83-mechanism catalogue is online in the web edition.

Appendix A - 4. The observe → react loop

A two-page synthesis of the observe → react loop. Five patterns make the fleet's live state legible and every bad state actionable, so an operator — human or agent — drives from typed signals plus written responses instead of scraping logs and reasoning from memory.

The capability

Turn a running-but-wrong pipeline — the worst failure because it is invisible — into a named signal and a procedure that answers it. It discharges two capabilities: *manage work, state, and resources*, and *govern the control estate itself*. Every substrate emits its health onto one typed surface; a written procedure answers each signal; a gate refuses new work while a serious one stands unresolved; and a standing map makes every signal interpretable. The operator reacts to structure, not to scraped text.

When to adopt this stack

Use this stack when:

- fleet health is scattered across a dozen logs in different shapes and learned late, by reading
- a long-running process goes silent and nothing tells a wedged one from a slow one
- a red signal fires but says nothing about what to do, so each response is re-reasoned under incident pressure
- operators keep piling work onto a known-broken substrate because ignoring a signal costs nothing
- an operating agent knows the symptom but not how the substrate is supposed to work, so it mis-operates the fleet

Typical domains:

- autonomous agent-fleet operations
- long-running deploy and CI pipelines
- distributed job systems
- on-call and SRE estates

Failure classes it covers

- **The scraped state.** Fleet health lives in a dozen logs in different shapes; the operator learns of a bad state late, by reading, and reacts to text rather than to structure.
- **The silent long-runner.** A deploy goes quiet; nothing distinguishes a wedged process from one grinding through a slow phase, so a hang is found only by a timeout much later.
- **The signal with no response.** A red state fires but says nothing about what to do; the operator re-reasons the response from scratch each time, inconsistently, under incident pressure.
- **The ignored alarm.** A high-severity alert fires and the operator keeps piling new work onto a possibly-broken substrate, compounding the failure it should have stopped to fix.
- **The symptom without a model.** An operating agent knows the symptom index but not how the substrate is *supposed* to work, so it treats symptoms without a model and mis-operates the fleet.

Composition

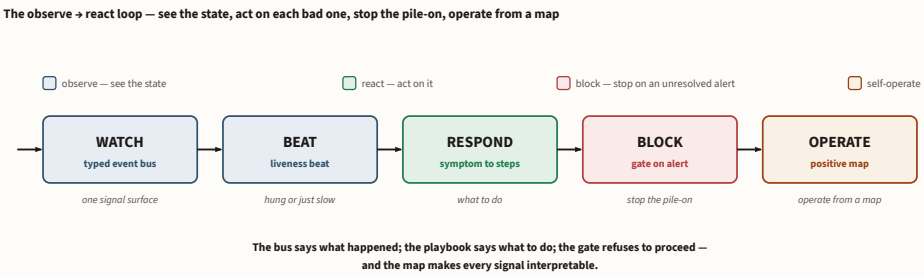


Figure 7.4-1: The observe → react loop in one picture. Five parts run left to right. Observe (fleet blue): WATCH is the typed event bus every substrate emits onto; BEAT is the liveness channel that tells a hung process from a slow one. React (green): RESPOND is a written playbook per signal. Block (churn red): BLOCK refuses new work-dispatch while a high-severity alert is unresolved. Self-operate (accent): OPERATE is the positive map of how the substrate works. The bus says what happened; the playbook says what to do; the gate refuses to proceed until it is cleared; the map makes every signal interpretable.

1. **WATCH** Orchestrator-as-reactor over an event bus (§A.4.1)
2. **BEAT** Deploy heartbeats + stale-worker detection (§A.4.2)
3. **RESPOND** Operational playbooks (§A.4.3)
4. **BLOCK** Cron-alerts gate (§A.4.4)
5. **OPERATE** Operator runbook skill (positive map first, symptom index fallback) (§A.4.5)

Two parts observe, two react, one gives the operator the standing map. The bus is the single surface the rest of the loop reads.

The constituent parts

Five parts run as a loop: a typed bus carries every substrate's health onto one queryable surface, heartbeats sharpen it so a hung process reads differently from a slow one, a playbook says what to do when a signal fires, a gate raises the cost of ignoring a serious one, and an operator skill supplies the standing map.

WATCH — the typed event bus

Make fleet health legible. Give every substrate one typed surface to announce its lifecycle and health on, so the operator reacts to a single signal stream instead of scraping logs. (WATCH.)

Receives — lifecycle and health facts from across the fleet: is cron running, is the merge-train yielding, are tombstones stuck. Nothing precedes it; this is where the loop starts.

Guarantees — a queryable, self-documenting signal surface. Every event carries a topic drawn from a closed const-string registry, so a typo cannot silently create a dead topic that disables a signal. Health is read from structure, not scraped from a dozen logs in different shapes — and read on a defined cadence, not by chance.

Hands to BEAT and RESPOND — the one surface the rest of the loop reads. The heartbeat rides it as a liveness channel, the playbook is keyed to its topics, and the blocking gate downstream watches it for high-severity events. Everything after reacts to what the bus says.

→ **Deeper treatment:** Appendix B - 8. Orchestrator-as-reactor over an event bus.

BEAT — periodic liveness heartbeats

Tell a hung process from a slow one. A long-running process emits a periodic beat carrying its phase and elapsed time, and a sweep flags a worker that has stopped beating; silence past the cadence reads as stale. (BEAT.)

Receives — the runtime state of long operations: a deploy that runs many minutes, a worker grinding through a slow phase. It rides the WATCH bus as that bus's liveness channel.

Guarantees — liveness turned into a signal instead of an ambiguity. The beat proves the process is *moving*, not just alive; a deadlocked process is alive too, but it stops advancing its phase. Silence past the known cadence reads as stale. The signal proves motion, not correctness — a deploy can beat steadily and still fail.

Hands to RESPOND — the difference between "still working" and "wedged." That distinction is exactly what tells the operator whether to wait or to act, which is the first thing the playbook downstream needs to know.

→ **Deeper treatment:** Deploy heartbeats + stale-worker detection (online).

RESPOND — situation-keyed operational playbooks

Answer every signal with a written procedure. For each situation the signals surface, a decision procedure names the trigger, the ordered steps, and the reflexes to avoid — so an operator under incident pressure follows a pre-reasoned response instead of re-deriving one badly. (RESPOND.)

Receives — a fired signal from WATCH: a broken deploy, a wedged cron, a worktree destroyed mid-flight, an alert gate that deadlocked. The bus says what happened; the playbook takes it from there.

Guarantees — a consistent, incident-tested response, reasoned once when no incident was burning. Each procedure gives the steps in order and the sharp edges to avoid — the flailing reset that destroys landed work, the naive cron restart that re-enters the same loop. The value is that the correct steps are written down and discoverable at the moment they are needed.

Hands to BLOCK — the response half of a matched pair. A signal keyed to no playbook is unactioned noise; the gate downstream leans on the playbook to say how a blocking alert gets cleared.

→ **Deeper treatment:** Appendix B - 11. Operational playbooks.

BLOCK — the high-severity alerts gate

Make ignoring a serious signal costly. While an unresolved high-severity alert stands, the gate refuses new work-dispatch until it is acknowledged or resolved. (BLOCK.)

Problem — surfacing a signal is not enforcing a response. An orchestrator can see, miss, or ignore a red state and keep piling new work onto a possibly-broken substrate, compounding the failure it should have stopped to fix.

Solution — read the unresolved high-severity alerts on the WATCH bus at session start, against what was last seen, and make the response mandatory: an unresolved alert refuses the dispatch, worktree-create, and merge tools outright. It is designed deadlock-free — exempt tools plus an alert-resolving dispatch mean the gate can always be cleared, never wedged shut.

Output — a fleet held still until the problem is addressed. The alert names it, the playbook says how to clear it, and this gate refuses to proceed until one or the other is done, leaving the operator skill to supply the standing map that makes the whole loop interpretable.

→ **Deeper treatment:** Cron-alerts gate (online).

OPERATE — the operator runbook skill

Operate from a model, not from memory. A loadable skill gives an operating agent the positive map of how the substrate works — its lifecycles and healthy baselines — first, and a symptom-to-resolving-doc catalog as the fallback when something breaks. (OPERATE.)

Receives — an operator, human or agent, who must know two things: how the substrate runs when healthy, and what to do when it breaks. Where WATCH and BEAT show state and RESPOND and BLOCK handle each bad one, this supplies the standing map those four are read through.

Guarantees — self-operation from a model of the substrate, not from memory. The skill leads with normal so a fault is spottable against a baseline, is keyed to what the operator *observes* rather than the doc they would have to already know, and is generated from a typed source so a reference-validity lint resolves every pointer's file and heading anchor — a moved doc becomes a build error, not a dangling chase.

Hands off — nothing further in the loop. It is the standing map the other four parts are interpreted through; without it, the signals still fire but the operator reads them from recall.

→ **Deeper treatment:** Operator runbook skill (positive map first, symptom index fallback) (online).

A DocAble example, end to end

A DocAble deploy runs long. Historically it would go silent and an operator could not tell a wedged build from a slow one. Now every substrate — deploy, cron, merge-train — emits onto **WATCH**, the one typed bus, and the long-running deploy emits a **BEAT** every interval carrying its phase and elapsed time; a sweep flags a worker that stops beating. When a merge-train tick fails, the event is not a log line to grep but a typed alert on the bus, keyed to a **RESPOND** playbook that names the recovery steps in order. If that alert is high-severity, **BLOCK** refuses the next dispatch, worktree-create, and merge until someone acks or resolves it — so the fleet cannot pile new work onto a broken substrate. And an agent asked to operate the repo loads **OPERATE** first: it reads how the substrate is supposed to work before it touches a symptom, so it drives from a model of the fleet rather than from memory.

Tradeoffs and adoption order

1. **WATCH is the floor.** Without one typed signal surface the rest of the loop has nothing to read. Its cost is one emit per lifecycle fact; the closed topic registry keeps a typo from silently disabling a signal.
2. **RESPOND pairs with it.** Neither half is useful alone — a signal keyed to no playbook is unactioned noise, a playbook with no signal never runs. Adopt them together.
3. **BLOCK when ignoring a signal is expensive.** A deterministic gate over unresolved alerts; it degrades to noise only if alerts are acked without being fixed.
4. **BEAT and OPERATE are complementary.** The loop functions without per-phase heartbeats or the operating map, but a long pipeline is far more legible with beats, and recovery is faster with the map. Add them where the substrate is long-running or agent-operated.

The full treatment

Each constituent links to its full pattern — in this appendix for the flagship members, online for the rest. The loop consumes the specification + verification stack (a proven invariant still needs a live

signal when it breaks) and feeds the governance-of-governance stack (the estate that governs the controls themselves). The full 83-mechanism catalogue is online in the web edition.

Appendix A - 5. The resource-mediation stack

A two-page synthesis of the resource-mediation stack. Four patterns share one machine among dozens of concurrent agent worktrees without letting them trample each other or drown the host: declare what must be serialized, hold each declaration with a host-level mediator, and govern the whole with a live pressure signal.

The capability

Run dozens of agent worktrees on one box at the right degree of parallelism — no collisions, no thrash, no swap. It serves a single capability — *manage work, state, and resources* — on the resource face. It declares which work must be serialized and which may run in parallel, holds each declaration with a host-level lock, and puts a live pressure signal over the whole that refuses new heavy work before it starts and sheds running work when the host spikes. The contract says how many; the mediators hold that many; the pressure gate decides whether they run at all.

When to adopt this stack

Use this stack when:

- many agents or jobs share one machine and trample each other's heavy work
- who may run what, and how many at once, lives as folklore in scattered wrapper scripts
- heavy tools run either one-at-a-time, wasting a capable host, or all-at-once, thrashing it
- a correctly-serialized fleet still drives the host into swap under its own admitted load
- you need parallelism that rises to the machine's capacity and stops there

Typical domains:

- shared build and CI hosts
- multi-agent worktree fleets
- self-hosted runners
- resource-contended shared development environments

Failure classes it covers

- **The undeclared contention.** Who may run a thing, and how many at once, lives as folklore in scattered wrapper scripts; a new call site oversubscribes a resource no one knew was contended.
- **The colliding monopoly.** Two worktrees run the heavy test suite at once on one machine; they saturate I/O and interfere, and both come back slow and flaky.
- **The mismatched cap.** Heavy builds and type-checks run either one-at-a-time — wasting a capable machine — or all-at-once, thrashing it; neither matches the host's actual capacity.
- **The self-inflicted swap.** Even a correctly-serialized fleet drives the host into swap: work admitted while the machine was healthy keeps running as pressure climbs, and the box wedges under its own load.

Composition

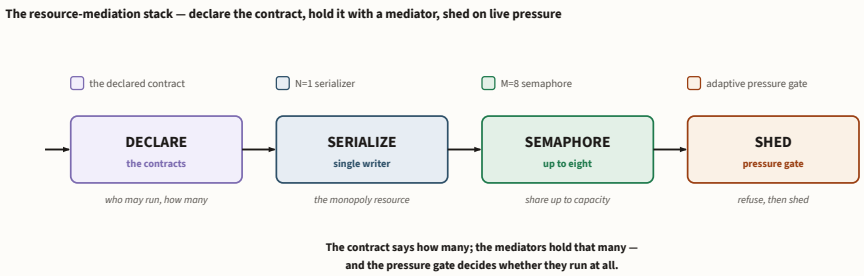


Figure 7.5-1: The resource-mediation stack in one picture. Four parts run left to right. DECLARE (violet) is the typed registry of concurrency contracts — what is serialized, what is single-writer. SERIALIZE (blue) is the host-level flock that admits one run of the heaviest tool at a time (N=1). SEMAPHORE (green) is the counting lock that admits up to eight concurrent runs of the adjacent heavy tools (M=8). SHED (accent) governs a saturable resource with a live pressure signal at two layers — an admission gate that refuses heavy work before dispatch and an execution shed that stops running work on a spike. The contract says how many; the mediators hold that many; the pressure gate decides whether they run at all.

1. **DECLARE** Mediator & single-writer contracts (§A.5.1)
2. **SERIALIZE** Test-serializer (N=1 flock on `dotnet test`) (§A.5.2)
3. **SEMAPHORE** Build-serializer (M=8 semaphore) (§A.5.3)
4. **SHED** Resource-pressure gating (admit before, shed during) (§A.5.4)

One part declares the contracts; two enforce them at fixed cardinality — a strict monopoly and a bounded pool; one adds a live signal over both.

The constituent parts

Four parts run as a chain: a typed registry names the contracts, a serializer enforces the strictest of them, a semaphore enforces the bounded-sharing ones, and a pressure gate bounds whether any of them should run at all, given the host's live state. Declared cardinality holds the count; the live signal holds the condition.

DECLARE — the concurrency-contract registry

Declare what must be serialized. A typed registry names each of the system's concurrency contracts — which subprocess invocations are serialized by a mediator, and which state-mutating functions are single-writer — so "how many at once" becomes data a check enforces. (DECLARE.)

Receives — the system's contended operations: the heavy test runner, the build tools, the state mutators several worktrees might touch at once. Undeclared, who-may-run-what lives as folklore in scattered wrapper scripts.

Guarantees — a declared, coverage-checked contract set. Each entry names its mediator and its cardinality, so a newly-added subprocess or mutator that should be contracted but isn't becomes a coverage-lint failure, not a race discovered late. The declaration is the model side the enforcers act against — an enforcer only ever sees the calls that reach it.

Hands to SERIALIZE and SEMAPHORE — the cardinalities the mediators read. The serializer looks up the single-writer contracts, the semaphore the bounded-sharing ones; both take their degree from this one registry rather than from a convention a new call site can quietly miss.

→ **Deeper treatment:** Mediator & single-writer contracts (online).

SERIALIZE — the single-writer test lock

Hold the strictest contract to one writer. A host-level flock admits a single run of the heaviest, mutually-destructive tool, so concurrent worktrees queue instead of colliding. (SERIALIZE.)

Receives — the single-writer contracts from DECLARE, and every attempt to run the heavy tool. The canonical case is a test runner: several worktrees running it at once saturate CPU, disk, and ports.

Guarantees — one run at a time on the contended resource. An exclusive flock admits a single caller; the second waits. Decisively, an enforcer inside the tool makes the un-mediated path impossible — a raw invocation from an agent worktree is refused, so the serialization is real rather than a convention agents forget under pressure. A wait cap fails loud, so a stuck lock surfaces instead of hanging forever.

Hands to SEMAPHORE — the sibling case. The resource that cannot share takes this flock; the resources that can share take the counting lock beside it — same registry, different cardinality, chosen by the resource's contention profile.

→ **Deeper treatment:** Appendix B - 5. Test-serializer (N=1 flock on `dotnet test`).

SEMAPHORE — the bounded build semaphore

Share up to capacity, then wait. A host-level counting semaphore admits a fixed number of concurrent runs of the adjacent heavy-compute tools, so worktrees get parallelism up to the machine's capacity without oversubscribing it. (SEMAPHORE.)

Receives — the bounded-sharing contracts from DECLARE, and every call to the heavy but parallel-safe tools: the compiler, the type-checker, the code-query and lint tools.

Guarantees — parallelism that rises to capacity and stops there. These tools are numerous and mostly parallel-safe, so single-writer would waste cores while all-at-once would thrash the host; a counting semaphore admits its bound and makes the next caller wait. A per-tool enforcer on each keeps the mediated path the only path. The bound is a tuned guess per machine — too low wastes a big host, too high thrashes a small one.

Hands to SHED — a fixed ceiling a live signal can override. The semaphore bounds how many run; the pressure gate downstream bounds whether they should run at all, and catches the thrash a static bound cannot foresee.

→ **Deeper treatment:** Build-serializer (M=8 semaphore) (online).

SHED — the live-pressure gate

Refuse and shed heavy work under pressure. A live pressure signal governs a saturable host resource at two layers — an admission gate that refuses or defers heavy work before dispatch, and an execution shed that stops heavy work already running when pressure spikes. (SHED.)

Receives — the host's live pressure over the saturable resource (load, memory), plus the heavy work the fixed-cardinality mediators would otherwise admit regardless of the machine's current state.

Guarantees — heavy work neither started into an overloaded host nor left running on one. A cardinality cap bounds how many jobs run, not whether the host can bear them right now. One coarse signal drives both layers: admission refuses before the cost of starting doomed work is paid, and

shedding catches pressure that rose after a job was admitted. One shared signal keeps the two from disagreeing; the same reading is callable, so the operator can consult it for dispatch judgment too.

Hands off — the stack's final bound. Where DECLARE, SERIALIZE, and SEMAPHORE fix how many may run, this decides whether they should run at all — the admission-and-shedding layer over the fixed-cardinality mediators beneath it.

→ **Deeper treatment:** Appendix B - 4. Resource-pressure gating (admit before, shed during).

A DocAble example, end to end

DocAble's development runs six to eight agent worktrees on one shared build machine. **DECLARE** names each contended invocation in a typed registry — the C# test runner is serialized, the build and type-check tools share up to a fixed count, this in-memory mutation is single-writer. **SERIALIZE** holds the strictest of those: the heavy test suite takes an exclusive flock, so when two agents reach it at once the second waits instead of colliding on I/O — both runs come back fast and clean rather than slow and flaky. **SEMAPHORE** holds the looser contracts: up to eight concurrent builds and type-checks run, and the ninth waits, so the machine runs at capacity and stops there. Over all of it, **SHED** watches host pressure — when memory climbs toward swap it refuses to admit the next heavy job, and if a running wave spikes the box it sheds work already in flight, so a correctly-serialized fleet still cannot drive the host into the ground.

Tradeoffs and adoption order

1. **DECLARE first.** Without the contract registry the mediators do not know what to serialize or to what degree. Its cost is authoring the contracts; an undeclared contended invocation is the gap, which the mediators' own coverage check surfaces.
2. **SERIALIZE and SEMAPHORE together.** Same registry, two cardinalities — the resource that cannot share takes the flock (N=1), the ones that can share take the counting lock (M>1). The serializer costs latency on the monopoly resource; the semaphore needs a tuned M per machine and degrades gracefully.
3. **SHED last, and it earns its place under load.** A live-signal gate adapts where a fixed cap cannot. It fails only if the signal lags the real pressure, so read the saturating resource as directly as possible.

The full treatment

Each constituent links to its full pattern — in this appendix for the flagship members, online for the rest. The stack pairs with the observe → react loop (the pressure signal rides the same observability surface) and the context-management stack (both share one machine among many agents). The full 83-mechanism catalogue is online in the web edition.

Appendix A - 6. The governance-of-governance stack

A two-page synthesis of the governance-of-governance stack. Six patterns treat the governance system as its own subject: model the controls as a graph, census which targets are covered, compute a change's blast radius before making it, convert each recurring failure into a proportionate new control, and hold the governing document itself as enforced, queryable infrastructure.

The capability

Govern the control estate the way you govern the product — model it, measure its coverage, bound its changes, and grow it by design. One capability defines it: *govern the control estate itself*. A fleet accumulates lints, gates, mediators, and registries until the estate is a system in its own right, with its own failure modes: controls that collide, targets no one guards, substrate changes with unknown blast radius, and a governing document that rots. The stack turns each of those into a modeled, queryable, self-repairing fact.

When to adopt this stack

Use this stack when:

- the control estate has grown large enough to be a system in its own right, with its own failure modes
- two guardrails collide over one shared resource and the clash surfaces only in production
- a target is guarded by soft aims or by nothing, and no artifact ever asks whether coverage is balanced
- a cross-cutting substrate change lands blind because its blast radius was a grep no one ran completely
- the same failure is re-patched locally each time it recurs instead of the class being closed once
- the governing document itself rots — renumbered, bloated past what fits in context, or drifted from the checks it names

Typical domains:

- agent-collaborative codebases
- large lint, gate, and mediator portfolios
- platform and developer-tooling teams
- long-lived, governance-heavy repositories

Failure classes it covers

- **The colliding guardrail.** Two controls contend over one resource — a lock, a file, a queue — and the collision surfaces only when they trip each other in production, because nothing modeled that they touch the same thing.
- **The blind spot.** A target guarded only by soft aims, or by nothing at all, is discovered the day it fails rather than the day the gap opened.
- **The unbounded change.** A cross-cutting substrate change lands blind; a control that silently assumed the old shape breaks, and the blast radius was a grep no one ran completely.
- **The re-patched class.** The same failure is fixed locally each time it recurs, so the class survives to bite the next agent — and even a team that means to convert it forgets on a long autonomous run.

- **The rotting document.** The doc that carries every other mechanism silently decays — a rule renumbered, bloated past what fits in context, or drifted from the check it names — and every agent boots on a lie.

Composition

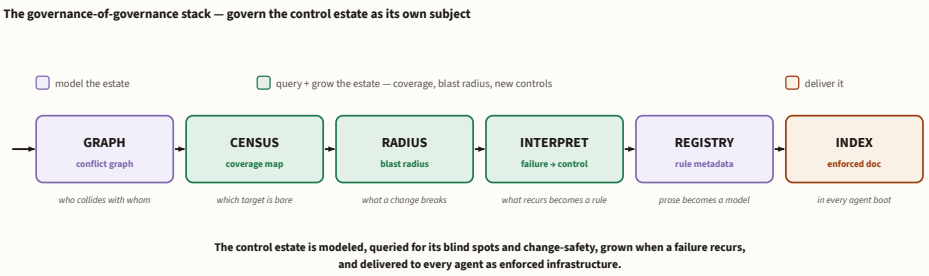


Figure 7.6-1: The governance-of-governance stack in one picture. Six parts run left to right. Model the estate (violet): GRAPH represents each control as a node tagged by trigger and resource footprint, with a conflict edge where two controls contend; REGISTRY attaches machine-readable metadata to each governance rule. Query the estate (green): CENSUS classifies each control by the target it guards, so a bare target is a re-derived gap; RADIUS makes each control declare its substrate assumption, so a change's blast radius is a computed query. Grow the estate (green): INTERPRET converts a recurring failure into a proportionate new control, fired on a cadence. Deliver it (accent): INDEX holds the governance document as a numbered, enforced, capped rule index in every agent's boot context.

1. **GRAPH** Governance graph (mechanism-interaction model) (§A.6.1)
2. **CENSUS** Control-coverage census (controls per governance target) (§A.6.2)
3. **RADIUS** Control↔substrate dependency (computed blast-radius) (§A.6.3)
4. **INTERPRET** Self-governance (detect your own recurring issues; convert each into a tasteful control) (§A.6.4)
5. **REGISTRY** Rule-metadata registry (machine-readable metadata on governance rules) (§A.6.5)
6. **INDEX** CLAUDE.md rule index (the governance document as a mechanism) (§A.6.6)

Two parts model the estate, two query it, one grows it, one delivers it. The graph and the registry are the map the rest of the stack reads.

The constituent parts

Six parts answer as a set: map the controls as a graph so a collision is caught at model time, census which targets are held and which are bare, compute what a substrate change will break before you make it, convert each recurring failure into a proportionate new control, extract the governing rules into a queryable model, and hold the top-level document itself as enforced, capped infrastructure.

GRAPH — the control-interaction graph

Catch colliding guardrails at model time. Model the fleet's governance mechanisms as a typed graph: each control a node tagged by the event it fires on and the resources it reads, writes, or locks; each edge a conflict where two controls contend over one shared resource. (GRAPH.)

Receives — the fleet's existing guardrails: turn-end hooks, pre-commit checks, dispatch gates, host-level lock mediators. Nothing precedes it; this is the map the rest of the stack reads.

Guarantees — collisions caught at model time, not in production. Two controls can place contradictory demands on one commit-set, or contend for one turn-end slot, and neither one's own code shows it — the failure lives in the interaction. A conflict edge over a typed, closed resource vocabulary makes that interaction visible, and a consistency check decides the mechanically-decidable conflicts before a new control is even wired.

Hands to CENSUS and RADIUS — one map, two queries. Because every control is a node with a typed footprint, the census can roll the nodes up per target and the blast-radius query can walk the substrate edges. Both read this graph rather than re-deriving the estate from scratch.

→ **Deeper treatment:** Appendix B - 19. Governance graph (mechanism-interaction model).

CENSUS — the per-target coverage census

Find the estate's blind spots. Classify each control by the target it guards — derived from its own code anchor, never hand-declared — and roll the set up per target, so a bare target is a first-class finding. (CENSUS.)

Receives — the graph's control nodes. It reads the same typed node-set GRAPH drew, now asking not how two controls collide but how many guard each target.

Guarantees — a re-derived map of the estate's own blind spots. A control portfolio grows toward the last painful failure, so effort piles onto the target that just hurt while another accretes nothing, and the imbalance stays invisible because no artifact ever asks whether coverage is balanced. Here a target with zero controls, or with only soft aims and no hard hold, is a first-class finding. A fail-loud classifier refuses any control it cannot place, so the map can never silently mis-credit a control to the wrong target.

Hands to INTERPRET — a named gap to fill. Where the census surfaces an under-watched target, the conversion loop downstream turns that gap into an actual new control rather than a noted absence.

→ **Deeper treatment:** Control-coverage census (controls per governance target) (online).

RADIUS — the computed substrate blast-radius

Compute a change's blast radius first. Each control declares the substrate assumption it bakes in as typed metadata, so "which controls depend on this part of the substrate, and what breaks if I change it" becomes a query, not a grep-and-read. (RADIUS.)

Receives — the same controls the graph holds as nodes, now read through their substrate edges. It reads each control's declared stance toward the substrate it checks against.

Guarantees — a computed blast radius. A control usually buries an assumption about its substrate (a service is a deployment under this directory; the manifest carries this field), invisible until a migration lands and the fleet fails two silent ways: a false FAIL that blocks every release, or a false PASS where a migrated thing drops out of every check. Lifting the assumption into a typed declaration makes the importer and its stance one joinable fact, and a declaration lint refuses any substrate-reading control that leaves it undeclared.

Hands to INTERPRET — bounded change safety. Where the census answers whether a target is covered, the radius answers what a substrate change will break, so the estate reasons about its own change before committing to it.

→ **Deeper treatment:** Appendix B - 14. Control↔substrate dependency (computed blast-radius).

INTERPRET — the failure-to-control conversion loop

Convert each recurring failure into a control. When a failure recurs, name the class and add the smallest durable guardrail that kills it, fired on a cadence rather than on whoever remembers. (INTERPRET.)

Problem — a re-patched instance leaves the class alive to bite the next agent, and even a team that means to convert it forgets on a long autonomous run. The estate stops growing where it most needs to.

Solution — key on the recurrence signal — the second occurrence, never the first — then add the smallest durable guardrail that kills the class: a constraint where one can be built, a sensor otherwise, preferring the constraint that makes the wrong move unrepresentable. Two halves carry it: the conversion judgment is soft, proposing and scaffolding rather than installing; a time-aware reflection hook is hard, firing the loop at most once per window so it aims without decaying into fatigue. Its input is a coverage gap from CENSUS or a bounded risk from RADIUS, plus any failure that recurred this session.

Output — a new control that needs a home. A converted failure becomes a rule or a check, which must land in the enforced, bounded document below, or the estate grows unindexed and the next conversion cannot see what exists.

→ **Deeper treatment:** Appendix B - 12. Self-governance (detect your own recurring issues; convert each into a tasteful control).

REGISTRY — the queryable rule-metadata registry

Make the governing prose queryable. Attach a machine-readable block to each rule in the governance document — identifier, scope, severity, enforcing check, canonical detail location — and extract those blocks into a typed registry the tooling can query. (REGISTRY.)

Receives — the knowledge the graph and census hold only implicitly, plus every rule the conversion loop lands. A rule is human prose until its block makes it extractable.

Guarantees — governance you can query instead of grep. As long as the rules are only paragraphs, every aggregate question — which rules have an automated enforcer, which govern this subtree, which are blocking — is a manual read that rots as the document grows. The metadata block is the bridge: once extracted, a rule citing an enforcer that no longer exists, or a detail pointer that dangles, is a build finding, so the document's claims stay reconciled with the system that enforces them.

Hands to INDEX — a machine-readable spine. The registry gives the index below it the queryable model that lets the governing document be checked against its own rules, not merely read by a human.

→ **Deeper treatment:** Rule-metadata registry (machine-readable metadata on governance rules) (online).

INDEX — the enforced rule index

Keep the governing document from rotting. Hold the top-level rule index as capped, conformance-checked infrastructure — a numbered, stable-numbered index loaded into every agent's boot context, held honest by its own enforcement counterpart. (INDEX.)

Receives — everything the registry models and the conversion loop produces: the rules the estate has learned, each now a short boot-context statement cross-referenced to the canonical doc that carries it in full.

Guarantees — a governing document that cannot silently rot. The document that carries every other mechanism fails two ways: it bloats until nothing in it is read, or its rules drift from the canonical docs they summarize, and both tax every agent, because every agent boots it. A cap lint fails the build when the index outgrows its scannable budget; a conformance lint fails when a rule stops citing its canonical doc; and an admission test governs what may enter, so the cap stays satisfiable without evicting real rules.

Hands off — the estate's delivery surface. This is why the whole stack matters in practice: the governance the graph maps, the census covers, and the radius protects is acted on only because the index puts it in front of every agent — enforced, capped, and conformance-checked.

→ **Deeper treatment:** Appendix B - 9. CLAUDE.md rule index (the governance document as a mechanism).

A DocAble example, end to end

DocAble's fleet runs on scores of controls. **GRAPH** models each as a node carrying its trigger and its resource footprint; when two mediators both take the same host lock, the graph derives a conflict edge and the collision is caught at model time, not in a production deadlock. **CENSUS** classifies each control by the target it guards, derived from its code anchor, and rolls up per target — so a target held only by a soft nudge, with no hard gate, shows up as a bare spot on a queryable coverage map. **RADIUS** has each control declare the substrate it reads, so before a cross-cutting change to the event bus an engineer queries exactly which controls depend on it. When a cherry-pick keeps false-rejecting a second time in one session, **INTERPRET** names the class and scaffolds a proportionate new control instead of re-patching the instance, fired on a cadence so the conversion is not left to memory. **REGISTRY** carries each governance rule's metadata as a queryable block, and **INDEX** delivers the whole numbered rule index into every agent's boot context, capped and conformance-checked so the governing document cannot silently rot.

Tradeoffs and adoption order

1. **GRAPH and REGISTRY first — model the estate.** Tag each control's trigger and footprint; embed each rule's metadata. The graph is only as complete as the controls that register their nodes; the registry as honest as the rules that ship their block, held by a presence lint.
2. **CENSUS and RADIUS next — query it.** Both derive from anchors and declarations that move with the code, so they do not drift the way a hand-kept list would.
3. **INTERPRET grows it.** Its hard half — the recurrence trigger and the periodic hook — is deterministic; its soft half, the taste to keep a new control proportionate, is aided but not replaced by a stronger model.
4. **INDEX delivers it, and is durable infrastructure regardless of model.** Its cost is the discipline of stable numbering and the cap; a rule added without its enforcement counterpart is the gap, which the conformance lint refuses.

The full treatment

Each constituent links to its full pattern — in this appendix for the flagship members, online for the rest. The stack reads the observe → react loop (a recurring alert is a recurrence INTERPRET converts) and shares its delivery surface with the context-management stack (the rule index is loaded into every boot there). The full 83-mechanism catalogue is online in the web edition.

Appendix A - 7. The context-management stack

A two-page synthesis of the context-management stack. Five patterns get the right policy and context to an agent at the moment a decision is due — so it acts on the relevant slice, not the whole corpus and not its memory. This is the book's explicit durable-versus-2026-transient exemplar: each part is marked for whether it is infrastructure that endures or a crutch a stronger model eases.

The capability

Deliver the policy an agent needs at the grain and moment it needs it — the task's constraints into the brief, the standing rules into the boot, an omitted step off the runtime lifecycle, a soft discipline as a rate-limited nudge. It meets one capability on its context face: *manage work, state, and resources*. Five delivery surfaces at four grains, so a bounded-context agent acts on the slice that matters rather than re-reading a whole rulebook or trusting its memory. Read against the years the parts split: some are infrastructure a larger window does not retire, some are 2026-era crutches whose pressure eases as models improve.

When to adopt this stack

Use this stack when:

- a bounded-context agent must act on the relevant policy slice, not a whole rulebook or its own memory
- agents read the wrong part of a rulebook and burn rounds repairing violations they would have honored had the rule been in front of them
- briefs ship missing a piece of safety boilerplate and the agent trips exactly that sharp edge downstream
- a step that must happen at a precise moment depends on someone remembering it
- several soft reminders each fire on their own cadence and compound into noise the operator learns to ignore

Typical domains:

- LLM-agent orchestration
- autonomous coding fleets
- brief and prompt delivery pipelines
- agent-collaborative codebases

Failure classes it covers

- **The unread rulebook.** An agent is handed a task and a whole rulebook; it reads none of it or the wrong part, and violates a constraint it would have honored had the constraint been in front of it.
- **The missing boilerplate.** A brief ships without a piece of safety boilerplate; the agent drifts its working directory, skips the commit cadence, or misses a submodule, and the omission surfaces downstream.
- **The unloaded standing policy.** The standing rules live in a document no one loads; an agent boots without them and re-derives conventions it should have inherited, inconsistently.
- **The forgotten step.** A step that must happen at a specific moment — a checkpoint before compaction, a check before a tool call — depends on someone remembering it, and the moment it is forgotten the state it protected is lost.

- **The wall of nudges.** Several soft reminders each fire on their own cadence; together they become noise the operator learns to ignore, and the alarm fatigue kills all of them at once.

Composition

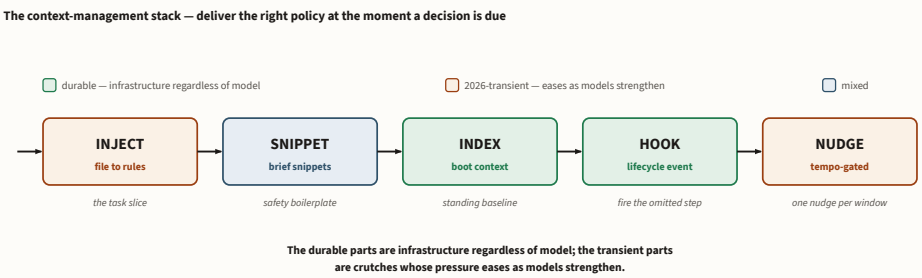


Figure 7.7-1: The context-management stack in one picture. Five parts run left to right, coloured by durability. INJECT (accent, 2026-transient) maps files-about-to-be-touched to their governing constraints and injects the slice into the brief. SNIPPET (blue, mixed) is the registry of mandatory brief snippets asserted at dispatch — transient delivery, durable enforcement. INDEX (green, durable) loads the numbered rule index into every boot context. HOOK (green, durable) binds a script to the runtime lifecycle so an omitted step fires deterministically. NUDGE (accent, 2026-transient) emits at most one tempo-gated reflection per window. The durable parts are infrastructure regardless of model; the transient parts ease as context windows grow.

1. **INJECT** Dynamic context injection (§A.7.1)
2. **SNIPPET** Mandatory snippet-table enforcement (§A.7.2)
3. **INDEX** CLAUDE.md rule index (the governance document as a mechanism) (§A.7.3)
4. **HOOK** Lifecycle hooks (interpose on the agent runtime's events) (§A.7.4)
5. **NUDGE** Reflection-facet substrate (tempo-gated policy nudges) (§A.7.5)

Four parts deliver *policy* at four grains — task-specific, per-brief mandatory, always-on standing, soft reminder; one delivers an *action* at a runtime moment. Each seam names what the part before it hands over.

The constituent parts

Five delivery modalities answer one principle — the right policy at the right moment — each meeting the agent at a different point: inject the constraints a task's files invoke into its brief, assert the standing safety boilerplate is present at dispatch, load the numbered rule index into every boot, fire the omitted step off a runtime lifecycle event, and nudge a standing discipline at a paced cadence.

INJECT — file-scoped constraint injection

Deliver the constraints this task invokes. Map the files an agent is about to touch to the exact constraints that govern them — lints, conventions, boundaries, tests — and inject that slice into the brief before the agent writes code. (INJECT.)

Receives — the task's target files and the fleet's addressable constraint registries. Nothing precedes it; this is where policy first meets the specific task.

Guarantees — the relevant constraints made binding, not merely available. An agent handed a task and a whole rulebook reads none of it or the wrong part, then burns rounds discovering and repairing

violations it would have honored had the rule been in front of it. Resolving the files-about-to-be-touched to the rules that govern them, and rendering that subset into the brief, moves detection left of the cheapest CI gate: prevention before the first commit. The relevance operator is fallible, so this shifts the odds; a downstream gate still guarantees the rule.

Hands to SNIPPET — the task-specific half of one brief. Where injection delivers the constraints THIS task's files invoke, the snippet table beside it delivers the invariant boilerplate every brief needs, and both land in the same brief at dispatch.

→ **Deeper treatment:** Appendix B - 2. Dynamic context injection.

SNIPPET — the mandatory brief-snippet table

Guarantee the safety boilerplate is present. A registry names the mandatory brief snippets — PATH export, commit cadence, worktree discovery, submodule check — and a dispatch-time lint refuses any brief missing a required one. (SNIPPET.)

Receives — the brief INJECT is filling, plus the registry of what every brief of this shape must carry.

Guarantees — no brief ships missing its safety boilerplate. An author who forgets one — say the PATH export, without which dozens of format tests fail for a missing binary — sends an agent that trips exactly that sharp edge twenty minutes in. A docs checklist has no reader and rots as snippets are added; the registry has one, a lint that greps for every required marker and refuses the dispatch on any absence. Some snippets are always-include, others conditional on the brief's shape, so a brief carries what it needs and nothing it does not.

Hands to INDEX — the per-brief half beneath the always-on baseline. Where the snippet table delivers the boilerplate mandatory for THIS dispatch, the rule index below it delivers the standing policy every agent shares, whatever the task.

→ **Deeper treatment:** Mandatory snippet-table enforcement (online).

INDEX — the boot-context rule index

Boot every agent on the same policy. Load the numbered rule index into every agent's boot context, so standing policy is present by construction at the start of every run rather than fetched on demand. (INDEX.)

Receives — the standing rules themselves, each a short boot-context statement cross-referenced to the canonical doc that carries it in full. It sits beneath INJECT and SNIPPET as the layer neither specializes.

Guarantees — every agent boots on the same shared world-model. Standing rules that live in a document no one loads leave an agent re-deriving conventions it should have inherited, inconsistently, one dispatch at a time. Loading the index by construction makes the minimum shared policy present at boot, rather than a reference the agent might consult. A cap lint keeps it inside a scannable budget and a conformance lint keeps each rule citing its canonical doc, so the always-on baseline stays worth booting.

Hands to HOOK — the shift from context to runtime. Where INJECT, SNIPPET, and INDEX deliver POLICY into a brief or a boot, the hook beside them delivers an ACTION at a runtime moment — the same just-in-time principle, applied to the lifecycle rather than the context.

→ **Deeper treatment:** Appendix B - 9. CLAUDE.md rule index (the governance document as a mechanism).

HOOK — the runtime lifecycle hook

Fire the omitted step deterministically. Bind a script to the agent runtime's lifecycle events — turn-stop, pre-compaction, session-start, before-a-tool-call — so a step someone keeps forgetting happens whether or not anyone remembered. (HOOK.)

Receives — the runtime's named lifecycle events and a step that must happen at one of them. Where the layers above deliver policy into context, this one reads the runtime itself.

Guarantees — the omitted step fires whether or not anyone remembered. Some failures live in the loop that drives the agent, not the code it writes: ending a turn with work still queued, compacting without a hand-off, editing outside the worktree. A lint cannot reach these: the omission happens at runtime, in the loop itself. The hook splits enforcement's two halves. The firing is hard, guaranteed by the runtime. The payload is either a hard block that denies the action or soft guidance re-injected into context. The reflex case, hard delivery of soft guidance, makes the aiming deterministic: the same reminder fired exactly at the decision point, every time.

Hands to NUDGE — the substrate you build on the second reflection hook. One hook re-arms one reflex; a second soft nudge starts the fatigue the tempo-gated substrate below resolves.

→ **Deeper treatment:** Appendix B - 7. Lifecycle hooks (interpose on the agent runtime's events).

NUDGE — the tempo-gated reflection substrate

Nudge without alarm fatigue. This is the stack's softest delivery — a rate-limited soft reminder of a standing discipline, at the gentle end of the delivery spectrum. (NUDGE.)

Purpose — reflect the running work against a standing discipline without becoming noise. Fire several nudges, each on its own cadence, and together they become a wall the operator tunes out, killing them all at once.

Mechanism — one tempo-gated substrate consolidates the reflection nudges: a registry of facets, each reflecting the context against one policy dimension it references rather than copies, the whole family emitting at most one reflection per window. It earns its keep at the second facet, not the first.

Guarantee — soft reminders that cannot compound into fatigue. One shared tempo budget caps the aggregate, so a class's facets round-robin for a single window's reflection. Each facet points at its canonical policy, so a moved doc trips a lint rather than rotting in a payload string; per-firing telemetry puts the family on a measured leash, pulled on over-fire or near-zero yield.

→ **Deeper treatment:** Reflection-facet substrate (tempo-gated policy nudges) (online).

A DocAble example, end to end

A DocAble agent is dispatched to change PDF tag-tree code. **INJECT** resolves the files it is about to touch to the constraints that govern them — the typed-seam rule, the ban-lint on raw library calls, the tests that pin the format — and drops that slice into the brief, so the agent sees exactly the rules its files invoke before it writes a line. **SNIPPET** asserts the brief also carries the invariant safety boilerplate: PATH export, commit cadence, worktree discovery, the submodule check. **INDEX** is already present — the numbered rule index loaded at boot gives the agent the always-on baseline the other deliveries specialize. Mid-run, **HOOK** fires a checkpoint before the context compacts, saving state the agent would otherwise lose to a forgotten step. And **NUDGE**, at most once per window, reflects the running work against one standing discipline — a soft reminder that aims without becoming the wall of alarms the operator tunes out.

Tradeoffs and adoption order

This stack is where the book makes the durable-versus-transient call explicit, part by part.

1. **INDEX and HOOK are durable.** An enforced rule index and a deterministic lifecycle binding lean on no model capability; they are infrastructure a larger window does not retire. Adopt them first.
2. **SNIPPET is mixed.** Its delivery half is 2026-transient — a model that reliably held every convention would need fewer pasted snippets — but its assert-at-dispatch enforcement half is durable, since a required snippet's absence is a deterministic check.
3. **INJECT and NUDGE are 2026-transient in degree.** Pre-selecting the relevant slice and nudging a forgotten discipline both compensate for a 2026 limit — a small window, a model losing track over a long run. Their pressure eases as windows grow, though relevance-focusing and fatigue-prevention never fall to zero.

The full treatment

Each constituent links to its full pattern — in this appendix for the flagship members, online for the rest. INDEX's govern-itself facet lives in the governance-of-governance stack the stack shares one machine among many agents with the resource-mediation stack. The full 83-mechanism catalogue is online in the web edition.

Appendix B

Flagship Mechanisms

Appendix B — Flagship Mechanisms

Flagship Mechanisms — the deep dives

The online catalogue tells you **what** each mechanism is. This appendix explains **why** an experienced engineer builds it, **when** they reach for it, and **what mistakes it replaces** — the engineering judgment it embodies. It is deliberately selective and interpretive, not a second copy of the reference documentation: it treats the mechanisms that carry the most weight, one at a time, and points you to the online catalogue for the exhaustive per-mechanism detail.

The stacks in Appendix A adopt whole capabilities; this appendix goes the other way, taking one mechanism at a time — the problem it kills, the shape that kills it, the engineering consequences, and the seam where you wire it in. Read a stack to compose; read a Flagship Mechanism to understand one piece in depth.

The notes are grouped by target zone — Agent, Models-bridge, Product — and numbered straight through, so a cross-reference names a stable locator (§B.1 … §B.29) regardless of which zone it falls in. Every mechanism here also appears as a brick in the Mechanism Catalog (Appendix C), and in full in the online catalogue.

Adopt by capability: the stacks

A stack is a capability you adopt whole — a handful of mechanisms that reinforce each other. These are the reader's first navigable choice: pick the capability you want, then follow it into the mechanisms that make it up.

- **The provenance + fidelity stack**
- **The model-coherence stack**
- **The specification + verification stack**
- **The observe → react loop**

- **The resource-mediation stack**
- **The governance-of-governance stack**
- **The context-management stack**

Appendix B - 1. Brief-linting

The judgment — Put the check at the point of no return.

Role	Agent
Family	Context & dispatch substrate
Used in stacks	—
Enforcement	Hard
Related mechanisms	*Enabler:* Mandatory snippet-table enforcement *See also:* Dynamic context injection *See also:* Role-typed dispatch

The Structure of Brief-linting — its shape at a glance:

The linter runs a battery of presence checks over the brief and returns a verdict at the point of no return: pass launches the agent, any failure refuses the dispatch.

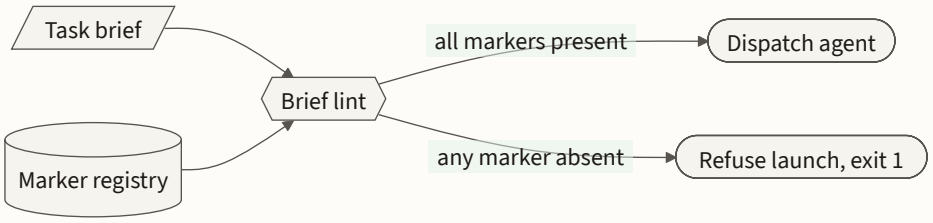


Figure 8.1-1: Accessible description: the brief and a registry of required markers both feed a brief-lint gate; when every required marker is present the agent is dispatched, and when any is missing the launch is refused before the agent starts.

Full description → Brief-linting.

Intent — Statically lint an agent's task brief *before the agent is spawned*, refusing to launch any brief that is missing the markers which make the agent's work safe and well-scoped.

Problem

A brief is the agent's entire world. It is the only instruction set the agent reads before it starts mutating a repository, and a brief with a missing marker does not fail loudly — it fails silently and downstream. Omit the worktree-isolation marker and the agent edits the main line directly, with no fence firing. Omit the dispatch identifier and the lifecycle substrate cannot track or clean the agent. Omit a mandatory safety snippet — a path export, the commit cadence, a submodule check — and the agent trips a sharp edge twenty minutes in.

Brief authoring is a manual act repeated for every dispatch, so the failure is not a one-off. It is a class that recurs on each launch and compounds across a concurrent fleet.

Mechanism

The linter runs a battery of presence checks over the brief text: the worktree-isolation marker present, the dispatch identifier present *and* its matching per-agent marker file on disk, every mandatory snippet's marker string present, the agent role declared. A non-zero exit means do not launch. The linter is not a standalone habit an operator remembers to run. The canonical dispatch path calls it

as a required step and rolls back the just-written marker file when the lint fails, so the dispatch path itself is the enforcement.

Engineering consequences

Review of a brief is a probabilistic check: it catches the missing marker only when the reviewer thinks to look, and reviewers reliably miss the boring structural markers precisely because they are boring. The lint is deterministic at the point of no return — after dispatch the agent is autonomous and the cost of an omission is unrecoverable. A gate catches the boring marker every time; review catches it only sometimes.

Structure is not correctness, though. The lint proves a brief is well-*formed*, not well-*scoped*: a brief with every marker present can still task the agent with the wrong thing. A green lint buys marker coverage, and the human still owns the scoping judgment.

Implementation seam

The lint needs briefs to be structured text with grep-able marker strings, a registry of mandatory snippets it can enumerate, and a dispatch wrapper that calls it on the canonical path and refuses to proceed on failure. Without that wrapper the lint is optional, and an optional gate is skipped under time pressure.

A mature lint adds a *content* layer on top of marker presence — does the brief cite a real file and line, declare its footprint, name a reachable commit? That layer has a trap: a check that fires on any mention of a file or a commit-shaped token false-positives on briefs where the citation is not required, and an operator who learns the lint cries wolf starts ignoring it. The fix is to have each brief declare its genre, so a content check fires only when the genre demands that citation. A lint tuned out is worse than a check never written.

Known limitations

Each new marker is a maintenance edge: a new mandatory snippet means both a new check and threading the marker into the brief template, and drift between the two produces false rejections. The whole gate is bypassable by design, through a human-only, audit-logged escape hatch, so its floor is the discipline of not misusing that hatch. And because it validates form rather than intent, it can wave through a perfectly-structured brief that points the agent at the wrong work.

Appendix B - 2. Dynamic context injection

The judgment — Push the constraints into context; don't hope the actor pulls them.

Role	Agent
Family	Context & dispatch substrate
Used in stacks	The context-management stack
Enforcement	Soft
Related mechanisms	*Counterpart:* Reflection-facet substrate (tempo-gated policy nudges) *See also:* Brief-linting

The Structure of Dynamic context injection — its shape at a glance:

One slicing operator runs in two directions. Forward maps target files to the constraints that will govern them and pushes them into the brief; reverse maps a diff's line ranges to the findings it introduced and feeds a self-heal gate.

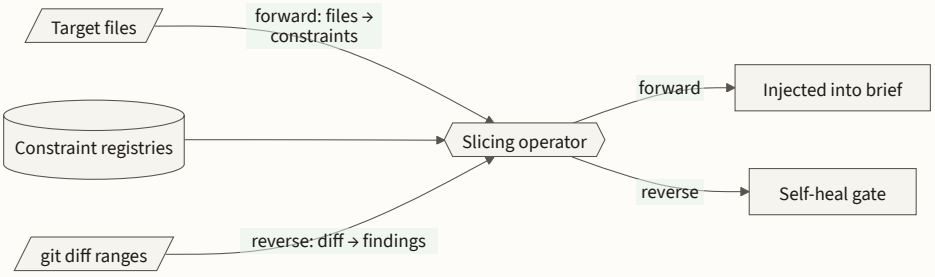


Figure 8.2-1: Accessible description: a slicing operator reads the constraint registries and runs in two directions — forward, mapping target files to the constraints injected into the brief before the agent writes, and reverse, mapping a diff's line ranges to the findings it introduced, which feed a self-heal gate.

Full description → Dynamic context injection.

Intent — Map the files an agent is about to touch to the exact constraints that govern those files — lints, conventions, component boundaries, tests — and inject that subset into the agent's brief before it writes code, moving detection left of the cheapest CI gate.

Problem

The recurring failure is constraint under-specification at dispatch time. A coding agent lacks the tacit knowledge an experienced engineer has of which rules apply to a given change, so it makes plausible edits that violate them, then spends rounds discovering and repairing the violations — "pinball." A layered validation hierarchy makes it worse: context is lost between where the agent authored the change and where the failure surfaces. Across many concurrent agents, the wasted work multiplies.

Mechanism

One slicing operator runs in two directions. Forward (`files → constraints`): given the target files, predict which constraints will apply and pull their declarations plus fix-hints — this powers pre-briefing, either as brief-time auto-injection or an agent-discovery-time pull once the agent knows what it will touch. Reverse (`diff → findings`): intersect checker output with the change's diff line

ranges to attribute which findings this change introduced — this powers CI self-heal, asking an agent to fix only what it broke.

Engineering consequences

The applicable rules land in context whether or not the agent would have gone looking — push, not pull. It is a change of status, relocating a rule from available to binding, because a brief is mandatory reading by construction where a doc is optional reference. Forward injection stays advisory, though: it raises salience and shifts the odds, but a downstream gate is still what guarantees the rule.

Implementation seam

A constraint-extraction tool for the forward slice and the diff-line-range attribution machinery for the reverse, plus one adapter per file-addressable registry — the lint fleet's scope tags, the component model, the banned-API list, the test corpus, the doc index. Each constraint must declare a file scope and carry an actionable fix-hint, or it cannot be selected or acted on.

Known limitations

Garbage-in: a rule with no scope tag can't be sliced, and one with no fix-hint can't be acted on — the mechanism depends on that discipline fleet-wide and does not create it. The relevance operator is itself fallible: over-injection floods the brief with noise and lowers the salience it trades on, while under-injection silently omits a governing rule. Each adapter must stay in sync with its registry, or the sliced set drifts from truth.

Appendix B - 3. Staged deploy gates (canary → smoke → promote)

The judgment — Gate before exposure, cheapest signal first; frequency raises the stakes.

Role	Agent
Family	Gates & merge-train
Used in stacks	—
Enforcement	Hard
Related mechanisms	—

Full description → Staged deploy gates (canary → smoke → promote).

Intent — A deploy pipeline that escalates *canary → smoke → promote*, blocking promotion to production until each cheaper stage passes on a traffic-free revision, so a bad build is caught before users see it, not after.

Problem

Shipping a build straight to production means a regression lands on users; the failure *is* the incident. This is standard practice everywhere, but it matters far more at agentic velocity, precisely because deploys are frequent and agent-initiated. The more often you ship, the more often an un-gated bad build reaches users.

Mechanism

Version bump → build → tag a canary revision that takes no production traffic → smoke-test against the canary URL → promote to production on green → GC old revisions. A pre-launch predicate gates the whole thing on cheaper signals so a doomed deploy is never started: confirm lints are green, no known flaky class is live, and the changed-since-main lint pass is green before paying for build minutes. Heartbeats emit liveness during the long phases.

Engineering consequences

Rollback is reactive and user-visible — by the time you roll back, users have already hit the break. Staged gates are proactive: the canary is smoke-tested on a revision no user can reach, so the break is caught before promotion. The pre-launch predicate pushes the gate earlier still, refusing to *launch* a deploy that will predictably fail rather than spending build minutes to discover it. Being standard practice, the value here is defense-in-depth, not novelty.

Implementation seam

The pipeline needs canary capability — deploying a revision that takes no production traffic — plus a smoke suite that meaningfully exercises the canary URL against real dependencies, not stubs. It needs promotion and rollback primitives with revision GC, and a pre-launch green signal (lints, changed-since-main, flaky-class check) standing as the pre-launch predicate.

Known limitations

Staging costs real build minutes; the pre-launch predicate exists to avoid spending them on a deploy that was never going to pass. Smoke is not full coverage — a thin suite lets real breaks through the gate, since the canary is only as good as what the smoke actually checks. The gate infrastructure can itself drift: validating via the deploy pipeline assumes the pipeline is sound, and a drifted canary or smoke path gives false confidence. Each stage also adds wall-clock before users get the change.

Appendix B - 4. Resource-pressure gating (admit before, shed during)

The judgment — Gate a saturable resource on live pressure at both admission and execution.

Role	Agent
Family	Mediators & resource locks
Used in stacks	The resource-mediation stack
Enforcement	Hard
Related mechanisms	*Counterpart:* Aggregate-compute protection (`lint-all` host mutex)

Full description → Resource-pressure gating (admit before, shed during).

Intent — Govern a saturable host resource with one live pressure signal read at two layers: an *admission gate* that refuses or defers heavy work before dispatch, and an *execution shed* that stops heavy work already running when pressure spikes (our instance: a GREEN/YELLOW/RED host-load monitor gating agent dispatch and shedding heavy compute at the mediators).

Problem

A cardinality cap bounds *how many* heavy jobs run, not *whether the host can bear them now*. Two failures follow. Dispatch-into-overload: the orchestrator admits a heavy agent onto a saturated machine because the only pre-dispatch check guards a different resource, such as free disk; the agent starts, is refused at the compute mediators, and burns wall-clock with no headroom. Run-into-overload: pressure rises after admission and nothing stops the now-too-heavy job. A cap and a single-resource check miss both.

Mechanism

A host monitor reports a coarse pressure level over the saturable resource — load or memory. Three consumers read that one signal:

- **Admission gate (pre-dispatch).** A gate sibling to the pre-dispatch checks refuses or defers a heavy dispatch under RED, so a heavy brief never lands on a red host.
- **Execution shed (at compute).** The compute mediators refuse-and-shed heavy-class work under RED, stopping a job pressure has overtaken.
- **Advisory read (callable).** A plain callable the operator consults in judgment, outside any gate.

Engineering consequences

Admission prevents the startup cost; execution shedding catches pressure that rose after admission, which the gate could not foresee — the two layers are not redundant. One shared reading keeps them from disagreeing; admit-then-shed churn is that disagreement, and moving the check left to dispatch is the same shift-left as a cheap gate before an expensive one.

Implementation seam

The signal needs a pre-dispatch gate seam sibling to existing admission checks, an execution-time shed at the compute step, and a heavy-versus-light work class so light work isn't gated by a signal

only heavy work saturates. As-built, the disk-floor admission gate and the pressure-driven execution shed are wired and the monitor is callable; the load-pressure admission gate is the extension that closes the dispatch-into-overload waste.

Known limitations

GREEN/YELLOW/RED is coarse: a too-eager RED starves throughput, a too-lax one still admits overload, so the thresholds are a tuning surface. Admission must *defer* with a wake condition, never drop, or a brief refused under sustained RED starves. If the two gates read different signals or thresholds, the churn returns. The advisory read is soft — its value is the operator choosing to consult it.

Appendix B - 5. Test-serializer (N=1 flock on `dotnet test`)

The judgment — Process isolation isn't resource isolation; ban the un-mediated path.

Role	Agent
Family	Mediators & resource locks
Used in stacks	The resource-mediation stack
Enforcement	Hard
Related mechanisms	*See also:* Build-serializer (M=8 semaphore)

Full description → Test-serializer (N=1 flock on `dotnet test`).

Intent — A host-level wrapper that serializes the test runner to a single writer via an exclusive flock, so concurrent agent worktrees on one machine don't saturate I/O and interfere with each other's test runs (our instance: an N=1 flock on `dotnet test`).

Problem

Several worktrees each running the test runner on one host saturate CPU and disk and interfere — port contention, shared build artifacts, I/O thrash — so tests flake or hang for reasons that have nothing to do with the code under test. The failure is false-flaky tests plus wall-clock blowup, and it recurs whenever two or more agents test at once, which under a fleet is most of the time. Worse, a false flake sends an agent chasing a non-bug.

Mechanism

The serializer acquires an exclusive flock on a host-global lock before invoking the test runner (N=1 writer), then runs it; a 30-minute wait cap fails loud if the lock is stuck. A module-initializer enforcer inside the test assembly makes the un-mediated path impossible: a raw test invocation launched from an agent-worktree working directory is refused, so the mediated path is the only path. When the filter names a fuzz or campaign run, coverage collection is auto-appended. Adjacent heavy tools — build, compiler, type-checker — route through a sibling serializer at a higher lock cardinality.

Engineering consequences

The ban makes the serialization real rather than a convention agents forget under time pressure: separate processes still share one host's CPU, disk, and ports, so process isolation alone does not prevent destructive contention. The distinction is a mediated single-writer whose raw call is structurally banned, versus uncoordinated processes contending for a shared machine. The deliberate trade is correctness of results over raw parallelism.

Implementation seam

Three parts carry the pattern: a host-global lock file every worktree contends on as the single point of serialization; an enforcer that can intercept the raw tool — here a module initializer that runs before any test — so the mediated path is the only path; and a wait cap that fails loud, so a stuck lock surfaces instead of hanging forever. An audited environment-variable escape exists for humans.

Known limitations

Serialization is wall-clock cost: N=1 means tests queue, and a long run blocks every other worktree behind it. A stuck lock stalls everyone — the fail-loud cap bounds the damage but does not eliminate it. The human bypass is a hole; misused, it reintroduces the contention the serializer exists to kill. And the lock coordinates one machine only — it does nothing across hosts.

Appendix B - 6. Agent registry (append-only log + marker cache)

The judgment — Read liveness from a record, never infer it from side effects.

Role	Agent
Family	Lifecycle & observability
Used in stacks	—
Enforcement	Hard
Related mechanisms	*Enabler:* Merge-train MIS batching *Consumer:* Tombstone commits (lifecycle close records)

Full description → Agent registry (append-only log + marker cache).

Intent — An append-only registry, dual-written by every lifecycle tool, that is the authoritative record of which agents are in flight, so cleanup, tombstone, and merge decisions read a recorded fact instead of guessing from filesystem timestamps.

Problem

With a fleet of concurrent agents living in worktrees, *"which agents are live right now?"* drives every reclaim decision: cleanup, tombstoning, merge readiness. Answer it by scanning worktree directories and comparing modification times and you get a heuristic that races with live agents — an agent mid-work looks identical to a stale one and has its worktree destroyed under it. The failure is unsafe reclaim of live work, and it recurs on every cleanup pass.

Mechanism

The dispatch wrapper's prepare step dual-writes an append-only registry log and a per-agent marker cache at dispatch; every lifecycle tool updates both. The stale-cleanup path queries a three-gate chain — registry, then marker, then git-lock — before removing anything. Tombstone and worktree-clean refuse to operate on an agent whose marker exists. The registry is authoritative; the marker cache is a fast index the registry wins over on any divergence.

Engineering consequences

Liveness becomes a lookup against a recorded fact, not an inference, and a recorded fact cannot race the way an inferred one does. The whole guarantee rests on universal dual-write: a side-door mutation that changes lifecycle without writing the registry silently brings the timestamp race back, so the record is only as strong as its weakest writer. Because it is consulted before every destructive op, the record is protective only where a gate actually queries it.

Implementation seam

Two artifacts carry the pattern: the append-only registry log plus its per-agent marker cache, and the destructive-op gates (cleanup, tombstone, worktree-clean) that query them before acting. The dispatch wrapper's prepare step is the single point that seeds both writes.

Known limitations

Completeness rides on dual-write discipline, and it is fragile: one tool that mutates lifecycle without writing the registry reintroduces the exact race the registry removes. The append-only log grows unboundedly and needs rotation. Registry and marker can diverge if one write fails; “registry authoritative” resolves it, but a tool that trusts the cache alone can be briefly misled.

Appendix B - 7. Lifecycle hooks (interpose on the agent runtime's events)

The judgment — Bind the step to a lifecycle event; the firing is hard, the payload can be soft.

Role	Agent
Family	Lifecycle & observability
Used in stacks	The context-management stack
Enforcement	Soft·Hard
Related mechanisms	*Counterpart:* Pre-commit hook (3-stanza, tree-sha markers) *See also:* Cron-alerts gate *Generalization:* Reflection-facet substrate (tempo-gated policy nudges) *See also:* Dynamic context injection

Full description → Lifecycle hooks (interpose on the agent runtime's events).

Intent — Bind a script to the agent runtime's lifecycle events — turn-stop, pre-compaction, session-start, before-a-tool-call — so a step the operator keeps omitting at runtime fires deterministically, whether or not anyone remembered it.

Problem

Some recurring failures live not in the code an agent writes but in the loop that drives it: the operator skipping a step at a predictable moment. Ending a turn with ratified work still queued. Compacting context without first writing a hand-off. Opening a session without reading the alert backlog. Editing outside the sanctioned worktree. A lint cannot reach these — there is no source artifact to analyze, and the omission happens at runtime. A house-rule only aims a probabilistic operator, and it rots, because the step gets skipped exactly when attention is thin.

Mechanism

The runtime exposes named lifecycle events and a surface to register scripts against them. A hook is a script bound to one event; the runtime invokes it and reads its result. This splits the two halves of enforcement a lint fuses. The firing is hard, guaranteed by the runtime. The payload is either a hard block that denies the action, or soft guidance re-injected into the agent's context that aims the next decision without compelling it.

Engineering consequences

The check runs whether or not anyone remembered; the reflex case is hard delivery of soft guidance — the same house-rule fired exactly at the decision point, every time. Two design constraints follow: keep the check cheap, because it fires on every occurrence of the event, and make a guidance hook fail-open, since a crash must not wedge the loop. A soft payload must also instrument its own firing: it dies silently otherwise — ceasing to fire unnoticed, or over-firing into tune-out — so it ships firing telemetry and lives on a measured leash with a written pull condition.

Implementation seam

Scripts registered against named runtime events, plus a build-time output-conformance check that validates every wired hook against the runtime's actual schema. Without that check a hook validated only against its own idea of the contract can emit a shape the runtime drops, pass its own test green, and run wired-but-dead in production — the worst fail-open, since it looks wired and does nothing.

Known limitations

It fires on every event, not only when needed, so the tax is constant unless the check stays cheap. A guidance hook can be ignored — only the blocking variant compels. A buggy hook is a loop-level outage: a misfiring block, or a guidance hook that crashes without fail-open, stalls the whole session. And it can nag — a hook that misreads its state cries wolf and gets tuned out. As reflection facets accrue, firing each as its own hook is the alarm-fatigue trap by another door; consolidate them into a single tempo-gated reflection (online).

Appendix B - 8. Orchestrator-as-reactor over an event bus

The judgment — Build the observe-then-react loop: a structured signal plus something that acts on it.

Role	Agent
Family	Lifecycle & observability
Used in stacks	The observe → react loop
Enforcement	Hard
Related mechanisms	*Enabler:* Cron-alerts gate *See also:* Agent registry (append-only log + marker cache) *See also:* Lifecycle hooks (interpose on the agent runtime's events)

The Structure of Orchestrator-as-reactor over an event bus — its shape at a glance:

Emitters publish topics drawn from a closed registry; the orchestrator polls the queryable surface, matches each event to its playbook entry, and takes the prescribed action — the playbook is the active half that turns a signal into a reaction.

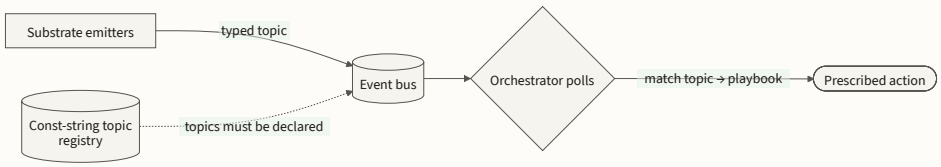


Figure 8.8-1: Accessible description: substrate emitters publish typed topics, drawn from a closed const-string registry, onto the event bus; the orchestrator polls the queryable surface, matches each event to its playbook entry, and takes the prescribed action.

Full description → Orchestrator-as-reactor over an event bus.

Intent — A structured event bus with a closed, const-string topic registry and a companion playbook, over which the substrate emits lifecycle and health events. The bus turns the orchestrator into a reactor over the fleet: it reads health from a queryable, self-documenting signal surface and reacts to each event with a playbook-prescribed response, keeping a fleet of agents productive over long sessions instead of drifting into silent breakage.

Problem

A fleet's health — is cron running, is the merge-train yielding, are tombstones stuck — is invisible without a signal surface, so degradation accretes silently: cron can be broken for hours before anyone notices. And without a reaction loop the orchestrator is a passive observer that can only steer the fleet if it reacts to what the substrate reports. The failure is silent substrate degradation paired with an un-reacting orchestrator, and it recurs continuously across a long session while each individual dispatch still looks locally fine.

Mechanism

Emitters call the bus with a topic drawn from a closed const-string registry, lint-enforced so a typo cannot create a dead topic that silently disables a signal. A playbook maps each topic to its healthy

baseline, its what-looks-wrong signs, and the response entry to open. A substrate-observability rule requires any design doc that introduces a topic to ship an observability block; the lint lands audit-only, then promotes to blocking. A monitoring-cadence rule sets consumption: the orchestrator polls at session start and after cherry-pick waves, with named anomaly triggers such as repeated merge-train yields or a prolonged no-op with tombstones queued.

Engineering consequences

A structured, queryable, self-documenting surface replaces the pull model of remembering to grep prose logs. The playbook is the active half: it turns a raw signal into a reaction rather than a passive read. Emission is hard and mechanical, but the bus itself does not block — a derived alerts gate does that. Acting on the signal depends on the orchestrator honoring the poll cadence.

Implementation seam

Four artifacts carry the pattern: the event bus and its const-string topic registry, the playbook keyed by topic, the observability-block lint that keeps every emitting substrate documented, and the session-start plus post-cherry-pick monitoring cadence.

Known limitations

A topic without a playbook entry is emitted but not interpretable — the observability-block rule exists because that gap is the common failure. Consumption is discipline, not machinery: the bus can carry a perfect signal and still be ignored if the orchestrator skips the cadence. And every new topic adds registry, emit-point, and playbook maintenance that must stay synchronized or the surface goes stale.

Appendix B - 9. CLAUDE.md rule index (the governance document as a mechanism)

The judgment — Govern the governing document like an artifact, or it rots.

Role	Agent
Family	Governance-doc mechanisms
Used in stacks	The governance-of-governance stack; The context-management stack
Enforcement	Soft·Hard
Related mechanisms	*Counterpart:* Epic Definition-of-Done (Final-Opus trust-nothing re-run) *Consumer:* Dynamic context injection *See also:* Docs hierarchy + governance index *See also:* Mandatory snippet-table enforcement

Full description → CLAUDE.md rule index (the governance document as a mechanism).

Intent — Treat the top-level governance document as enforced infrastructure: a stable-numbered rule index, loaded into every agent's boot context, held honest by its own enforcement counterpart — a bloat/cap lint plus a rule-conformance lint — so the document that carries every other mechanism cannot silently rot.

Problem

The failure class is governance that doesn't bind and doesn't last. A convention that lives in someone's head or a stale wiki page never reaches a fresh agent, so the failure it was meant to prevent recurs. A living rules document rots the other way: it bloats until nothing in it is read, or its rules drift out of sync with the canonical docs they summarize. Both compound under a fleet — the document is booted by every agent, so a bloated or drifted index taxes or misleads every dispatch, continuously.

Mechanism

Each rule is a short boot-context statement plus a cross-reference to the canonical deep doc that carries it in full. Numbers are stable, never renumbered, so the index doubles as a citable namespace addressable across the codebase. An admission predicate — the earns-its-spot test: regression-preventing, and non-derivable from the local file, and non-local — decides what may enter; a router sends everything else to a sub-doc or a code comment. Two blocking lints hold the form: a bloat/cap lint fails the build past the scannable budget, and a rule-conformance lint fails when a rule stops cross-referencing its canonical doc.

Engineering consequences

A rule written here is enforced on every subsequent agent boot without re-inspection — binding by construction, not advisory reference. The document is governed the way an artifact is: it carries a budget, an admission predicate, and lints that fail the pipeline. The same load that makes it binding is its price — it taxes every dispatch across the whole fleet, so benefit and cost are the same thing.

Implementation seam

The governance document itself, its boot-context loader, the bloat/cap lint, the rule-conformance lint, and the “what belongs in this file” meta-section that carries the admission rule and its router. The loader makes the index binding; the two lints keep it scannable and undrifted.

Known limitations

A hard budget means perpetual triage: admitting a new rule eventually means evicting one to a sub-doc, so the index is never done and the eviction call is judgment-heavy. Presence is not obedience — a rule in the index does not make agents follow it, which is why a separate audit re-runs owned checks at HEAD rather than trusting the index (Appendix B). Stable numbering accretes history: retired rules leave gaps forever.

Appendix B - 10. Epic Definition-of-Done (Final-Opus trust-nothing re-run)

The judgment — Re-verify at the boundary; recorded claims rot.

Role	Agent
Family	Governance-doc mechanisms
Used in stacks	—
Enforcement	Hard
Related mechanisms	*Counterpart:* CLAUDE.md rule index (the governance document as a mechanism)

Full description → Epic Definition-of-Done (Final-Opus trust-nothing re-run).

Intent — An Epic-close gate mandating a “trust nothing” final review that re-runs every owned pin test and lint at HEAD, rather than trusting phase markers or prior claims, so an Epic cannot close on stale or rotted assertions.

Problem

An Epic spanning many dispatches accumulates claims — phase markers, “lints pass,” pin-test counts — and those claims rot as sibling Epic sweeps churn the substrate underneath them. Close on the stale claims and you ship an Epic whose defenses no longer hold; across one back-catalogue review, 7 of 7 Epics self-marked done carried quality-of-defense gaps. The failure is a premature or false close, and it recurs at every Epic boundary.

Mechanism

An Epic-close tool enforces that every cited commit is reachable from the mainline by ancestry or patch-id; a missing one requires a logged override carrying a reason. On a clean close it rewrites the status and closed-date fields atomically, moves the file into the closed tree, and regenerates the index. The Definition-of-Done itself carries a fixed set of mandatory criteria — among them the final re-run of owned pins and lints at HEAD, the docs and index updates, the tag routing-audit, and the filing of any design-doc follow-up.

Engineering consequences

The gate trusts nothing recorded; it re-derives the verdict from the substrate as it stands now. That makes it the audit counterpart to the rule index: the index’s lints keep its form honest, and this re-run keeps an Epic’s claims honest. Whatever the re-run surfaces routes into filed work rather than a footnote, so a rediscovered gap becomes a task, not a note.

Implementation seam

The reachability- and patch-id-gated close tool plus the multi-criterion Definition-of-Done template. The tool reads each Epic’s declared set of owned pin tests and lints to re-run, so the “owned” declaration is the contract the gate verifies against.

Known limitations

The re-run is expensive — reviewer time plus a full pass of owned pins and lints at close, deliberately heavyweight because a false close is worse. The override is a hole: it exists for legitimately unreachable commits and is logged, but it is a way past the reachability check. And “owned” must be accurate — an Epic that under-declares what it owns re-runs too little and can still close on a rotted but unlisted defense.

Appendix B - 11. Operational playbooks

The judgment — Pre-reason the response before the incident; re-deriving under pressure fails.

Role	Agent
Family	Governance-doc mechanisms
Used in stacks	The observe → react loop
Enforcement	Soft
Related mechanisms	—

Full description → Operational playbooks.

Intent — A library of documented, situation-keyed decision procedures — *when situation X arises, take these steps in this order* — that agents and orchestrators consult instead of reasoning from scratch, so a recurring operational situation (a broken deploy, a wedged cron, a stuck worktree, a chicken-and-egg recovery) gets a consistent, pre-reasoned, incident-tested response.

Problem

Operational situations recur: a deploy fails a known way, a cron loop can't self-recover, a worktree is destroyed mid-flight, an alert gate deadlocks. Each time, an agent under incident pressure re-derives a response from first principles and gets the sharp edges wrong. A flailing reset destroys landed work, a naive cron restart re-enters the same loop, a "cleanup" removes a live worktree. The response comes out inconsistent and error-prone, and the cost lands hardest exactly when time is shortest.

Mechanism

Each playbook names a triggering situation, the ordered response steps, and the anti-pattern reflexes to avoid — the reset that destroys, the restart that loops. Playbooks are cross-referenced two ways: from a terse rule index that points at the long-form procedure, and from the substrate that emits the triggering signal, so the observability surface for a topic carries baseline-healthy, what-looks-wrong, and *which playbook to open*. The orchestrator is instructed, at the trigger, to consult the relevant playbook rather than improvise.

Engineering consequences

The correct steps are written down and discoverable at the moment they are needed, reasoned once when no incident was burning, so the response encodes hard-won judgment instead of a guess made under pressure. A playbook sits closer to a reusable skill than to prose documentation: it names the trigger, gives the ordered steps, and lists the reflexes to avoid. The leverage is entirely discoverability plus habit — the procedure has to be findable at the trigger, or it is never opened.

Implementation seam

The per-situation entry carries a fixed shape: the triggering question, the inspect command, a quantified baseline-healthy, and a *what-looks-wrong → what-it-means → what-to-do* row that includes the deadlock or wedge escape where a substrate can get stuck. Each playbook needs a home plus

cross-references from the rule index and from the signal that triggers it. Add one entry per signal the substrate emits.

Known limitations

The mechanism is soft: a playbook informs, nothing forces the agent to open or follow it. Playbooks rot — when the substrate changes, a playbook whose steps aren't updated actively mis-directs the response, worse than no playbook at all. One written for a retired failure mode is dead weight. And a playbook is not prevention: a situation that recurs often enough should be designed out or gated by a hard sensor, not merely documented.

Appendix B - 12. Self-governance (detect your own recurring issues; convert each into a tasteful control)

The judgment — Convert a recurring failure into its smallest class-killing control, on a cadence.

Role	Agent
Family	Governance-doc mechanisms
Used in stacks	The governance-of-governance stack
Enforcement	Soft·Hard
Related mechanisms	*Sibling:* Operator runbook skill (positive map first, symptom index fallback) *Enabler:* Lifecycle hooks (interpose on the agent runtime's events) *Consumer:* CLAUDE.md rule index (the governance document as a mechanism) *Generalization:* Blocking semantic lints

The Structure of Self-governance (detect your own recurring issues; convert each into a tasteful control) — its shape at a glance:

Two halves, packaged. A hard reflection hook fires on a lifecycle event, at most once per window. It prompts a soft conversion loop: name the failure **class**, then pick the smallest control from an ordered vocabulary — prefer a constraint, fall back to a sensor — scaffold it, and hand it off to be installed in the bounded rule index. A design-time audit runs the same stance forward, preventing a class by construction.

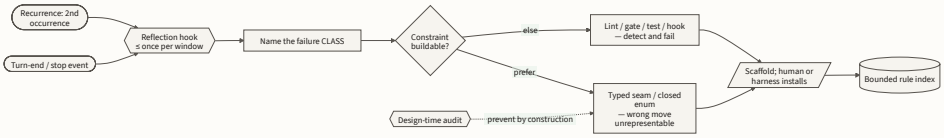


Figure 8.12-1: Accessible description: a turn-end event and a recurrence signal both feed a reflection hook that fires at most once per window — the hard, deterministic half. The hook prompts the soft loop: name the failure class, then decide whether a constraint is buildable; if so emit a typed seam or closed enum that makes the wrong move unrepresentable, else emit a sensor (lint, gate, test, hook) that detects and fails it. Either way the scaffolded control is handed to a human or the harness to install into the bounded rule index. A design-time audit prevents a class by construction before it is ever felt.

Full description → Self-governance (detect your own recurring issues; convert each into a tasteful control).

Intent — Give the system permission to govern the way it is governed: let it detect its own recurring issues and introduce *tasteful* — proportionate, right-sized — controls that prevent recurrence, rather than re-patching each instance by hand. When a failure recurs, classify the failure *class* and add the smallest durable guardrail that kills it, fired on a cadence so the loop runs by design (our instance: a loadable failure-interpretation skill invoked by a turn-end reflection hook at most once per window).

Problem

The recurring failure is re-patching an instance of a class the fleet will hit again. The same cherry-pick false-rejects a second time; the same lint mis-fires; the same manual step gets re-done by hand. Fixed locally each time, the class survives to bite the next agent. Two sub-failures compound it. Turning an instance fix into a class-killing control depends on an operator noticing the recurrence and choosing

to build the guardrail — the judgment skipped when the queue is deep. And even a team that believes in conversion forgets: on a long autonomous run the trigger lives only in fallible memory.

Mechanism

Two halves, one soft and one hard, packaged together.

- **The conversion loop.** On a recurrence, name the failure class, not the instance. Then pick the durable control from a small ordered vocabulary: prefer a *constraint* — a structured seam, a closed enum, an architecture that makes the wrong move unrepresentable — and fall back to a *sensor* — a lint, gate, test, or runtime hook — when no constraint can be built. The loop scaffolds the control; it proposes, it does not install.
- **The time-aware trigger.** A reflection hook bound to a runtime lifecycle event fires the loop on a cadence, at most once per window, asking one question: did a failure recur that should become a control? The deterministic trigger prompts the soft conversion judgment.
- **A design-time companion.** Run before a subsystem exists, the same stance audits it for predictive smells — shared mutable state, an irreversible operation, a duplicated fact — so a class need never be felt to be closed.

Engineering consequences

The conversion becomes cadence-driven rather than memory-dependent: the hard hook guarantees the *prompt* even on a deep queue, so the governance estate grows by design and velocity turns into durable trust instead of re-solving solved problems. The word *tasteful* carries weight — the loop adds the smallest guardrail that closes the class, so governing the system does not calcify it.

Implementation seam

The loop keys on a recurrence signal — memory, an incident log, an operator's recall — so a second occurrence reads as *seen before* rather than novel. The cadence half binds to a lifecycle event the harness exposes. A closed control vocabulary makes "pick the durable control" checkable, and each converted failure lands in a bounded, enforced home so the next conversion can see what already exists.

Known limitations

The proposing half is soft — it recommends and scaffolds, it cannot block. Cadence tuning is a real cost: too often and the reflection becomes alarm fatigue; too rarely and a recurrence ages past the moment it was cheapest to convert. Taste does not automate — choosing the right-sized guardrail, and resisting the over-control reflex, stays human. Left undisciplined, the loop can grow a thicket of low-value checks that themselves need governing.

Appendix B - 13. Composed state-machine model (typed lifecycles + cross-machine invariants)

The judgment — Model the composition, not the machines; the invariants that break live between them.

Role	Models-bridge
Family	System models
Used in stacks	The specification + verification stack
Enforcement	Hard
Related mechanisms	*Counterpart:* Process view (concurrent processes, lanes, and racing edges) *Sibling:* Agent-orchestration model (developer journeys) *Consumer:* Formal invariant verification (temporal form → model checking) *Layer:* Mediator & single-writer contracts

Full description → Composed state-machine model (typed lifecycles + cross-machine invariants).

Intent — Model a concurrent lifecycle as a *set* of structured state machines running at once, their cross-machine predicates declared as first-class, shape-verified invariants.

Problem

A distributed lifecycle rarely lives in one machine: a parent job fans out into chunks, each moves through its own states, and a completer fans the results back in. The properties that matter span the machines — a chunk is never both leased and free, output is uploaded before the row is marked done, exactly one completer fires. Left implicit, each is a scatter of boolean flags asserted nowhere, and the failures are the worst kind: a rare interleaving double-completes a job, a crash between upload and commit strands a corrupt output as "done." A suite that walks each machine alone reports green while the composition is broken.

Mechanism

- **Each lifecycle is one structured machine** — states as an enum, a transition table of legal edges, a terminal-safe set; an illegal transition is unrepresentable, a state no edge reaches a build finding.
- **The machines are named as a composed set** — the model declares the parent, per-chunk, and fan-in machines with their hand-off seams; the composition is the subject.
- **Cross-machine predicates are first-class invariants** — declared entities on the model, not comments or hopeful test names.
- **Each invariant declares a consumed temporal form,** routing safety to an exhaustive state-space search, liveness to a temporal checker, a linear ordering to a property test — so a hairy concurrency invariant is forced onto the strongest checker.
- **A drift gate holds it to reality,** reconciling the declared states against the live lifecycle vocabulary on every build; it lands audit-only, then promotes to blocking.

Engineering consequences

The lifecycle gains one authoritative source of truth: a new state or invariant is a model edit or the drift gate fails, and that friction is the freshness guarantee. The effort concentrates on naming the cross-machine predicates, the part a single-machine view never forces you to state.

Implementation seam

The model sits on the executable-source-of-truth substrate Appendix B, so the machines are data the build reads; each invariant carries a required, consumed temporal-form field; and at least one exhaustive checker must exist to route to. A separate verifier reads the invariants and runs the checker each form demands — this entry specifies, that one proves, neither useful alone.

Known limitations

A form no checker reads only looks verified, so the field must stay required and consumed. The check proves invariants across the modeled interleavings only; a bug the model abstracts away is out of scope, so the proof is only as strong as the model's fidelity. And it needs a real lifecycle enacted through addressable state to reconcile against — without that, this is a hand-drawn diagram, not a checked model.

Appendix B - 14. Control↔substrate dependency (computed blast-radius)

The judgment — Declare the assumption so the blast radius is queryable before you change anything.

Role	**Models-bridge**
Family	System models
Used in stacks	The governance-of-governance stack
Enforcement	Hard
Related mechanisms	*Enabler:* Deployment & tier topology *See also:* Meta-model consumption discipline (read, don't hardcode) *See also:* Model query surface (`repo-query`) *See also:* Drift & parity gates

The Structure of Control↔substrate dependency (computed blast-radius) — its shape at a glance:

Each substrate-reading mechanism declares its stance in a typed field. A query joins those declarations against the substrate model and, given a migration target, prints exactly which mechanisms a change puts in scope — the blast radius, computed before you touch the substrate.

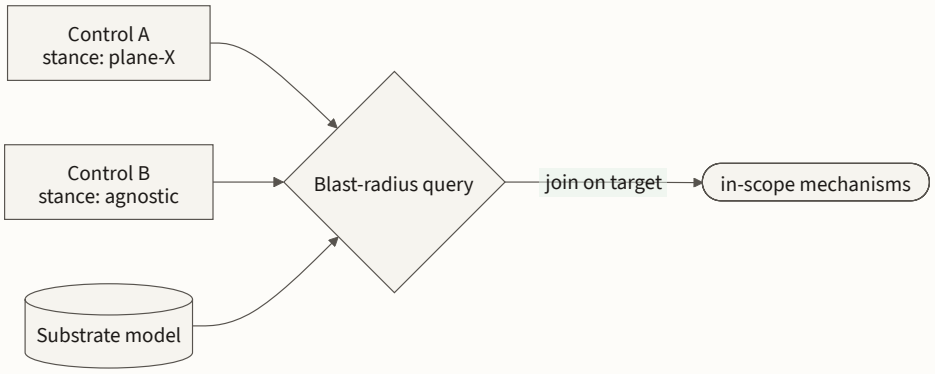


Figure 8.14-1: Accessible description: two mechanisms each declare a substrate stance. A query joins those declarations against the substrate model and, for a migration target, emits the table of mechanisms that change puts in scope. The table is derived, not hand-maintained.

Full description → Control↔substrate dependency (computed blast-radius).

Intent — Make each control *declare* the substrate assumption it bakes in as structured metadata, so "which controls depend on which part of the substrate, and what breaks when I change it" becomes a computed query rather than a grep-and-read. Before a cross-cutting substrate change, the static-analysis blast radius is known up front.

Problem

A control — a lint, a gate, a validator — usually bakes an assumption about the substrate it checks: a service is a Deployment under this directory, the manifest carries this field. The assumption sits buried in the control's body, invisible until you change the substrate. Then the change lands and the fleet fails two silent ways at once. In a **false FAIL**, a migrated service is validated against manifests or fields that no longer exist; on a no-baseline deploy gate, one buried assumption blocks every release.

In a **false PASS**, a migrated service drops its old annotation and vanishes from every totality, smoke, and disjointness check, looking clean because nothing checks it anymore — the worse of the two. The engineer planning the migration cannot see this coming: "which checks assume the old substrate?" is answerable only by grepping the importers, then reading each body.

Mechanism

- **Declare the assumption.** Extend the metadata block a control already carries with one structured field naming its substrate stance — a small closed enum (assumes-plane-A-only, assumes-plane-B-only, branches-per-plane, substrate-agnostic). A control that reads the substrate model must declare it.
- **Compute the join.** A query joins every declaration against the substrate model, emitting a table: per control, its assumption, the facts it reads, and, given a target substrate, whether a migration puts it in scope and whether it bakes the old assumption. That table is the blast radius, computed from declarations rather than grepped.
- **Enforce the declaration.** A lint fails any substrate-reading control missing its stance; it lands audit-only while the fleet is back-filled, then promotes to blocking. The deploy realization gate composes the guard and refuses to ship if any substrate consumer lacks a declaration.

The whole thing is a stable lint reading declarations: the table is derived from the fields at query time, so nothing is generated and no hand-maintained map can drift.

Engineering consequences

Every substrate-reading control gains one declaration — the intended tax, and what makes the dependency edge queryable. Grep finds the importers but not their assumption; lifting the stance into a declaration makes the importer and its posture toward the substrate one queryable fact, so the blast radius is computed before you touch the substrate.

Implementation seam

Four pieces: the closed enum of substrate stances, the structured field extending the control's existing metadata block, the join query that prints the blast-radius table, and the declaration lint the deploy gate composes. It requires the substrate already be a queryable model to join against Appendix B, and extends the read-don't-copy discipline from values to dependency edges Appendix B.

Known limitations

It pays off only at a substrate change; in a stable system the declarations sit inert, over-built if added speculatively to a substrate you will never change. A stance the closed enum cannot express forces an enum change — the honest signal that the substrate model itself grew a dimension — while an open string field would reintroduce the drift and typo class the enum exists to remove. And the existing controls must be back-filled before the lint can promote from audit-only to blocking.

Appendix B - 15. Coverage → model-node mapping (which invariants are actually tested)

The judgment — Measure coverage against invariants, not lines; the aggregate hides the gap.

Role	**Models-bridge**
Family	System models
Used in stacks	The specification + verification stack
Enforcement	Soft·Hard
Related mechanisms	*Counterpart:* Formal invariant verification (temporal form → model checking) *Enabler:* Executable source-of-truth models *See also:* Drift & parity gates

Full description → Coverage → model-node mapping (which invariants are actually tested).

Intent — Project test coverage onto the *model's nodes* — its states, seams, and invariants — so "is this invariant tested?" becomes a queried fact instead of a guess from a line-coverage percentage. The map turns the model into a test work-list: an invariant node with no covering test is a visible gap that drives the next test.

Problem

Line and branch coverage tell you what fraction of the code ran under test, not which of the system's invariants are actually exercised. A model can show 90% line coverage while a critical race invariant has zero tests touching its states, because coverage counts lines, not meanings. So the invariants you most need verified hide inside a high aggregate number, and "are we testing the thing that matters?" is unanswerable from the coverage report. The failure is a false sense of test adequacy: a green coverage number sitting over an untested critical invariant.

Mechanism

A mapping projects the suite's coverage data onto the model's nodes. Each node — a state-machine state, an IPC seam, an invariant — joins to the tests that exercise it, for instance at function granularity by naming which covered functions realize the node. The result is a per-node fact: covered, uncovered, or covered by which tests. Two uses ride on it: a backlog, where the uncovered critical nodes are the next tests to write and a sweep walks them; and, once promoted, a gate, where a critical invariant node with no covering test fails the build. It sits on the executable-source-of-truth substrate Appendix B.

Engineering consequences

The question becomes per-node — which invariants, states, and seams have a covering test, and which have none — instead of one aggregate a threshold satisfies while a specific invariant stays untested. The uncovered node is actionable: a concrete next test. Backlog-versus-gate is a judgment: gating every node starves throughput, gating none leaves criticals untested, and the criticality scope is the tuning surface between them.

Implementation seam

The pattern rests on the coverage-to-node mapping, the node-to-code join that attributes coverage to the right node, a criticality policy naming which nodes must be covered, and the two consumers — the backlog sweep and the promotable gate. It requires a structured model with addressable nodes and coverage data attributable to the code realizing each node.

Known limitations

The join is only as good as the node-to-code mapping: a node mapped to the wrong functions gets mis-attributed coverage. Coverage is not correctness — a covered node is exercised, not proven, so pair it with a proof mechanism Appendix B for invariants that need proof. Function-granularity coverage cannot separate two invariants realized by the same function.

Appendix B - 16. Drift & parity gates

The judgment — Enforce parity in both directions; a one-way check lets the model lie.

Role	**Models-bridge**
Family	System models
Used in stacks	The model-coherence stack
Enforcement	Hard
Related mechanisms	—

The Structure of Drift & parity gates — its shape at a glance:

The gate reads the model and enumerates reality, then checks the two set-inclusions that together make parity: model ⊆ reality (no phantom rows) and reality ⊆ model (nothing on disk goes unmodeled). Either direction failing turns the gate red.

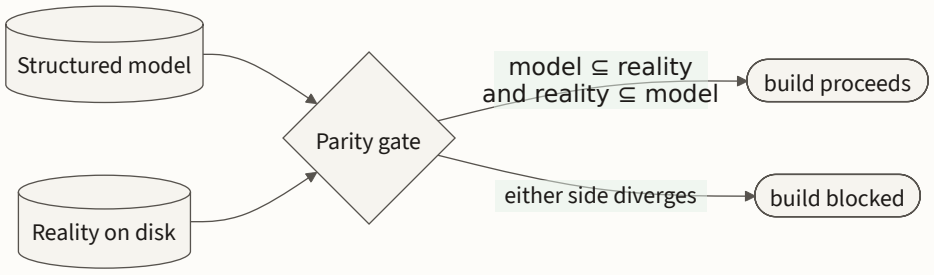


Figure 8.16-1: Accessible description: the parity gate takes two inputs — the structured model and the enumerated reality on disk. It checks both inclusions: every model row exists in reality, and every real thing is modeled. When both hold the build proceeds; when either side diverges the build is blocked.

Full description → Drift & parity gates.

Intent — A fleet of lints and tests enforces *bidirectional parity between each model and reality* — every model row maps to a real thing on disk, and every real thing maps to a model row — so a model cannot silently drift from the code it governs.

Problem

An executable model is only trustworthy while it stays true. The failure this kills is silent drift: the model says one thing, the code does another, and everything downstream — dispatch, code generation, deploy — reasons from a lie. Because the model *looks* authoritative, drift is worse than a missing model. It recurs whenever code changes without the model, or the model changes without the code.

Mechanism

Each model gets a gate — sometimes a pair — that reads the model at check time and enumerates reality, then asserts both inclusions: every model row exists on disk, and every real thing is modeled. A service-flow parity lint compares tree to spec both ways; an API-drift lint compares handler to contract; a lock lint compares each declared lock to a real lock site. Either direction diverging turns the gate red and blocks the build.

Engineering consequences

The model can now generate parts of the system — network policy, wiring — because the gate guarantees the generated side stays equal to the declared side. A one-way regenerate-from-code check cannot express that; it makes code the source of truth and leaves the model free to lie. Bidirectional parity is stricter, so it also fails on legitimate transitions. That is the point: a change must update both sides in the same commit.

Implementation seam

Each gate needs a machine-readable model, a machine-readable reality to compare it against, and blocking placement in the build. Where a lint can read the model file directly, prefer that over code generation, and generation over a hand-copied assertion.

Known limitations

Every model carries the cost of a gate to author and maintain, a real breadth of enforcement surface. A wrong parity predicate is its own hazard — it produces phantom drift that erodes trust, or false confidence that hides the real thing. And a model with no machine-readable reality to check against cannot be gated at all.

Appendix B - 17. Executable source-of-truth models

The judgment — Make authoritative system knowledge structured data, continuously read and held true.

Role	**Models-bridge**
Family	System models
Used in stacks	The model-coherence stack
Enforcement	Hard
Related mechanisms	*Counterpart:* Drift & parity gates

The Structure of Executable source-of-truth models — its shape at a glance:

The structured model sits at the center. Agents read it to reason; the build generates artifacts from it; a drift gate holds it equal to reality. Because it is read and checked on every run, it cannot go stale.

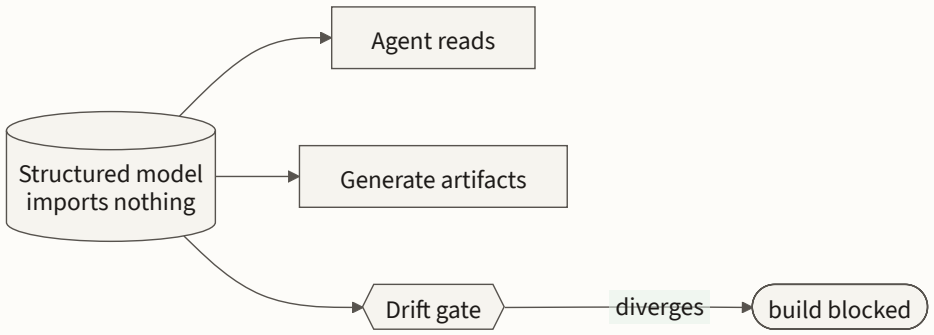


Figure 8.17-1: Accessible description: one structured model that imports nothing feeds three consumers — an agent that reads it, generators that emit real artifacts from it, and a drift gate that blocks the build when the model diverges from reality. Continuous reads and checks keep it from staling.

Full description → Executable source-of-truth models.

Intent — Model the system as structured data that tools read on every run and generate real artifacts from. The model becomes executable documentation that cannot drift, and the codebase becomes operable by a context-bounded agent — the interface through which that agent operates a context-exceeding codebase.

Problem

A large codebase exceeds any agent's context window; no agent can hold hundreds of thousands of lines. Left to read the raw code, an agent gets lost, re-derives the architecture badly, and drifts. The architecture itself lives only implicitly, scattered across the code, so humans re-derive it too. The failure is no shared, authoritative, compact representation of the system, which caps how large a codebase agents can operate on at all.

Mechanism

The model catalog holds structured models — a service dialect for some, structured loaders for the rest — that import nothing: pure data. Consumers read them at run and lint time, with a stable lint that reads the meta-file preferred over codegen, which is preferred over a hand-rolled copy. Every model

is pinned by a doc-derived characterization test, held true by a drift and parity gate, and frequently read or generated-from, so it is exercised constantly. Prose architecture docs drift because nothing forces them true; an executable model is read and checked continuously, so divergence surfaces as a failed build instead of a stale paragraph nobody reopened.

Engineering consequences

Because the models are continuously used and validated, they cannot go stale — the build fails the moment a model diverges from the code. A limited slice of artifacts, such as config, docs, and IPC contracts, is generated from the model, so those cannot diverge either. Why now: maintaining the models and satisfying the drift gates is tedious upkeep humans resent, but agents do that disciplined, repetitive regeneration without complaint, so agentic engineering finally makes model-based system engineering practical and lets an agent operate a codebase larger than its context.

Implementation seam

The catalog holds structured YAML, JSON, and loaders that import nothing; the preference order runs stable-lint-reads-meta-file over codegen over hand-rolled copy; each model carries a doc-derived characterization pin; and a drift and parity gate per model is the counted sensor that makes “cannot drift” true Appendix B.

Known limitations

Upkeep is real: the models must be maintained and the drift gates satisfied on every change, exactly the tedium that stops humans and the reason it needs agents. A wrong model is worse than none — an authoritative-looking model that has drifted misleads everything downstream, which is why the drift gates are not optional. And deciding what to model, and in what dialect, is design work up front.

Appendix B - 18. Formal invariant verification (temporal form → model checking)

The judgment — Prove an invariant by the exhaustive search its shape demands; sampling misses the race.

Role	Models-bridge
Family	System models
Used in stacks	The specification + verification stack
Enforcement	Hard
Related mechanisms	*Counterpart:* Drift & parity gates *Enabler:* Executable source-of-truth models

Full description → Formal invariant verification (temporal form → model checking).

Intent — Give every model invariant a temporal form — a temporal-logic operator saying whether it is *safety* (always holds) or *liveness* (eventually leads to) — and make that form the routing input that derives which exhaustive checker verifies it. An invariant is then proven by the method its shape demands, a state-space model-check rather than a sampled test, and it cannot be silently mis-verified.

Problem

Some invariants concern a single reachable state: "a job is never both leased and free." Others concern interleavings over time: "a preempted job eventually re-runs." Stated in prose, or pinned by one example test, either kind can be believed true while a rare interleaving violates it. A property test *samples* the input space and sails past the one adversarial schedule; a distributed race has failure traces no hand-picked example hits. Worse, a mis-declared liveness invariant gets routed to a safety runtime that structurally cannot see its violation, and reports nothing.

Mechanism

Each invariant records a temporal-logic form in standard operator syntax — always (safety), infinitely often, leads-to (liveness) — as a required field an invariant cannot be constructed without. From that operator the model derives the verification tier and the checker: a safety property routes to an exhaustive state-space search (a BFS over reachable states, or a model checker); a liveness property routes to a temporal model checker. A lint asserts the routed checker matches the operator, so a leads-to body must carry the leads-to token and a safety body must not. Reuse one mature formal engine rather than build a parallel one.

Engineering consequences

The check is exhaustive within bounds: it either proves the invariant over every interleaving of the modeled state space or returns a concrete counterexample trace, where a sampled test reports green on a schedule it never visited. The cost is weight — a model checker is not a per-commit gate, so it runs dev/CI-only, and the derived-tier routing reserves it for the hairy multi-actor races that earn it.

Implementation seam

The temporal-form field on each cross-service invariant plus the derivation from operator to tier; the exhaustive runtimes it routes to — a temporal model checker and a bounded-BFS “simworld” over the reachable state space; and the match lint that makes a mis-routed invariant a build error rather than a silent gap.

Known limitations

The proof is exhaustive only within bounds — the model is an abstraction, and a bug outside it is out of scope, so the guarantee is only as strong as the model’s fidelity to the real system. And the form must stay honest: a decorative temporal string no checker reads is worse than none, because it looks verified and isn’t.

Appendix B - 19. Governance graph (mechanism-interaction model)

The judgment — Model your controls as a system; catch guardrail collisions at authoring, not production.

Role	**Models-bridge**
Family	System models
Used in stacks	The governance-of-governance stack
Enforcement	Soft·Hard
Related mechanisms	*Counterpart:* Drift & parity gates *Sibling:* Agent-orchestration model (developer journeys) *Enabler:* Synchronization model (meta-sync) *See also:* Model query surface (`repo-query`)

The Structure of Governance graph (mechanism-interaction model) — its shape at a glance:

Nodes are typed mechanisms; the closed resource vocabulary is the join key. An edge joins two mechanisms that collide on one resource, typed by a four-value conflict taxonomy. Each edge's nature is *derived* from its conflict type: a decidable edge routes to a consistency lint, a semantic one to a human prompt. A drift lint re-resolves each node's code anchor to hold the graph equal to the wired reality.

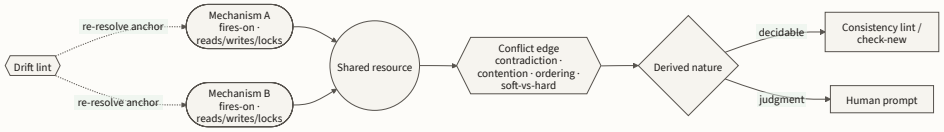


Figure 8.19-1: Accessible description: two mechanism nodes, each tagged by its firing event and its typed read/write/lock footprint, both touch one shared resource. Their collision is a typed conflict edge in a closed four-value taxonomy — contradiction, contention, ordering, soft-versus-hard. The edge's nature is derived from its conflict type: a mechanically decidable edge (contention, ordering) routes to a consistency lint or the pre-wiring check-new query; a judgment edge (contradiction, soft-versus-hard) routes to a human prompt. A drift lint re-resolves each node's code anchor so the graph tracks the wired mechanisms.

Full description → Governance graph (mechanism-interaction model).

Intent — Model the fleet's process-governance mechanisms themselves as a structured graph — each mechanism a node tagged by the event it fires on and the shared resources it reads, writes, or locks; each edge a *conflict* between two mechanisms over one resource — so a collision between two guardrails is caught by construction, not at the moment they trip each other in production.

Problem

A governed fleet accumulates guardrails: turn-end hooks, pre-commit checks, dispatch gates, host-level lock mediators. Each earns its place alone. But two can place contradictory or contending demands on one shared resource — a commit-set, an OS lock, the turn-end slot — and nothing sees the collision until a real operation trips both. It recurs as the fleet grows, because every new mechanism is a new pair against every existing one. Two case-study collisions show the shape: one path required a commit to take a squashed form while a separate scope-check flagged out-of-brief changes on that *same* commit-set; and two hooks fired on the *same* turn-end event with no declared

order, each able to block. Both are invisible in any single mechanism's code — they live in the interaction, and the interaction was drawn nowhere.

Mechanism

Each node names a mechanism and carries its firing event, its reads/writes/locks over a closed shared-resource vocabulary, its output power, and its soft-versus-hard enforcement. The resource vocabulary is the join key — conflicts flow *through* resources — and the turn-end slot and the context budget are resources too, so hook-pileup falls under the same machinery as lock contention. Edges are conflicts in a closed four-value taxonomy: contradiction, contention, ordering, and soft-versus-hard; each names the resource, its type, and a resolution, and whether the edge is decidable-by-machine or needs-judgment is *derived* from the conflict type. A query surface answers what fires on this event, what touches this resource, and — the load-bearing one — whether a *proposed* mechanism is consistent, run before it is wired. A drift lint re-resolves each node's code anchor and reddens when a mechanism is wired but absent.

Engineering consequences

The decidable conflicts route to a sensor; the semantic ones route to a human prompt, so a lint never has to decide whether two constraints on a commit-set actually compose. And checking a proposed mechanism against the graph turns "caught at collision" into "caught at authoring."

Implementation seam

The governance-graph model in the fleet's model layer, its four-value conflict taxonomy with a derived decidable-or-judgment nature, the check-new query that runs the deterministic checks against a proposed mechanism, and the drift lint that anchors each node to its wired hook or mediator by symbol, not line.

Known limitations

The graph must not drift — a stale interaction model claims conflicts are covered when a mechanism changed underneath it, so the drift lint is not optional. Resource granularity is a tuning surface: too coarse and every pair sharing a broad token looks like a conflict; too fine and the graph is noise. And the model describes, it does not mandate — it checks proposed mechanisms on request and hardens an edge into a lint only where a collision recurs, lest the governance become a tower nobody wants.

Appendix B - 20. Meta-model consumption discipline (read, don't hardcode)

The judgment — Derive the value at use time; a copied snapshot is a drift bug waiting.

Role	**Models-bridge**
Family	System models
Used in stacks	The model-coherence stack
Enforcement	Hard
Related mechanisms	*Consumer:* Dynamic context injection *See also:* Model query surface (`repo-query`)

Full description → Meta-model consumption discipline (read, don't hardcode).

Intent — Consume the models by querying them at runtime, never by embedding a hardcoded snapshot — so a lint, test, or brief always reasons from the live model, and a copied-out value can't drift behind the model it was copied from.

Problem

The models are a bridge only if consumers read them. The moment a consumer hardcodes a snapshot — "our packages are A, B, C" pasted into a lint or test — that copy drifts the instant the model changes, and the consumer keeps passing while reasoning about a stale world. This is the single most common substrate-drift vector: the model migrates, the copy is left behind, and the check now verifies the wrong thing. It recurs at every consumer that reaches for a quick literal instead of a query.

Mechanism

Consumers read the models at run- or lint-time — through the model query tool for agents and orchestration, by direct import for other tools — rather than embedding values. A preference order codifies it: a lint that *reads* the meta-file beats codegen, which beats a hand-rolled copy. A forward-policing lint fails a test that embeds a snapshot of a queryable value, and a further rule has lints declare their component tags against the component model rather than hardcoding scope.

Engineering consequences

There is one authoritative answer, and consumers derive it, so a model change updates every consumer at once. A snapshot instead mints a private answer at each site, and each is a drift bug the day the model moves. The cost is slight ceremony — a query call instead of a literal — plus a run/lint-time coupling: the consumer now depends on the model being loadable when it runs.

Implementation seam

The query surface consumers read through, and the snapshot-ban lint that fails a test embedding a queryable value. The meta-file-preference rule and the lint-scope-declares-against-the-model rule sit alongside as the same read-don't-copy discipline.

Known limitations

The ban-lint's accuracy bounds the whole discipline: it must recognise a queryable value to flag its snapshot, so it has to be built before it can be relied on as a live gate. And querying only helps where a read path exists — a value with no queryable model behind it has nothing to derive from.

Appendix B - 21. Model-derived test-obligation census (derive what should be tested, lint the gap)

The judgment — Derive the test-obligation set from the models; coverage can't see a test never written.

Role	**Models-bridge**
Family	System models
Used in stacks	The specification + verification stack
Enforcement	Hard
Related mechanisms	*Sibling:* Journey task-closure (type the terminal post-condition, derive its strength) *Enabler:* Fuzz campaigns (+ auto-coverage) *Consumer:* Executable source-of-truth models *Generalization:* Coverage → model-node mapping (which invariants are actually tested)

Full description → Model-derived test-obligation census (derive what should be tested, lint the gap).

Intent — Derive the set of things that *should* be tested from the structured models themselves — every external seam that should be fuzzed, every failure edge that should have an injection test, every invariant that should have a checker — and lint the gap between that derived obligation set and the tests that actually exist. Coverage stops being a percentage over lines you happened to write and becomes a walk over the model: an untested obligation is a named, listable finding, not an absence nobody notices.

Problem

Line coverage measures the code you wrote *and* tested; it is blind to the code you should have written a test for and didn't. The dangerous gaps are the ones nothing points at — an external seam never fuzzed, a failure edge with no injection test, a cross-service invariant with no checker. A percentage climbs toward a hundred while whole *categories* of obligation sit at zero, because coverage counts what exists and cannot count what's missing. And the obligation set is not static: every new seam, edge, and invariant adds one, and a coverage number never says "you added a thing that should be tested and didn't test it."

Mechanism

Walk the structured models that declare a testable surface: the seam registry yields fuzz targets, the error-path model yields injection obligations, the invariant model yields the checkers owed. Join each derived obligation against the test corpus by naming convention, tag, or registry. An obligation with no matching test is a finding — named, and attributable to the model element that generated it; and, in reverse, a test naming an obligation the model no longer declares is a rename-orphan finding, so the join is linted both ways. Because the set is *derived*, adding a seam adds an obligation, so a newly-introduced surface with no test reopens the gap until a test closes it. One derive-and-lint shape covers all three obligation kinds, rather than a separate hand-audit per kind.

Engineering consequences

A whole class of obligation sitting at zero becomes a listable set of findings, not a blind spot a rising percentage hides. The denominator flips from *what you built* to *what the models say should be tested* — the set a coverage report structurally cannot see.

Implementation seam

The censuses that derive the should-be-fuzzed and should-have-injection sets from the seam and error-path models, the same shape reused for invariant checkers, and the gate that treats an unmet obligation as a build finding — a derived list nobody checks is just another report.

Known limitations

The join must stay accurate: a test the census fails to match to its obligation reports a false gap, and a stale match reports false safety, so the matching rule tracks how tests are named and tagged. And it measures obligation coverage, not test quality — a matched obligation counts as covered even if its test is weak, so it closes the “no test at all” gap and leans on other mechanisms to judge test strength.

Appendix B - 22. Symbol-anchored traceability graph (derived edges)

The judgment — Derived edges defend; snapshotted ones drift — anchor joins to symbols and re-derive them.

Role	**Models-bridge**
Family	System models
Used in stacks	The model-coherence stack
Enforcement	Hard
Related mechanisms	*Counterpart:* Drift & parity gates *Enabler:* Executable source-of-truth models *See also:* Coverage → model-node mapping (which invariants are actually tested) *See also:* Control↔substrate dependency (computed blast-radius)

The Structure of Symbol-anchored traceability graph (derived edges) — its shape at a glance:

Every edge terminates on a `SymbolAnchor` — a resolvable `(path, symbol, resolver)` reference, carrying no line number — and joins two of five node genres (a model element, its lint, its code root, its proof, its registry) under a closed edge vocabulary. Each edge carries a non-optional derivation. A meta-lint re-resolves every anchor at check time and reddens on a broken one; the same anchors make the graph bidirectionally traversable.

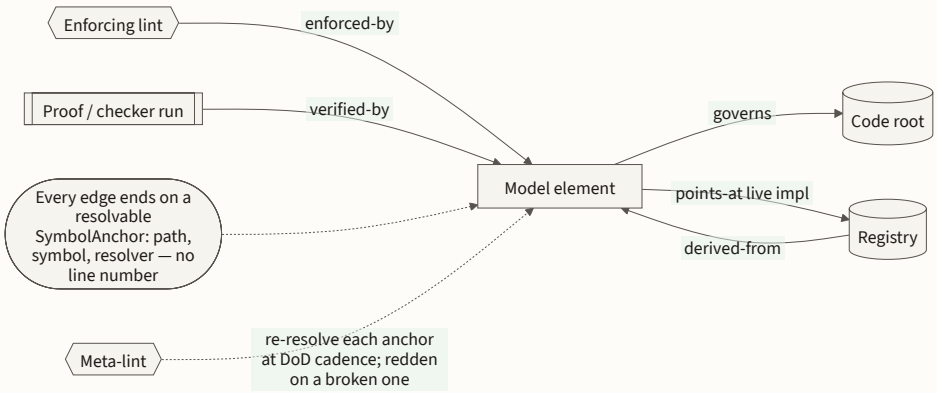


Figure 8.22-1: Accessible description: a central model-element node is joined to four other node genres by typed, closed-vocabulary edges — enforced-by its lint, governs its code root, verified-by its proof or checker run, derived-from its registry, and points-at the live implementation. Every edge terminates on a resolvable SymbolAnchor of (path, symbol, resolver) that carries no line number, so it survives a refactor above it. A meta-lint re-resolves each anchor at definition-of-done or audit cadence and reddens a broken one. The same resolving anchors let an agent traverse the graph both ways — code to model, or model to code — on one index.

Full description → Symbol-anchored traceability graph (derived edges).

Intent — Link every model to its lint, its code entry-point, its proof, its related models, and its registry as a structured graph whose every edge is a *derived* obligation a lint re-checks — each edge terminating on a resolvable *symbol*, never a line number — so when the code moves and breaks an edge, the model↔code drift becomes mechanically visible at scan time. The governing principle: derived edges defend; snapshotted ones drift.

Problem

The executable models that let a context-bounded agent operate a context-exceeding codebase are useful only while the map equals the territory. A model states more than facts about *itself*: it names the lint that enforces it, the code root it governs, the test that verifies it, the registry it reconciles against — the joins an agent walks between levels of abstraction, which rot the moment the code moves without them.

The failure is silent traceability rot: a model's reference to a code symbol goes stale when the symbol is deleted, renamed, or moved, and nothing notices — the model still *looks* authoritative while pointing at a ghost. The design was read off real drift: an audit of recently-closed work harvested roughly two dozen drift instances — the clean cases all read the source of truth at check time, the drifted ones kept a rotting parallel list.

Mechanism

- **Anchor to symbols, not lines.** Every edge terminates on a resolvable `(path, symbol, resolver)` reference with no line number — line numbers churn under every edit above them. The resolver is chosen by file extension — a language-aware analyzer for code, a membership check for registry keys, a heading for docs — so an anchor cannot silently prove a code symbol textually; a textual-presence fallback is a declared, visible weak edge, never accidental.
- **Structure the edges over a closed vocabulary.** An edge carries a source, a destination, an edge kind from a closed set (governs / enforced-by / verified-by / derived-from / points-at / related-to), and a non-optional derivation. A kind-pair table declares which node genres each edge kind may connect, so a mis-shaped edge is caught.
- **Re-derive every edge at check time.** A meta-lint walks the graph and, per edge, runs the edge's derivation against its target anchor and asserts it resolves. A vanished symbol reddens the edge.
- **Guard the anchors themselves.** One drift class needs its own guard: a *replacement* implementation is built but the surfaces naming which to run are never repointed, so the system defaults to the retired one. A registry declares, per seam, the live implementation and the census of pointer surfaces that must name it; a derived lint asserts they agree.
- **Walk the graph both ways.** The same anchors that catch drift make the graph a navigable cross-layer index. An agent at a symbol jumps up to the invariant it realizes; an agent at a model jumps down to the code — pulling just the relevant slice into context, not the whole tree.

Engineering consequences

Drift-detection and traversal are two faces of one property: an anchor that resolves means both that the model's claim is currently true *and* that the agent's traversal is a current slice of the system. A sharp by-product: a model referencing a symbol with no clean anchor — logic buried inline in a god-function — surfaces that absence as an abstraction-completeness finding routing to a refactoring target, not an error. Against a fan-out over twelve models, it classified roughly six-hundred anchors and caught about fourteen genuine drifts no existing lint fired on.

Implementation seam

The symbol-anchor reference and its per-extension resolvers, from static analyzers that already ship; the edge type with its closed kind vocabulary and kind-pair table; the re-derivation meta-lint,

landing audit-only then blocking; and the active-implementation registry with its pointer-agreement lint. Resolution is costly — a cross-reference round-trip per symbol over a large tree — so it runs at definition-of-done or audit cadence, a fast keyword companion catching the cheap cases inline.

Known limitations

Resolution catches deletion, not demotion: a symbol that still exists but no longer plays the role the edge claims resolves green, so the keyword companion for present-tense role-currency is the complement. A weak-prover fallback is a standing warning — a code anchor that resolves only by textual presence re-admits, if left un-burned-down, the drift the strong prover exists to remove. And the edge vocabulary must fit the domain: a relationship the closed kind set can't express forces an enum change, the honest signal that the join web grew a dimension, not a licence for a free-form string edge.

Appendix B - 23. Synchronization model (meta-sync)

The judgment — Declare locks and their ordering as a checkable model; invert one and fail at author time.

Role	**Models-bridge**
Family	System models
Used in stacks	—
Enforcement	Hard
Related mechanisms	*Counterpart:* Drift & parity gates *See also:* Mediator & single-writer contracts

Full description → Synchronization model (meta-sync).

Intent — A structured registry that models the system's synchronization behaviour: every OS-level lock, the shared resource it guards, and the required acquisition ordering, so concurrency contracts are declared and checkable rather than tribal.

Problem

A fleet of agents on one host contends over shared resources through OS locks — the test-runner serializer, the build semaphore, the whole-repo lint mutex, the commit serializer. Left undocumented, two failures lurk: an *undeclared* lock nobody knows guards what, and an *inverted acquisition order* between two locks that deadlocks. Both are invisible in the code and catastrophic at runtime, and they recur as new locks are added.

Mechanism

The registry composes three records — one per OS lock primitive (its path, its cap, its bypass, its audit log), one per acquisition site (or a declared "none" carrying a rationale), one per ordering constraint (a before/after edge with a rationale). A coverage lint scans the real lock call sites and requires each to be declared or carry a "not a sync lock" annotation. An ordering lint walks the declared edges plus the call graph to catch an inverted acquisition before the code runs.

Engineering consequences

A declared model lets a lint answer "which locks exist, and in what order must they be taken?" at author time, so an inverted acquisition fails then rather than deadlocking in production, where the lesson arrives too late to act on. The cost is that every new lock becomes a registry entry — an undeclared lock fails the coverage lint deliberately — and the ordering graph must be maintained, since a missing edge lets a real inversion through. Exempt sites carry a rationale drawn from a small, closed carve-out set.

Implementation seam

Two artifacts carry the pattern: the registry of lock, acquirer, and ordering records, and its two lints — the coverage lint over the real lock call sites and the ordering-constraint lint over the declared graph.

The model is induced from the code and reconciled at build, so it tracks the locks that exist rather than an aspirational list.

Known limitations

Coverage rides on the lint seeing every lock site; a site the lint cannot reach stays undeclared and unmodelled. The ordering graph is only as complete as its declared edges — an unstated ordering is an unchecked one. And this model is the OS-lock layer — which locks exist and in what order they are taken — not the higher-level *mediator & single-writer contracts* that declare who may run a call and how many at once; that is a separate DECLARE surface layered above it.

Appendix B - 24. PdfModel (sole PDF mutation surface)

The judgment — Make the bug class unrepresentable — one structured seam, raw API banned.

Role	Product
Family	Canonical models & seams
Used in stacks	The model-coherence stack
Enforcement	Hard
Related mechanisms	*See also:* Office Models ({Slides,Docs,Sheets}Model) *See also:* Canonical walkers (one traversal per tree)

Full description → PdfModel (sole PDF mutation surface).

Intent — Route *all* reads and writes of a complex file format through one structured model, with raw access to the underlying library banned by a lint, so the format's invariants are compiler-checked and every mutation passes through a surface that already encodes them.

Problem

The raw PDF library is a minefield of failures invisible at the call site. Forget to mark an indirect object modified and the write is silently dropped on save. Add a tag or write a dictionary key directly and you can corrupt the structure-tree root — the exact class that produced a real production corruption. Nothing enforces these invariants in one place, so scattered raw calls make the same bug class recur at every site.

Mechanism

Callers read through one entry point and write through typed mutators. A ban-lint fails the build on any raw constructor, tag insertion, or dictionary write. Each typed mutator wires the mark-modified discipline and attribution stamping, so the invariants ride on the model and a new verb cannot land un-wired.

Engineering consequences

The bug becomes unrepresentable, not merely discouraged: the raw API is unreachable, so a reviewer no longer has to spot a missing mark-modified call in a diff that looks correct. The model is the construction that removes the class; the ban-lint is the counted sensor that keeps every call site on the seam. The seam also pins the library version, because a minor bump can silently change auto-tagging.

Implementation seam

Two artifacts carry the pattern: the typed mutators under one primitives module, and the ban-lint on the raw API. Standing up the seam means migrating every existing call site, then holding the line with the lint.

Known limitations

The model must cover the whole surface callers need; a missing operation forces either a lint escape (a hole) or a model extension (friction, but the right fix). Pinning the library for tag-tree stability makes upgrades deliberate, gated work. The mutators plus the ban-lint are code to keep current as the format's needs grow.

Appendix B - 25. ContentValidator (input ⊆ output fidelity)

The judgment — Guard silent failures with a machine-checked post-condition, not a spot-check.

Role	Product
Family	Validation & conformance
Used in stacks	The provenance + fidelity stack
Enforcement	Hard
Related mechanisms	*See also:* Cross-source coherence lints

Full description → ContentValidator (input ⊆ output fidelity).

Intent — A deterministic gate asserting that every piece of the user's original content survives remediation (input content ⊆ output content), run in production, with a per-pass staging variant that pinpoints which pass dropped content.

Problem

Remediation mutates the document across many passes and four formats. A bug in any pass could silently drop or alter user content: a deleted paragraph, a mangled table, a lost caption the author never sees go. For a fidelity-critical tool that is the worst possible outcome — the output looks fine and quietly isn't what the author wrote. The failure recurs on every remediation pass.

Mechanism

The gate extracts the input's content and asserts it is a subset of the output's, checked mechanically on every production job and failing the job on violation. A staging-only per-pass variant, gated by an environment label, emits a dedicated fidelity marker and a nonzero exit code, so the offending pass is identified before delivery. That turns "content was lost somewhere" into "pass N lost it."

Engineering consequences

The guarantee becomes a deterministic post-condition rather than trust in the mutation code or a human spot-check that misses *silent* drops — you don't notice the paragraph that's gone. A post-condition that fails the job takes the trust out of the loop. The costs are concentrated in two places. Everything rests on the extractor: lossy or over-eager extraction yields false positives that block good output, or false negatives that miss a real drop. And subset semantics are subtle — reordering, whitespace, and reformatting must be normalized or the gate cries wolf. It runs on every job, an accepted cost.

Implementation seam

The production fidelity gate runs post-remediation and pre-delivery; the per-pass hook adds localization in staging. Both need a content extraction comparable across input and output, and a subset predicate that tolerates legitimate reformatting and reordering without firing.

Known limitations

The extractor defines what “content” means, so the guarantee is exactly as complete as the extraction — anything it doesn’t extract, it can’t protect. The subset predicate must tolerate legitimate reordering without false positives, or it blocks correct output. And the production gate detects a loss without localizing it; only the staging per-pass variant names the pass that caused it.

Appendix B - 26. Blocking semantic lints

The judgment — Move the audit into a lint — mechanize the invariant the compiler can't express.

Role	Product
Family	Validation & conformance
Used in stacks	The specification + verification stack
Enforcement	Hard
Related mechanisms	*See also:* Cross-source coherence lints

The Structure of Blocking semantic lints — its shape at a glance:

Each lint scans the source for its invariant and reports findings. A blocking lint fails the build; a scoped, reason-bearing suppression comment escapes a legitimate exception.

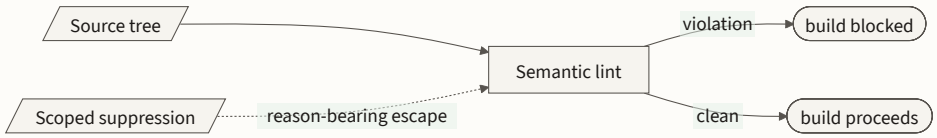


Figure 8.26-1: Accessible description: a semantic lint scans the source tree for its invariant. A violation blocks the build; a clean scan lets it proceed. A scoped, reason-bearing suppression comment escapes a legitimate exception on a single line.

Full description → Blocking semantic lints.

Intent — A fleet of blocking semantic lints over the tool's own source — banned APIs, silent-catch bans, diagnostic-console bans, structured-seam violations — that fail the build on domain-invariant violations the compiler and review can't catch.

Problem

The codebase carries hundreds of structural invariants: no silent catch, no banned API in prod, every cross-boundary call through its seam. Review cannot hold hundreds of invariants in a reviewer's head, and the compiler enforces none of them — a silent catch, a banned API, a raw diagnostic-console call all compile fine. The failure is structural drift that quietly reintroduces a defect class, and it recurs continuously as code is written.

Mechanism

The lint fleet runs at commit and deploy. Each lint declares, in a self-describing header, which components it scopes to, its severity, and a verb-of-checking docstring; blocking lints fail the build, audit-only ones surface. A legitimate exception escapes through a scoped, reason-bearing suppression comment on the offending line. The fleet sits atop a maxed-out commodity floor — the platform's own static analysis, strict type-checking, and fast linters — rather than replacing it.

Engineering consequences

A semantic lint encodes an invariant the type system can't express and fails the build, doing mechanically what review does by attention and the compiler doesn't do at all — the "move audits to lints"

discipline made concrete. The costs are real. Hundreds of lints are hundreds of things to keep current. A too-strict lint blocks legitimate code until a suppression is added, a small audited hole. And the custom fleet only earns its keep atop the commodity floor; without that floor, the quality grade rests on the wrong thing.

Implementation seam

A lint framework with per-lint scope and severity declaration, the invariants made mechanically detectable, and a runner wired into the gates with a scoped escape hatch. A runnable example carries the whole shape: the self-describing declaration block, the find-violations then emit then exit-code body, the suppression escape, and the audit-only-to-blocking migration.

Known limitations

An invariant you can’t express as a mechanical check can’t join the fleet. A lint is itself code that can fail: one built on a regex once backtracked catastrophically on a real input and hung the deploy gate — the checker that guards the fleet became the thing that stalled it. The cure was not a better regex but deleting the surface, reaching for the parser and letting the whole class go. A checker over structured source belongs on a parser, not a pattern.

Appendix B - 27. Fuzz campaigns (+ auto-coverage)

The judgment — Fix to the spec point, not the seed; real producers beat random bytes as an adversary.

Role	Product
Family	Regression tests
Used in stacks	—
Enforcement	Hard
Related mechanisms	*See also:* FsCheck property tests

The Structure of Fuzz campaigns (+ auto-coverage) — its shape at a glance:

The campaign generates malformed inputs and runs the tool against them; coverage is collected automatically and compared to a baseline. A finding routes to a fix aimed at the spec point, not the individual seed.

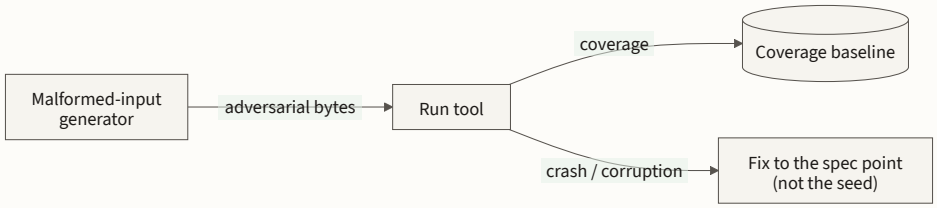


Figure 8.27-1: Accessible description: a generator feeds adversarial bytes to the tool; coverage is collected and compared against a baseline so reach is measurable. A crash or corruption routes to a fix aimed at the stable spec point rather than the individual failing seed.

Full description → Fuzz campaigns (+ auto-coverage).

Intent — Campaigns that feed malformed and adversarial inputs to the tool to find crashes and corruption, with coverage collected automatically, plus an RCA discipline that fixes to the specification, not the failing seed.

Problem

Real-world documents are malformed in ways no hand-written test anticipates: a truncated stream, an odd encoding, a structure right at the edge of what the spec allows. A fuzzer finds the crash or corruption on inputs you would never think to write, hiding across an input space far too large to enumerate.

Mechanism

Fuzz harnesses run the tool against generated malformed inputs. On a finding, the RCA discipline fixes to the stable point in the format spec, not the seed, so the fix passes every spec-allowed input, not just the one that crashed.

Two moves sharpen this beyond random bytes:

- **The producer-dialect corpus.** The failure space that matters is producer diversity, not pure randomness. A format has one specification but many independent writers, each emitting its own accent within the grammar — a rival office suite, a different PDF writer, a conversion tool, a

from-code generator — producing legal-but-unusual object orderings your own writer never emits. Round-trip a real document through a genuine third-party producer and let its dialect be the adversarial input: no mutation engine needed, the diversity is already out there. This reaches the class that hurts in production — a user uploads a file from a tool you never tested.

- **The model as oracle.** Plain fuzzing has a coarse oracle — never crash, never corrupt — with no declared notion of a correct answer on a wild input. A structured model already names a stable spec point: a closed set of legal outcome classes, an invariant predicate, a state-transition table. Point the wild inputs at the model's entry surface and judge the result against that set. A clean structured rejection of an illegal input passes; an outcome outside the set — an unexpected exception, a corrupted structure, a forbidden transition, a false invariant — fails.

Engineering consequences

The two moves compose: feed the producer's wild dialect to the model's entry point and classify the result against the model's declared outcome set. Wild input and a rich oracle at once — and a fix that holds for the whole format, because the oracle is the model's own declaration, so a fix aimed at its stable point closes every input the specification allows rather than the one seed. This is the RCA discipline expressed structurally: the model is the stable spec point written down, so RCA-to-the-spec-point and judged-against-the-model become one move.

The synthesis scales from formats to concurrency. There the input is an interleaving of concurrent steps, the generator is an interleaving-fuzzer or an exhaustive walk over reachable states, and the oracle is an invariant predicate over the model's states — no two workers hold the same lease, a job never leaves a terminal state, a queued item is eventually served. Naming the predicate points the search straight at the interleaving that violates it, a defect a strong-but-static unit suite walks right past. Because the oracle is a declared property and not a per-seed check, a campaign that finds nothing is proof-shaped — one linear-invariant campaign cleared 200 adversarial inputs with the invariant holding on every one, and the same technique caught a real zip-bomb, a never-raise-contract breach, and four latent parser crashes before the model-derivation half landed.

Implementation seam

Fuzz and campaign harnesses plus a corpus of malformed inputs — random and adversarial bytes, and for the sharper corpus a set of real third-party producers of the format to round-trip through. A coverage collector, aggregator, and baseline make reach measurable. The model-as-oracle form additionally needs a structured model that declares the stable spec point — a closed legal-outcome set, an invariant predicate, or a state-transition table — for the wild input to be judged against.

Known limitations

Campaigns cost real compute; coverage is tracked to know when they have saturated, and the baseline must be re-based only on intentional coverage-shape changes. Seed-fixing is the anti-pattern the RCA discipline exists to prevent — patching only the failing input leaves the spec-class open. And the model-as-oracle form is only as good as the declared outcome set: an outcome the model never named as legal or illegal escapes judgment entirely.

Appendix B - 28. Per-mutator attribution stamps

The judgment — Make provenance reconstructible from the artifact, not merely logged.

Role	Product
Family	Provenance & attribution
Used in stacks	The provenance + fidelity stack
Enforcement	Hard
Related mechanisms	*Counterpart:* F10 mutator-stamp-wiring lint *Consumer:* `derive-changelog` (reconstruct mutations)

Full description → Per-mutator attribution stamps.

Intent — Every operation that mutates a document embeds an attribution stamp *in the artifact itself*, written through one sanctioned stamp-writer per format, so any change stays attributable and the mutation history can be reconstructed after the fact.

Problem

When a remediated document comes out wrong, you need to know which pass made which change. Without attribution, a mutation is anonymous: the output is visibly broken, but nothing says who wrote it. Root-cause analysis becomes guesswork across many passes and four file formats, and the failure recurs on every mutation.

Mechanism

Each format exposes one stamp-writer, and the raw stamp mutator is lint-banned so the writer stays the sole surface. PDF passes stamp through a stamp-writer helper; the Office formats stamp through an append-only attribution registry. A visibility model keeps delivery honest: stamps default to a debug tier and are stripped before the document ships, while user-visible passes opt into a preserved tier that stays in the output.

Engineering consequences

Attribution lives with the artifact, not in a log that scrolls away, so a changelog tool can rebuild the full attributed history from the delivered file at any time. The cost is document overhead — debug stamps add content, which is why the strip step removes them before delivery. Bypassing the helper yields a non-uniform stamp, so the ban-lint holds the single surface.

Implementation seam

The stamp-writer helper (PDF) and the attribution registry (Office) are the two wiring points; a raw-mutator ban-lint fails the build on any call that skips them. A separate wiring lint makes it blocking that *every* remediation verb stamps, so a new verb cannot land unattributed.

Known limitations

Completeness rides on that wiring lint: a verb the lint does not cover can mutate silently. The debug/preserved split is an authoring decision, so a pass that picks the wrong tier either leaks scaffolding

into delivery or loses attribution the operator wanted. And the stamp is only as trustworthy as the strip step that runs before the document leaves the pipeline.

Appendix B - 29. Closed remediation-verb sets

The judgment — Narrow the action space to a closed vocabulary instead of reviewing arbitrary output.

Role	Product
Family	Repair vocabulary
Used in stacks	—
Enforcement	Hard
Related mechanisms	*See also:* Typed `ViolationCategory` / `FailureCategory` enums

Full description → Closed remediation-verb sets.

Intent — Route the remediator's mutations through a bounded, named set of structured verbs — a closed mutator layer — rather than free-form edits, so the move-space is enumerable and every move can be stamped, validated, and policy-checked (our instance: the document models' `Primitives`).

Problem

Free-form document editing is unbounded: any pass could do anything, and an unbounded edit cannot be stamped, validated, or checked against policy as a set. The failure is an unbounded mutation that is un-attributed, un-validated, or off-policy, and it recurs per pass unless the move-space itself is constrained.

Mechanism

The `Primitives` mutators are the sanctioned verb set, and the masked-pass architecture routes all changes through them. A closed set makes the move-space enumerable: every verb wires an attribution stamp Appendix B, every insert registers with the inserted-content registry, and the fidelity gate Appendix B covers the outcome. A new verb must wire its stamp and, if it inserts, register — nothing mutates the document outside this set.

Engineering consequences

A bounded, named action-space is what makes attribution, validation, and policy tractable at all. Free-form mutation leaves the governance questions — is every mutation stamped? is every insert validated? are all moves on-policy? — unanswerable, because the set of possible moves is open. The cost is friction of the intended kind: a needed action absent from the set forces adding a verb, which keeps the space closed and every move governed. The verb set is a maintenance surface that grows with remediation capability.

Implementation seam

Three parts carry the pattern: the structured mutator layer all mutation routes through, the stamp and validate wiring each verb carries, and the lints that hold no change happening outside the verb set.

Known limitations

The guarantee holds only while every mutation truly routes through the set; a raw edit that escapes the layer is unbounded again, so the closure is exactly as strong as the discipline and lints that enforce it. And the set must grow to cover each new remediation capability, making completeness an ongoing obligation rather than a one-time closure.

Appendix C

Mechanism Catalog

Appendix C — Mechanism Catalog

How to use this catalogue

This appendix is a reference manual, not a chapter. Browse it; you are not meant to read it straight through. Each mechanism appears in its smallest useful form — a diagram, a short engineering summary, and a line of metadata — so you can recognize a mechanism and find it fast.

The chip line under each mechanism tells you four things at a glance: its **family**, its **primary concern**, whether it enforces **softly or hard**, and whether it is **essential** (expected in nearly every Governed Engineering Environment) or **specialized** (worth adopting when your domain calls for it). Essential and specialized mark expected applicability, not quality — a specialized mechanism is not a lesser one.

When a mechanism is central to MAGE, its Engineering Note in Appendix B carries the judgment behind it, and the brick points you there. The full online catalogue holds the complete documentation, implementation guidance, and extended examples.

Surface	Purpose	The reader's question
Appendix A	Reusable engineering architectures	*How do these mechanisms work together?*
Appendix B	Engineering judgment	*Why would an experienced engineer build it this way?*
Appendix C (this appendix)	Mechanism catalogue	*What mechanisms exist, and where do I find them?*
Online catalogue	Complete documentation	*I want the full entry — implementation, examples, the rest.*

Reading a brick. Under each mechanism: **Family · Primary concern · Enforcement · Applicability**. *Enforcement* — **Hard** blocks, **Soft** guides, **Soft·Hard** ships a soft aim with a hard sensor. *Applicability* — **Essential** (expected in nearly every environment) or **Specialized** (adopt when the domain calls for it); this marks expected fit, not quality. A **technique** brick leads with the transferable pattern name and points to its advanced examples; an **instance** brick names the technique it folds under, loudest for the document-accessibility instances whose transferable lesson is the technique. A **flagship** mechanism also carries an **Engineering Note** link to its deeper discussion in Appendix B.

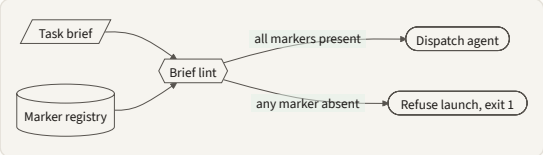
The techniques. 24 canonical techniques organize the catalogue. Every *instance* brick names the technique it folds under; every *technique* brick lists its advanced examples. Here are the techniques, by number of in-book examples:

Executable Source of Truth (19) · **Staged Admission Gates** (5) · **Caused-By Provenance** (4) · **Model-Derived Assurance Coverage** (4) · **One Door Enforced** (4) · **Closed Action Vocabulary** (3) · **Drift / Parity Gate** (3) · **Mediated Resource Admission** (3) · **Point-of-Action Policy Delivery** (2) · **Read the Model, Don't Copy It** (2) · **Validated Dispatch** (2) · **Authoritative Lifecycle State** (1) · **Encoded Operational Judgment** (1) · **Fleet Observability Surface** (1) · **Generative Validation** (1) · **Governance Graph** (1) · **Governed Knowledge Base** (1) · **Composed State-Machine Model** · **Computed Control Blast Radius** · **Derived Traceability** · **Machine-Enforced Semantic Policy** · **Preservation Invariant** · **Re-Derived Definition of Done** · **Self-governance**

C.1 Agent

Agent — the fleet that produces the work, and the substrate that runs it — how agents are dispatched, isolated, gated, and observed.

30 mechanisms. Each brick links to its full Gang-of-Four entry in the online catalogue; a flagship also carries a deep-dive note in Appendix B.



TECHNIQUE · Validated Dispatch

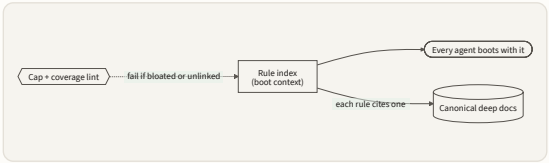
Brief-linting

Statically lint an agent's task brief *before the agent is spawned*, refusing to launch any brief missing the markers that make the agent's work safe and well-scoped.

Advanced examples → Epic & design-doc templates, Mandatory snippet-table enforcement

Context And Dispatch · Admission · Hard · Essential

Engineering Note → B.1

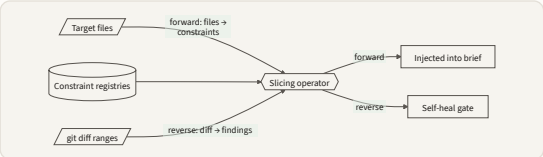


Docs hierarchy + governance index (online)

An instance of: Governed Knowledge Base →

Give every agent a single, numbered, cross-referenced **rule index** (loaded into its boot context) that points at the canonical deep doc for each rule, so the fleet shares one authoritative map instead of re-deriving invariants from scattered files.

Context And Dispatch · Context · Soft·Hard · Essential



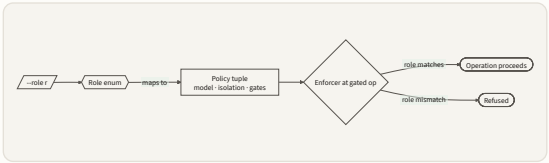
Dynamic context injection

An instance of: Point-of-Action Policy Delivery →

Map the files an agent is about to touch to the *exact constraints that govern those files* (lints, conventions, component boundaries, tests) and inject that subset into the agent's brief **before it writes code**, moving detection left of the cheapest CI gate.

Context And Dispatch · Context · Soft · Essential

Engineering Note → B.2

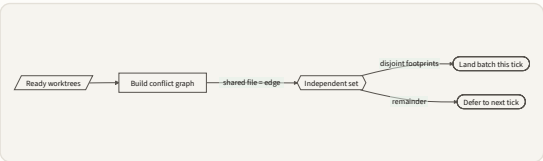


Role-typed dispatch (online)

An instance of: Closed Action Vocabulary →

Dispatch every agent under a **typed role** (`sonnet-active` / `opus` / `lint-runner` / `commit-slave`) that determines its LLM, isolation mode, permissions, and which gates apply, so those choices are policy-by-type, not a per-dispatch judgment call.

Context And Dispatch · Constraint · Hard · Essential

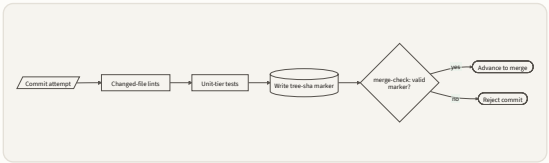


Merge-train MIS batching (online)

An instance of: Staged Admission Gates →

A merge-train that lands the largest set of *non-conflicting* agent worktrees per tick by computing a **Maximum Independent Set** over their file footprints, so many agents' work merges in one conflict-free pass instead of thrashing sequentially.

Gates And Merge Train · Admission · Hard · Specialized

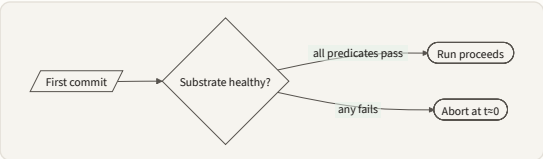


Pre-commit hook (3-stanza, tree-sha markers) (online)

An instance of: Staged Admission Gates →

A three-stanza pre-commit hook that runs changed-file lints and unit-tier tests and writes tree-sha-keyed marker files, so an agent's commit cannot advance to merge unless the cheap checks actually passed on *exactly this tree*.

Gates And Merge Train · Admission · Hard · Essential

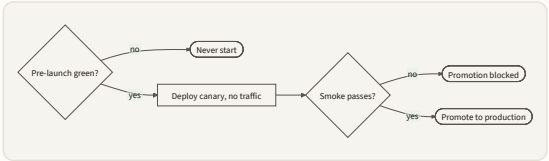


Sentinel first-commit early-abort (online)

An instance of: Staged Admission Gates →

A health check on an agent's *first* commit that surfaces orphaned-worktree and broken-substrate failures at minute zero, before the agent burns its whole budget producing work that can never land.

Gates And Merge Train · Admission · Hard · Specialized



TECHNIQUE · Staged Admission Gates

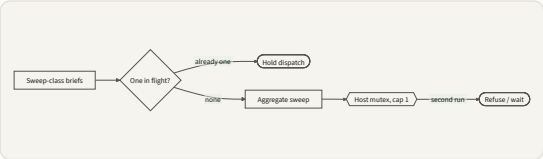
Staged deploy gates (canary → smoke → promote)

A deploy pipeline that escalates **canary → smoke → promote**, blocking promotion to production until each cheaper stage passes on a traffic-free revision, so a bad build is caught before users see it, not after.

Advanced examples → Merge-train MIS batching, Pre-commit hook (3-stanza, tree-sha markers), Sentinel first-commit early-abort, Cron-alerts gate, Test-onion tiers (Smoke / Lite / targeted / full)

Gates And Merge Train · Admission · Hard · Essential

Engineering Note → B.3

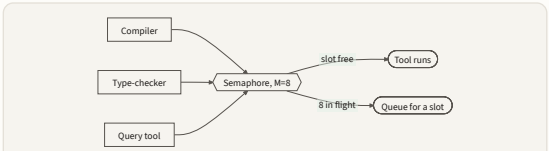


Aggregate-compute protection (`lint-all` host mutex) (online)

An instance of: Mediated Resource Admission →

A one-per-host mutex on `lint-all` (the aggregate lint sweep) plus a one-in-flight declaration per orchestrator, so the single heaviest compute job cannot run twice on a machine or be triggered by many agents at once.

Mediators And Resource Locks · Orchestration · Hard · Specialized

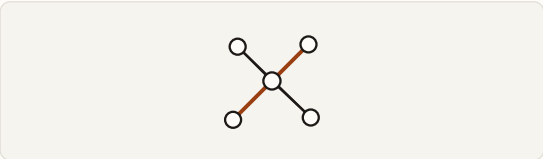


Build-serializer (M=8 semaphore) (online)

An instance of: Mediated Resource Admission →

A host-level **counting semaphore (M=8)** over the adjacent heavy-compute tools (`dotnet build`, `tsc`, `csharp-query`, jedi lints, `pyright`), so concurrent worktrees get parallelism up to the machine's capacity without oversubscribing it.

Mediators And Resource Locks · Orchestration · Hard · Specialized



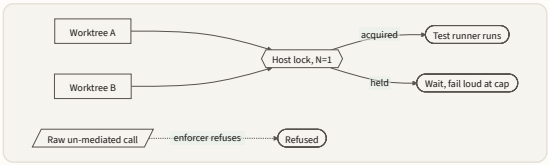
Resource-pressure gating (admit before, shed during)

An instance of: Mediated Resource Admission →

Govern a saturable host resource with one live pressure signal read at two layers: an admission gate deferring heavy work before dispatch, and an execution shed stopping it when pressure spikes. Heavy work is never started into, or left running on, an overloaded host.

Mediators And Resource Locks · Orchestration · Hard · Specialized

Engineering Note → B.4



TECHNIQUE · Mediated Resource Admission

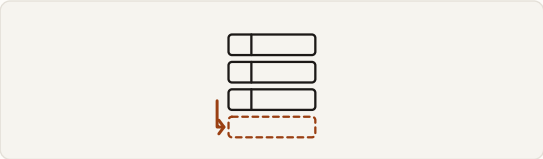
Test-serializer (N=1 flock on `dotnet test`)

A host-level wrapper that serializes `dotnet test` to a **single writer** via an exclusive flock, so concurrent agent worktrees on one machine don't saturate I/O and interfere with each other's test runs.

Advanced examples → Aggregate-compute protection (`lint-all` host mutex), Build-serializer (M=8 semaphore), Resource-pressure gating (admit before, shed during)

Mediators And Resource Locks · Orchestration · Hard · Specialized

Engineering Note → B.5



TECHNIQUE · Authoritative Lifecycle State

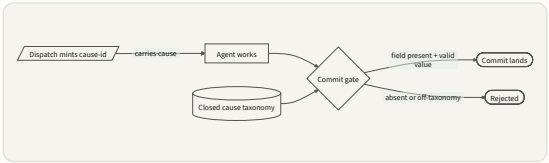
Agent registry (append-only log + marker cache)

An append-only registry, dual-written by every lifecycle tool, that is the *authoritative* record of which agents are in flight, so cleanup, tombstone, and merge decisions read a fact instead of guessing from filesystem timestamps.

Advanced examples → Tombstone commits (lifecycle close records)

Lifecycle And Observability · Orchestration · Hard · Essential

Engineering Note → B.6

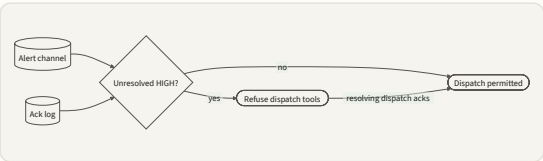


Caused-by provenance (agent-side change traceability) (online)

An instance of: Caused-By Provenance →

Traces every agent-driven change back to the reason that caused it, drawn from a closed taxonomy and enforced as a field at the commit gate. Correlation is upgraded to causation when the cause is known at the moment of action; the trace exists to explain, not to be optimized.

Lifecycle And Observability · Provenance · Hard · Essential

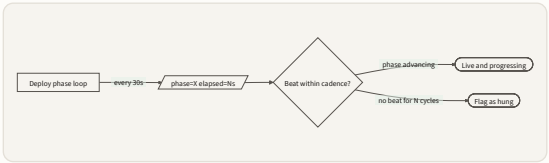


Cron-alerts gate (online)

An instance of: Staged Admission Gates →

A gate that **blocks new orchestrator work-dispatch** while an unresolved HIGH-severity cron alert exists, forcing the orchestrator to ack or resolve it before piling more work onto a possibly-broken substrate.

Lifecycle And Observability · Admission · Hard · Specialized

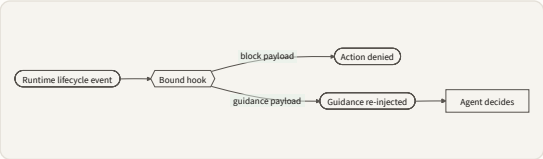


Deploy heartbeats + stale-worker detection (online)

An instance of: Fleet Observability Surface →

Periodic `[heartbeat] phase=X elapsed=Ns` emissions from long-running deploys, plus a stale-worker sweep, so a *hung* deploy or worker is distinguishable from a merely *slow* one.

Lifecycle And Observability · Orchestration · Hard · Specialized



TECHNIQUE · Point-of-Action Policy Delivery

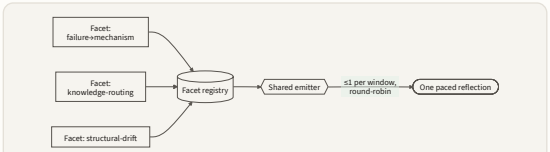
Lifecycle hooks (interpose on the agent runtime's events)

Bind a script to the agent runtime's lifecycle events (turn-stop, pre-compaction, session-start, before-a-tool-call) so a step the operator keeps *omitting at runtime* fires deterministically, whether or not anyone remembered it.

Advanced examples → Dynamic context injection, Reflection-facet substrate (tempo-gated policy nudges)

Lifecycle And Observability · Orchestration · Soft·Hard · Essential

Engineering Note → B.7

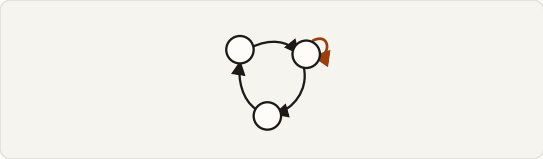


Reflection-facet substrate (tempo-gated policy nudges) (online)

An instance of: Point-of-Action Policy Delivery →

Consolidate an operator's policy-reflection nudges into one tempo-gated substrate: a registry of facets, each checking context against a single policy dimension it references, never copies. The family emits at most one reflection per window, preventing alarm fatigue.

Lifecycle And Observability · Orchestration · Soft·Hard · Specialized

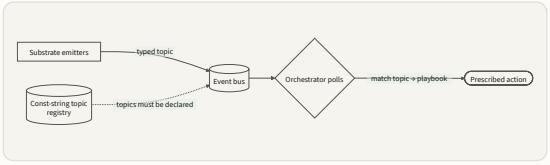


Tombstone commits (lifecycle close records) (online)

An instance of: Authoritative Lifecycle State →

A tombstone commit written at a worktree's branch tip that durably records its lifecycle close and *disposition* (cherry-picked / declared-skipped), so cleanup can *prove* a worktree is safely reclaimable instead of guessing.

Lifecycle And Observability · Orchestration · Hard · Specialized



TECHNIQUE · Fleet Observability Surface

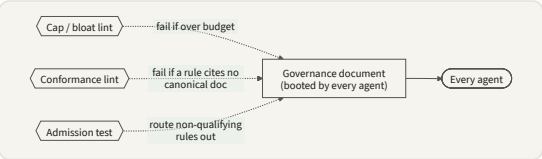
Orchestrator-as-reactor over an event bus

A structured event bus with a closed topic registry and a companion playbook, over which substrate emits lifecycle and health events. It turns the orchestrator into a reactor that reads fleet health from a queryable signal surface and responds with a prescribed action.

Advanced examples → Deploy heartbeats + stale-worker detection

Lifecycle And Observability · Orchestration · Hard · Essential

Engineering Note → B.8



TECHNIQUE · Governed Knowledge Base

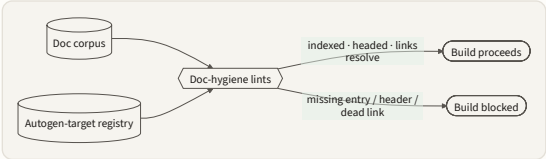
CLAUDE.md rule index (the governance document as a mechanism)

Treat the top-level governance document as enforced infrastructure: a stable-numbered rule index loaded into every agent's boot context. It is held honest by its own enforcement counterpart, a size-cap lint plus a rule-conformance lint, so it cannot silently rot.

Advanced examples → Docs hierarchy + governance index

Governance Doc Controls · Context · Soft·Hard · Essential

Engineering Note → B.9

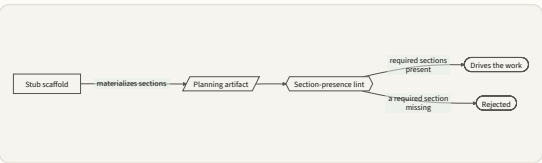


Doc-hygiene lints (index coverage, auto-gen provenance) (online)

An instance of: Drift / Parity Gate →

A family of lints that hold *documentation* to mechanical checks (index coverage, auto-generated-file provenance headers, cross-reference validity), so docs cannot silently drift, go stale, or be hand-edited where they will be overwritten.

Governance Doc Controls · Synchronization · Hard · Essential

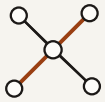


Epic & design-doc templates (online)

An instance of: Validated Dispatch →

Fixed section-templates for the two core planning artifacts, Epics and design docs. Every effort uses the same required sections: scope, phase decomposition, second-order dynamics, definition-of-done, observability. The template drives the work, not documents it after.

Governance Doc Controls · Admission · Soft·Hard · Essential



TECHNIQUE · Re-Derived Definition of Done

Epic Definition-of-Done (Final-Opus trust-nothing re-run)

An Epic-close gate mandating a "trust nothing" Final-Opus review that re-runs every owned pin test and lint *at HEAD*, rather than trusting phase markers or prior claims, so an Epic cannot close on stale or rotted assertions.

Governance Doc Controls · Validation · Hard · Essential

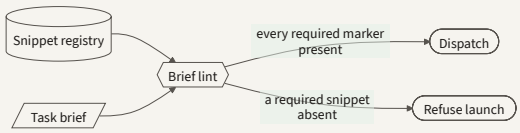
Engineering Note → B.10

STRUCTURE DIAGRAM

Independent pre-implementation design review (online)

Before any implementation phase, a **fresh reviewer that did not author the design** re-derives it from the code, verifies its load-bearing claims empirically, and **rules on the open design forks** — the reviewer wins conflicts, and implementation proceeds only on the ratified design (our instance: every founding design earns one second, independent review pass before…

Governance Doc Controls · Admission · Soft·Hard · Specialized



Mandatory snippet-table enforcement (online)

An instance of: Validated Dispatch →

A registry of mandatory agent-brief snippets (PATH export, commit-cadence, worktree discovery, submodule check, …) whose presence is asserted at dispatch by brief-linting, so every dispatched brief carries the safety and context boilerplate it needs.

Governance Doc Controls · Context · Hard · Essential

TECHNIQUE · Encoded Operational Judgment

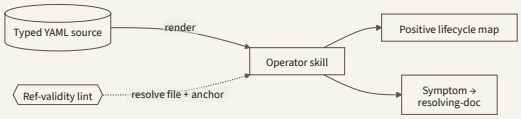
Operational playbooks

A library of documented, devops-themed decision procedures that agents and orchestrators consult instead of reasoning from scratch. Recurring operational situations — a broken deploy, a wedged cron, a stuck worktree — get a consistent, incident-tested response.

Advanced examples → Operator runbook skill (positive map first, symptom index fallback)

Governance Doc Controls · Governance · Soft · Essential

Engineering Note → B.11



Operator runbook skill (positive map first, symptom index fallback) (online)

An instance of: Encoded Operational Judgment →

A loadable skill giving an operating agent the positive map of how the substrate works and its healthy baselines first, with a symptom-to-resolving-doc catalog as fallback. Its content is generated from a structured source of truth, kept honest by a reference-validity lint.

Governance Doc Controls · Governance · Soft·Hard · Specialized

TECHNIQUE · Self-governance

Self-governance (detect your own recurring issues; convert each into a tasteful control)

Give the system permission to govern the way it is governed: detect recurring issues and introduce proportionate controls that prevent them, rather than repatching each instance. Classify the failure class, then add the smallest guardrail that kills it, on a cadence.

Governance Doc Controls · Governance · Soft·Hard · Essential

Engineering Note → B.12

Enforce at the right semantic level (online)

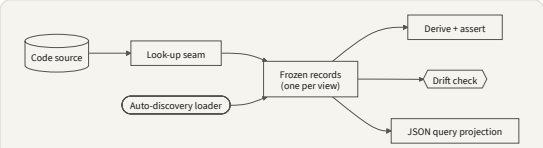
Place a check at the level where the property it verifies first becomes visible, not the cheapest point. A check fired too early can't see the property, or rejects a legitimate partial state. Example: check model-code drift when an agent finishes a task, not at each commit.

Governance Doc Controls · Governance · Soft · Essential

C.2 Models-bridge

Models-bridge — the structured models the fleet reasons through — the shared map a bounded agent uses to operate a codebase larger than its context.

35 mechanisms. Each brick links to its full Gang-of-Four entry in the online catalogue; a flagship also carries a deep-dive note in Appendix B.



The agent-first MBSE harness (online)

An instance of: Executable Source of Truth →

Builds the structured system models as a thin, hand-rolled harness over plain frozen records: adopt a modeling genre's vocabulary and schema per view, but skip its runtime. Build-time checks then keep the model equal to the code it describes.

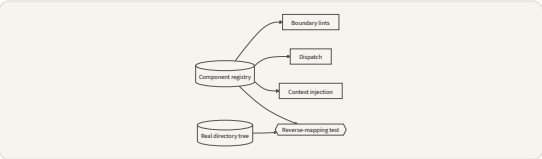
System Models · Modeling · Hard · Essential

Agent-orchestration model (developer journeys) (online)

An instance of: Executable Source of Truth →

Models the agent fleet and the orchestrator's loop with the same method applied to the product: structured lifecycle states and seams, invariants whose tier is derived, not hand-set. The substrate producing the software becomes as checkable as the software itself.

System Models · Modeling · Hard · Specialized



Component & zone model (online)

An instance of: Executable Source of Truth →

A typed catalogue of every component's code zone (focus-dirs, tags, boundary kind, external seams, read surfaces), so "which component owns this file, and what may touch it" is a queried fact, not a guess.

System Models · Modeling · Hard · Essential

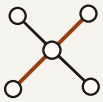
TECHNIQUE · Composed State-Machine Model

Composed state-machine model (typed lifecycles + cross-machine invariants)

Model a concurrent lifecycle as structured state machines running at once, and name cross-machine predicates as first-class invariants. Derive each invariant's check from its shape: safety gets exhaustive verification, liveness a temporal check, linear a property test.

System Models · Modeling · Hard · Specialized

Engineering Note → B.13



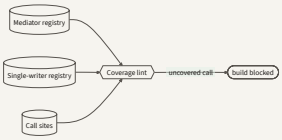
TECHNIQUE · Derived Traceability

Symbol-anchored traceability graph (derived edges)

Links every model to its lint, code entry-point, proof, and registry as a structured graph whose edges are derived obligations a lint re-checks, anchored to resolvable symbols rather than line numbers. Derived edges defend against drift; snapshotted ones decay.

System Models · Synchronization · Hard · Specialized

Engineering Note → B.22

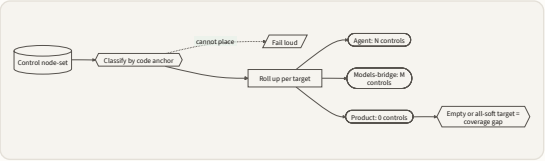


Mediator & single-writer contracts (online)

An instance of: Executable Source of Truth →

Typed registries of the system's *concurrency contracts*: which subprocess invocations are serialized by a mediator, and which state-mutation functions are single-writer / monopoly. "Who may run this, and how many at once" becomes declared and enforceable.

System Models · Modeling · Hard · Specialized

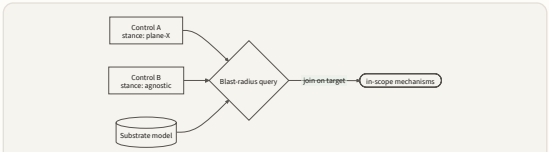


Control-coverage census (controls per governance target) (online)

An instance of: Governance Graph →

Classifies every governance control by which complementary control target it guards, derived from the control's own code anchor rather than hand-declared. Rolls the control set up per target, so a target with zero controls or only soft aims surfaces as a re-derived coverage gap.

System Models · Governance · Soft·Hard · Specialized



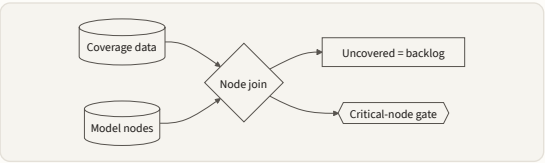
TECHNIQUE · Computed Control Blast Radius

Control↔substrate dependency (computed blast-radius)

Each control declares the substrate assumption it bakes in as structured metadata, so "what depends on this, and what breaks if I change it" becomes a computed query, not a grep-and-read. This surfaces the blast radius before any cross-cutting substrate change.

System Models · Governance · Hard · Specialized

Engineering Note → B.14



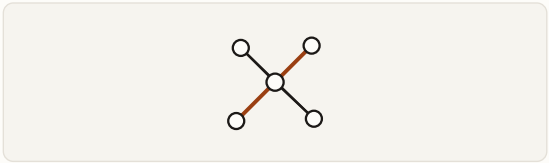
Coverage → model-node mapping (which invariants are actually tested)

An instance of: Model-Derived Assurance Coverage →

Projects test coverage onto a model's own nodes — its states, seams, and invariants — so "is this tested?" is a queried fact, not a guess from a line-coverage percentage. An invariant node with no covering test is a visible gap that becomes the next test to write.

System Models · Validation · Soft·Hard · Specialized

Engineering Note → B.15

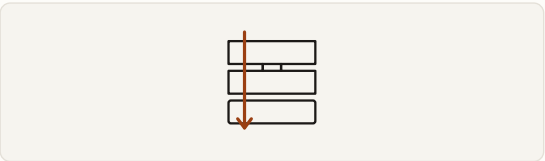


Compliance data-flow model (typed sinks and edges for privacy reasoning) (online)

An instance of: Executable Source of Truth →

Model where governed data can flow as a structured graph of sinks and edges. A question like where personal data lands, and whether it can all be erased, is answered by walking the model, not by grepping code. An unmodeled sink is a build finding, not an audit surprise.

System Models · Modeling · Hard · Specialized

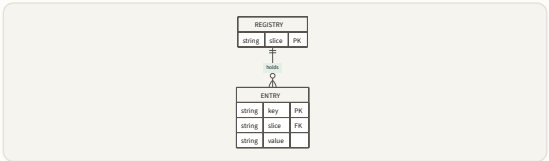


Deployment & tier topology (online)

An instance of: Executable Source of Truth →

Structured models of *where things run and how they layer* (the managed-deployment topology, each service's tier class, and the agent-substrate's layer boundaries), so deploy scripts and layering lints reason about a declared topology, not scattered constants.

System Models · Modeling · Hard · Specialized

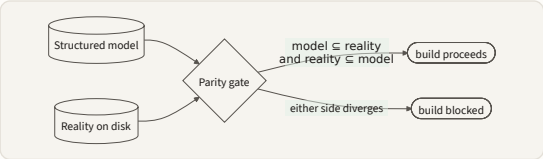


Domain registries (online)

An instance of: Executable Source of Truth →

A set of frozen, typed registries for the system's *domain facts* (the supported filetypes, the WCAG coverage gaps, the periodic-GC cron entries, the UX write-authority surfaces, the competitor set, the CLAUDE.md rule metadata), each the single source of truth for its slice.

System Models · Modeling · Hard · Essential



TECHNIQUE · Drift / Parity Gate

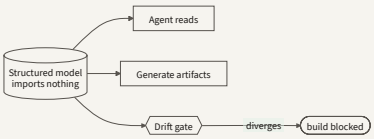
Drift & parity gates

The fleet of lints and tests that enforce **bidirectional parity between each model and reality** (every model row ↔ a real thing on disk, and every real thing ↔ a model row), so the models *cannot drift*.

Advanced examples → Doc-hygiene lints (index coverage, autogen provenance), Cross-source coherence lints, DDT pin-trailers (doc-derived characterization)

System Models · Synchronization · Hard · Essential

Engineering Note → B.16



TECHNIQUE · Executable Source of Truth

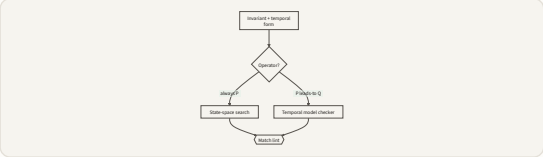
Executable source-of-truth models

Model the system as **typed data that tools read on every run and generate real artifacts from**. The model becomes *executable documentation that cannot drift*, and the codebase becomes operable by a context-bounded agent.

Advanced examples → The agent-first MBSE harness, Agent-orchestration model (developer journeys), Component & zone model, Mediator & single-writer contracts, Compliance data-flow model (typed sinks and edges for privacy reasoning), Deployment & tier topology, Domain registries, Invariant-DAG execution policy (a typed Scheduler separates correctness from resource + cost), Lifecycle model (typed operational map → generated runbook), Model-driven codegen, Process view (concurrent processes, lanes, and racing edges), Required-configuration-per-role manifest (admission policy on complete env), Rule-metadata registry (machine-readable metadata on governance rules), Service-flow / API model, Synchronization model (meta-sync), Telemetry-collection provenance (per-stream origin, landing, per-env coverage), Time-out-budget ordering model (nested wall-clock budgets, checked), Typed contract surfaces (the contract is a checked model, not a comment), User-journey model (product-goal → implementation bridge)

System Models · Modeling · Hard · Essential

Engineering Note → B.17



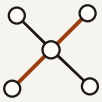
Formal invariant verification (temporal form → model checking)

An instance of: Model-Derived Assurance Coverage →

Gives every model invariant a temporal form — safety (always holds) or liveness (eventually leads to) — and uses that form to derive which exhaustive checker verifies it. An invariant is proven by the method its shape demands, a state-space check rather than a sampled test.

System Models · Validation · Hard · Specialized

Engineering Note → B.18



TECHNIQUE · Governance Graph

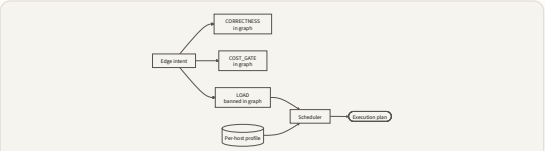
Governance graph (mechanism-interaction model)

Model the fleet's governance mechanisms as a structured graph: each mechanism a node tagged by its firing event and the resources it reads, writes, or locks. Each edge marks a conflict between two mechanisms over one resource, caught by construction, not in production.

Advanced examples → Control-coverage census (controls per governance target)

System Models · Governance · Soft·Hard · Essential

Engineering Note → B.19



Invariant-DAG execution policy (a typed Scheduler separates correctness from resource + cost) (online)

An instance of: Executable Source of Truth →

Keeps a build/deploy dependency graph a statement of correctness only, pushing concurrency and cost concerns into a Scheduler that reads a per-host profile. Edges carry an intent tag — correctness, cost-gate, or load — separating a dependency from a resource accommodation.

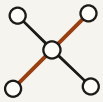
System Models · Orchestration · Soft·Hard · Specialized

Journey-criticality → test-tier placement (which host a test runs on, derived) (online)

An instance of: Model-Derived Assurance Coverage →

A journey's criticality is the single input that derives which environment tier its tests run in, never hand-drawn. A coverage floor requires every high-criticality journey-part to carry a test in the fast local tier, so a green local run means the major paths work.

System Models · Validation · Soft·Hard · Specialized

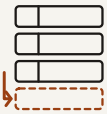


Journey task-closure (type the terminal post-condition, derive its strength) (online)

An instance of: Model-Derived Assurance Coverage →

Express a journey's terminal post-condition as a boolean formula over reusable, observable leaf-predicates. Derive a closure-strength verdict from it, and require every major journey to reach TASK_CLOSED. A journey test can no longer pass while the task is broken.

System Models · Validation · Soft·Hard · Specialized

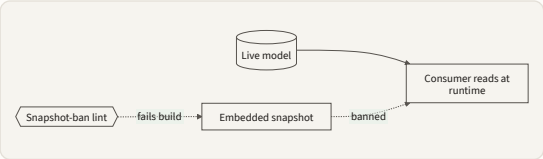


Lifecycle model (typed operational map → generated runbook) (online)

An instance of: Executable Source of Truth →

Models how the operating substrate works as a structured object: each lifecycle a named subsystem with a healthy-state predicate, each symptom a row keyed to its lifecycle. The operator's runbook is generated from this model, so it cannot drift from the system it describes.

System Models · Modeling · Hard · Specialized



TECHNIQUE · Read the Model, Don’t Copy It

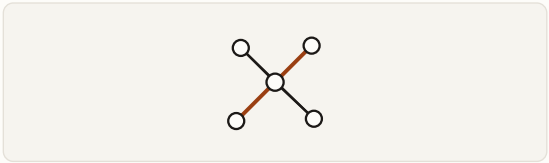
Meta-model consumption discipline (read, don’t hardcode)

Consume the models *by querying them at runtime*, never by embedding a hardcoded snapshot — so a lint, test, or brief always reasons from the live model, and a copied-out value can’t drift behind the model it was copied from.

Advanced examples → Model-graded finding severity (distance-graded gate), Model query surface (`repo-query`)

System Models · Modeling · Hard · Essential

Engineering Note → B.20



TECHNIQUE · Model-Derived Assurance Coverage

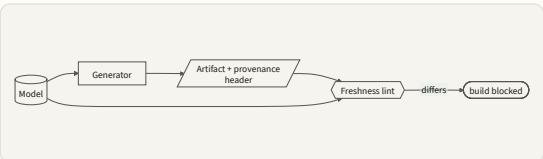
Model-derived test-obligation census (derive what should be tested, lint the gap)

Derives what should be tested — seams to fuzz, failure edges needing injection tests, invariants needing checkers — directly from the structured models. Lints the gap between that obligation set and the tests that exist, turning missing coverage into a named finding.

Advanced examples → Coverage → model-node mapping (which invariants are actually tested), Formal invariant verification (temporal form → model checking), Journey-criticality → test-tier placement (which host a test runs on, derived), Journey task-closure (type the terminal post-condition, derive its strength)

System Models · Validation · Hard · Specialized

Engineering Note → B.21

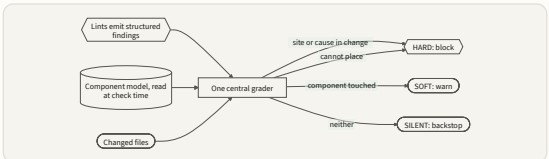


Model-driven codegen (online)

An instance of: Executable Source of Truth →

Generate real artifacts **from** the models — NetworkPolicy, service catalog, env wiring, public-API docs, competitor catalog, wire-contract types, docker — each carrying a provenance header, so the model *drives the system* (not merely describes it) and hand-edits are caught.

System Models · Modeling · Hard · Specialized



Model-graded finding severity (distance-graded gate) (online)

An instance of: Read the Model, Don’t Copy It →

Grades each lint finding’s severity — block, warn, or silence — by its distance, in a structured component model, from the files the commit actually changed. One central join computes this over every finding at check time, rather than each lint scoping itself.

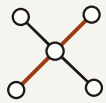
System Models · Governance · Hard · Specialized

STRUCTURE DIAGRAM

Orphan-coverage metric (walk code → governance; score the un-covered remainder) (online)

Point a tracer at the *code* and ask, for each governance-relevant site, “does any model row or any control node reach this?” Score the remainder — the **orphans**, sites nothing governs — as a rate, cluster them, and treat each cluster as a candidate for a new model or a new control. It walks the…

System Models · Governance · Soft · Specialized

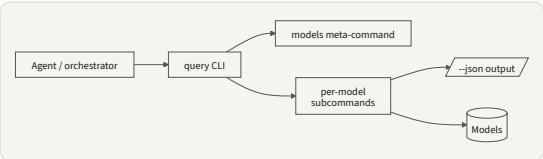


Process view (concurrent processes, lanes, and racing edges) (online)

An instance of: Executable Source of Truth →

Project a system’s concurrency as an explicit process view: the processes running at once, the lanes they run in, and the racing edges where two touch shared state simultaneously. It answers what is live at once and where it can collide, not what transitions are legal.

System Models · Modeling · Hard · Specialized

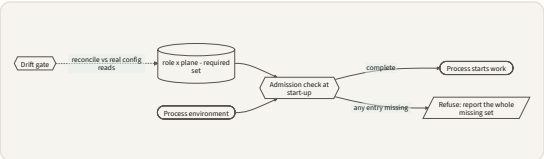


Model query surface (`repo-query`) (online)

An instance of: Read the Model, Don’t Copy It →

One canonical, self-describing query API over all the models — a `repo-query` CLI with deterministic `--json` subcommands — so an agent reads the system’s compressed truth *through a tool* rather than parsing raw files, and the tool itself documents how the models load.

System Models · Modeling · Soft · Essential

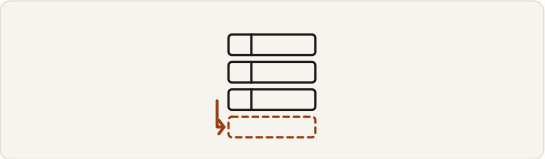


Required-configuration-per-role manifest (admission policy on complete env) (online)

An instance of: Executable Source of Truth →

Declares, per operating role and plane, the complete configuration a process must have to start, as a structured manifest an admission check reads. A process whose environment is missing any required setting is refused at launch rather than failing quietly deep in a request.

System Models · Admission · Hard · Specialized

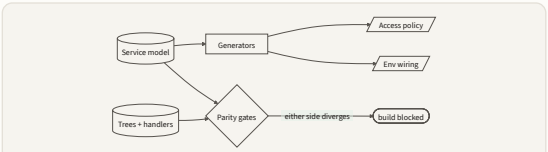


Rule-metadata registry (machine-readable metadata on governance rules) (online)

An instance of: Executable Source of Truth →

Attaches machine-readable metadata — identifier, scope, severity, enforcing check, detail location — to each governance rule, then extracts it into a queryable registry. Questions like “which rules lack an enforcing lint” become registry queries, not manual reads.

System Models · Governance · Hard · Essential

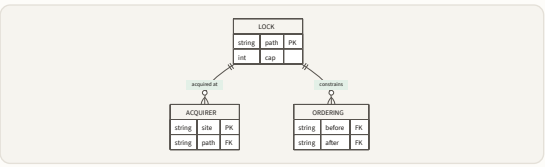


Service-flow / API model (online)

An instance of: Executable Source of Truth →

A Backstage-dialect model of the service-oriented architecture — every service, its APIs, inter-service auth, URL wiring, and frontend trees — that is the **source of truth** the deployment, NetworkPolicy, and API docs are *generated from* and validated against.

System Models · Modeling · Hard · Specialized



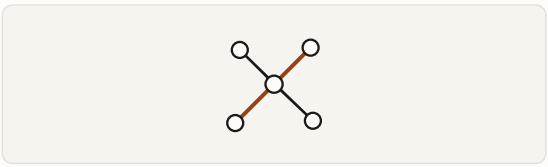
Synchronization model (meta-sync)

An instance of: Executable Source of Truth →

A typed registry that models the system’s **synchronization behaviour** — every OS-level lock (`flock`/`lockf`), which shared resource it guards, and the required acquisition *ordering* — so concurrency contracts are declared and checkable, not tribal.

System Models · Modeling · Hard · Specialized

Engineering Note → B.23

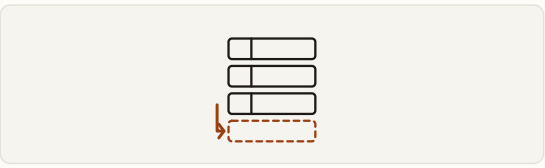


Telemetry-collection provenance (per-stream origin, landing, per-env coverage) (online)

An instance of: Executable Source of Truth →

Records each telemetry stream’s provenance — origin, landing sink, and which environments actually collect it — as a checkable fact, not an assumption. This catches the silent failure of a metric that exists in production but is quietly missing elsewhere.

System Models · Provenance · Hard · Specialized

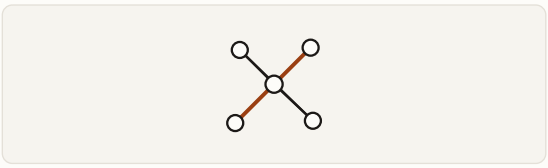


Timeout-budget ordering model (nested wall-clock budgets, checked) (online)

An instance of: Executable Source of Truth →

Gathers a system’s scattered wall-clock budgets — timeouts, deadlines, lock waits, retry windows — into one surface. States the required nesting order as a machine-checkable invariant, so a timeout raised past its container is a build finding, not a production hang.

System Models · Modeling · Hard · Specialized

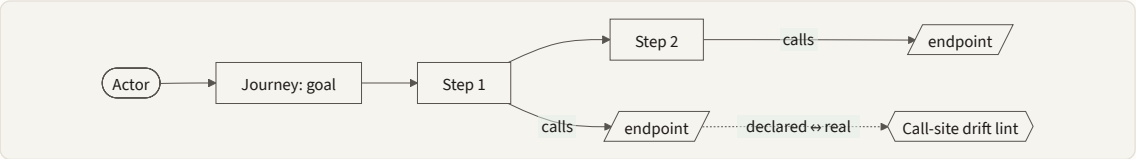


Typed contract surfaces (the contract is a checked model, not a comment) (online)

An instance of: Executable Source of Truth →

Turns the contract between two parties — an API and its clients, a CLI and the scripts parsing its output — into a checked model instead of a shared assumption. A breaking change then reddens a build gate instead of silently breaking the other side at runtime.

System Models · Modeling · Hard · Essential



User-journey model (product-goal → implementation bridge) (online)

An instance of: Executable Source of Truth →

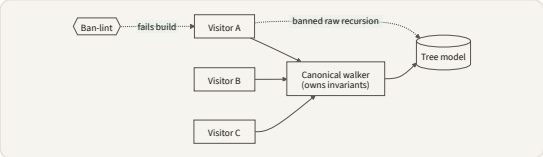
Model the product's user journeys as first-class structured entities: an actor pursuing a goal through ordered steps, each boundary-crossing step joined to the endpoint it calls. This becomes a queryable model, grafted into the existing service-architecture dialect.

System Models · Modeling · Hard · Specialized

C.3 Product

Product — the shipped artifact itself — its canonical seams, its content-fidelity validation, and its conformance controls.

20 mechanisms. Each brick links to its full Gang-of-Four entry in the online catalogue; a flagship also carries a deep-dive note in Appendix B.

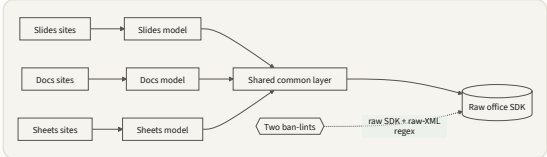


Canonical walkers (one traversal per tree) (online)

A document-accessibility instance of the technique: One Door Enforced →

Give each tree exactly one canonical walker that owns its traversal invariants, and route all traversal through it instead of ad hoc recursion, so those invariants live in one place (our instances: one walker for the PDF structure tree, one for the checking pass, one per Office part: `PdfStructTreeWalker`, the `RuleWalkers`, `DocxTopLevelPartWalker`).

Canonical Models And Seams · Constraint · Hard · Specialized

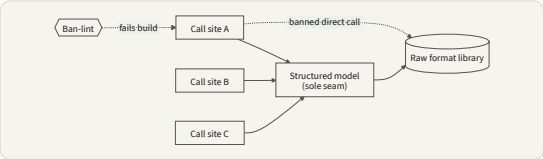


Office Models ({Slides,Docs,Sheets}Model) (online)

A document-accessibility instance of the technique: One Door Enforced →

Route all remediation of a format family through one structured model, with raw library access (and raw string-matching into the serialized form) banned by lint. The same construction+ban-lint pattern as pdf-model, on a second object model (our instance: `{Slides,Docs,Sheets}Model` over `DocumentFormat.OpenXml`).

Canonical Models And Seams · Constraint · Hard · Specialized



TECHNIQUE · One Door Enforced

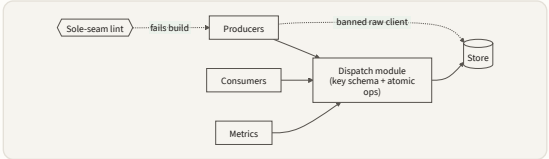
PdfModel (sole PDF mutation surface)

Route *all* reads and writes of a complex file format through one structured model, with raw access to the underlying library banned by a lint, so the structure is compiler-checked and every mutation passes through a surface that encodes the format's invariants (our instance: `PdfModel` over the canonical PDF library).

Advanced examples → Canonical walkers (one traversal per tree), Office Models ({Slides,Docs,Sheets}Model), Sole raw-Redis seam (the dispatch module), ServiceClient (typed cross-service seam)

Canonical Models And Seams · Constraint · Hard · Specialized

Engineering Note → B.24

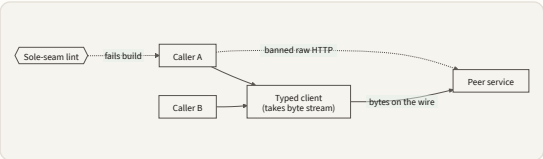


Sole raw-Redis seam (the dispatch module) (online)

An instance of: One Door Enforced →

Confine *all* raw-Redis access to one module, with every queue key declared there, so the queue's atomicity and schema invariants live in exactly one lint-enforced place.

Canonical Models And Seams · Constraint · Hard · Specialized

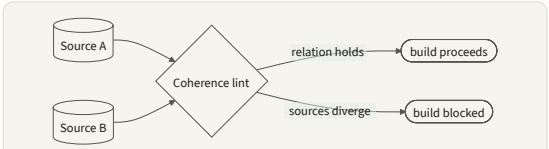


ServiceClient (typed cross-service seam) (online)

An instance of: One Door Enforced →

Route all cross-service HTTP through one typed client whose `BinaryIO` signature makes the file-path-over-wire bug class *impossible at the type level* direct `requests.post` against another service is banned.

Canonical Models And Seams · Constraint · Hard · Specialized

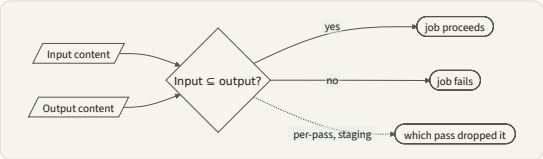


Cross-source coherence lints (online)

An instance of: Drift / Parity Gate →

Lints that assert two or more independent sources *agree* (config record ↔ sample JSON, registry ↔ its consumers, enum ↔ its uses), catching drift *between* sources that each look valid on their own.

Validation And Conformance · Synchronization · Hard · Essential



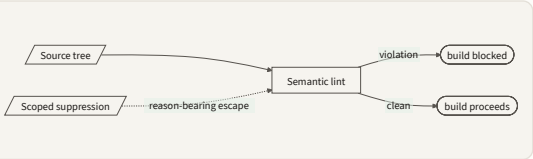
TECHNIQUE · Preservation Invariant

ContentValidator (input ⊆ output fidelity)

A deterministic gate asserting that every piece of the user's original content survives remediation (*input content* ⊆ *output content*), run in production, with a per-pass variant in staging that pinpoints which pass dropped content (our instance: the `ContentValidator` fidelity gate).

Validation And Conformance · Validation · Hard · Specialized

Engineering Note → B.25



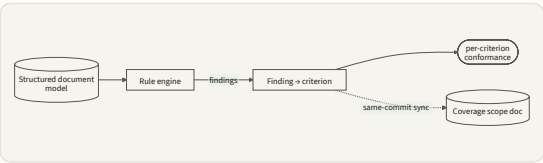
TECHNIQUE · Machine-Enforced Semantic Policy

Blocking semantic lints

A fleet of blocking semantic lints over the tool's *own source* (banned APIs, silent-catch bans, `Console.WriteLine`-in-prod, typed-seam violations) that fail the build on domain-invariant violations the compiler and review can't catch.

Validation And Conformance · Constraint · Hard · Essential

Engineering Note → B.26

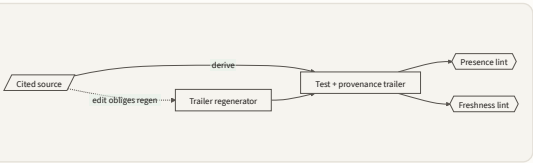


Standards / WCAG rule engine (online)

A document-accessibility instance of the technique: Conformance-to-External-Spec Engine →

A rule engine that maps each finding to the exact external-standard clause it closes, turning "does this conform?" into a deterministic, standards-grounded predicate rather than a judgement call (our instance: the **WCAG 2.1 AA / Section 508 / PDF-UA** accessibility rule engine).

Validation And Conformance · Validation · Hard · Specialized

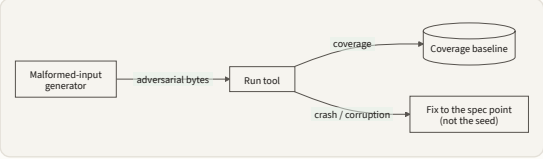


DDT pin-trailers (doc-derived characterization) (online)

An instance of: Drift / Parity Gate →

Doc-derived tests that characterize *current* behaviour before structural churn, each carrying a provenance trailer (audited / source / pins) so it is regenerated when the source it cites is edited.

Regression Tests · Synchronization · Hard · Specialized



TECHNIQUE · Generative Validation

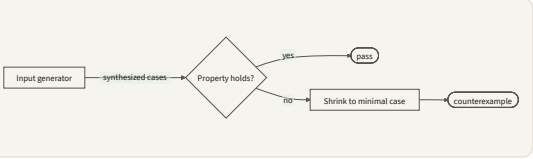
Fuzz campaigns (+ auto-coverage)

Campaigns that feed malformed and adversarial inputs to the tool to find crashes and corruption, with coverage collected automatically, plus an RCA discipline that fixes to the *spec*, not the failing seed.

Advanced examples → FsCheck property tests

Regression Tests · Validation · Hard · Specialized

Engineering Note → B.27

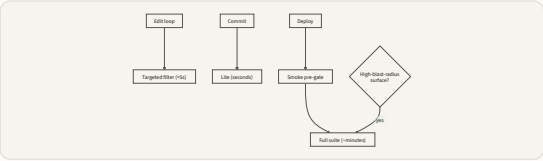


FsCheck property tests (online)

An instance of: Generative Validation →

Property-based tests that assert *invariants* (round-trip, combinatorial) over machine-*generated* inputs, catching bugs in the input space that example-based tests never reach.

Regression Tests · Validation · Hard · Essential

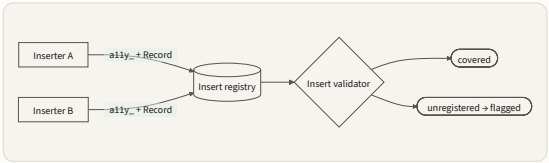


Test-onion tiers (Smoke / Lite / targeted / full) (online)

An instance of: Staged Admission Gates →

A tiered test suite that stratifies ~4000+ tests by cost and coverage (Smoke, Lite, targeted, full) so cheap tiers gate fast iterations and the full tier gates deploy, pinning behaviour at the right price per decision.

Regression Tests · Validation · Hard · Essential

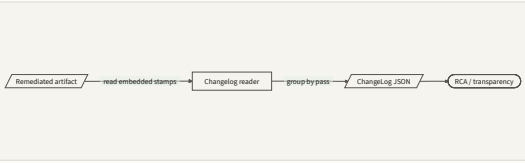


`a11y_` **prefix convention** (online)

A document-accessibility instance of the technique: Caused-By Provenance →

A reserved naming prefix that marks tool-inserted, invisible-to-author artifacts (user-visible insertions keep ordinary names), so inserted content is distinguishable from authored content and a validator can cover it by prefix (our instance: the `a11y_` prefix + `InsertedContentValidator`).

Provenance And Attribution · Provenance · Hard · Specialized

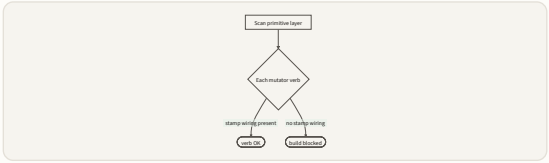


`derive-changelog` **(reconstruct mutations)** (online)

A document-accessibility instance of the technique: Caused-By Provenance →

A command that reconstructs the document's mutation history from the embedded stamp registry, turning the attribution stamps into a readable, attributed ChangeLog after the fact.

Provenance And Attribution · Provenance · Hard · Specialized



F10 mutator-stamp-wiring lint (online)

A document-accessibility instance of the technique: Caused-By Provenance →

A lint that fails the build if *any* mutator verb in the Model `Primitives` lacks stamp wiring, so a new mutator cannot land producing unattributable mutations.

Provenance And Attribution · Provenance · Hard · Specialized

TECHNIQUE · Caused-By Provenance

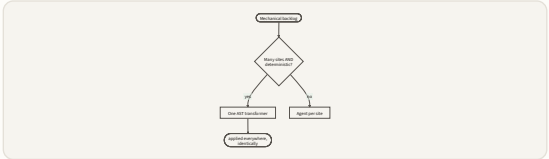
Per-mutator attribution stamps

Every remediation verb that mutates a document embeds an attribution stamp in the artifact itself, written through one sanctioned stamp-writer per format. This makes every change durably attributable and the mutation history reconstructable after the fact.

Advanced examples → Caused-by provenance (agent-side change traceability), `a11y_` prefix convention, `derive-changelog` (reconstruct mutations), F10 mutator-stamp-wiring lint

Provenance And Attribution · Provenance · Hard · Specialized

Engineering Note → B.28

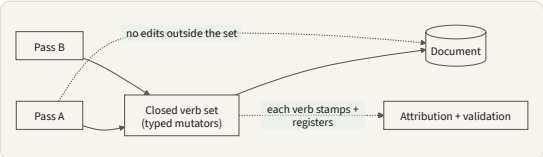


Codemod-first threshold (N≳50 → AST transformer) (online)

An instance of: Closed Action Vocabulary →

For a large mechanical backlog (N≳50 sites with a deterministic fix shape), author *one AST-level transformer* instead of dispatching N agents, bounding *how* large mechanical changes are made.

Repair Vocabulary · Constraint · Soft · Essential



TECHNIQUE · Closed Action Vocabulary

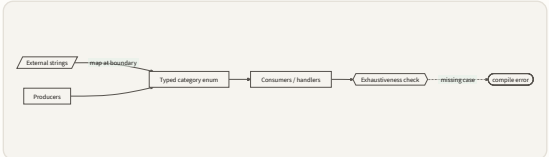
Closed remediation-verb sets

Route the remediator's mutations through a *bounded, named set* of typed verbs (a closed typed-mutator layer) rather than free-form edits, so the move-space is enumerable and every move can be stamped, validated, and policy-checked (our instance: the document models' `Primitives`).

Advanced examples → Role-typed dispatch, Codemod-first threshold (N≳50 → AST transformer), Typed `ViolationCategory` / `FailureCategory` enums

Repair Vocabulary · Constraint · Hard · Specialized

Engineering Note → B.29



Typed `ViolationCategory` / `FailureCategory` enums (online)

An instance of: Closed Action Vocabulary →

Replace free-form failure strings with typed enums, so the categorical move-space is a closed, enumerable set that the compiler and lints can check for exhaustive handling.

Repair Vocabulary · Constraint · Hard · Essential

Reference: every mechanism

The 29 flagship patterns are printed in full below; all 85 mechanisms — including these — are online in the web catalogue at https://davisjam.github.io/model-based-agentic-software-engineering. A flagship row links to its page in this appendix; a web-only mechanism links to its live catalogue entry, marked *(online)*.

Agent

Context & dispatch substrate

- Appendix B - 1. Brief-linting

- Docs hierarchy + governance index (online) `package` · under Governed Knowledge Base
- Appendix B - 2. Dynamic context injection
- Role-typed dispatch (online) · under Closed Action Vocabulary

Gates & merge-train

- Merge-train MIS batching (online) · under Staged Admission Gates
- Pre-commit hook (3-stanza, tree-sha markers) (online) · under Staged Admission Gates
- Sentinel first-commit early-abort (online) · under Staged Admission Gates
- Appendix B - 3. Staged deploy gates (canary → smoke → promote)

Mediators & resource locks

- Aggregate-compute protection (`lint-all` host mutex) (online) · under Mediated Resource Admission
- Build-serializer (M=8 semaphore) (online) · under Mediated Resource Admission
- Appendix B - 4. Resource-pressure gating (admit before, shed during)
- Appendix B - 5. Test-serializer (N=1 flock on `dotnet test`)

Lifecycle & observability

- Appendix B - 6. Agent registry (append-only log + marker cache)
- Caused-by provenance (agent-side change traceability) (online) `package` · under Caused-By Provenance
- Cron-alerts gate (online) · under Staged Admission Gates
- Deploy heartbeats + stale-worker detection (online) · under Fleet Observability Surface
- Appendix B - 7. Lifecycle hooks (interpose on the agent runtime's events) `package`
- Reflection-facet substrate (tempo-gated policy nudges) (online) `package` · under Point-of-Action Policy Delivery
- Tombstone commits (lifecycle close records) (online) · under Authoritative Lifecycle State
- Appendix B - 8. Orchestrator-as-reactor over an event bus

Governance-doc controls

- Appendix B - 9. CLAUDE.md rule index (the governance document as a mechanism) `package`
- Doc-hygiene lints (index coverage, autogen provenance) (online) · under Drift / Parity Gate
- Epic & design-doc templates (online) `package` · under Validated Dispatch
- Appendix B - 10. Epic Definition-of-Done (Final-Opus trust-nothing re-run) `package`
- Independent pre-implementation design review (online) `package`
- Mandatory snippet-table enforcement (online) · under Validated Dispatch
- Appendix B - 11. Operational playbooks
- Operator runbook skill (positive map first, symptom index fallback) (online) `package` · under Encoded Operational Judgment
- Appendix B - 12. Self-governance (detect your own recurring issues; convert each into a tasteful control) `package`
- Enforce at the right semantic level (online) · under P8 (Enforce at the right semantic level)

Models-bridge

System models

- The agent-first MBSE harness (online) `package` · under Executable Source of Truth

- Agent-orchestration model (developer journeys) (online) `package` · under Executable Source of Truth
- Component & zone model (online) `package` · under Executable Source of Truth
- Appendix B - 13. Composed state-machine model (typed lifecycles + cross-machine invariants) `package`
- Mediator & single-writer contracts (online) `package` · under Executable Source of Truth
- Control-coverage census (controls per governance target) (online) · under Governance Graph
- Appendix B - 14. Control↔substrate dependency (computed blast-radius) `package`
- Appendix B - 15. Coverage → model-node mapping (which invariants are actually tested)
- Compliance data-flow model (typed sinks and edges for privacy reasoning) (online) `package` · under Executable Source of Truth
- Deployment & tier topology (online) `package` · under Executable Source of Truth
- Domain registries (online) `package` · under Executable Source of Truth
- Appendix B - 16. Drift & parity gates
- Appendix B - 17. Executable source-of-truth models `package`
- Appendix B - 18. Formal invariant verification (temporal form → model checking)
- Appendix B - 19. Governance graph (mechanism-interaction model) `package`
- Invariant-DAG execution policy (a typed Scheduler separates correctness from resource + cost) (online) `package` · under Executable Source of Truth
- Journey-criticality → test-tier placement (which host a test runs on, derived) (online) `package` · under Model-Derived Assurance Coverage
- Journey task-closure (type the terminal post-condition, derive its strength) (online) `package` · under Model-Derived Assurance Coverage
- Lifecycle model (typed operational map → generated runbook) (online) `package` · under Executable Source of Truth
- Appendix B - 20. Meta-model consumption discipline (read, don't hardcode)
- Appendix B - 21. Model-derived test-obligation census (derive what should be tested, lint the gap)
- Model-driven codegen (online) · under Executable Source of Truth
- Model-graded finding severity (distance-graded gate) (online) · under Read the Model, Don't Copy It
- Orphan-coverage metric (walk code → governance; score the un-covered remainder) (online)
- Process view (concurrent processes, lanes, and racing edges) (online) `package` · under Executable Source of Truth
- Model query surface (`repo-query`) (online) · under Read the Model, Don't Copy It
- Required-configuration-per-role manifest (admission policy on complete env) (online) `package` · under Executable Source of Truth
- Rule-metadata registry (machine-readable metadata on governance rules) (online) `package` · under Executable Source of Truth
- Service-flow / API model (online) `package` · under Executable Source of Truth
- Appendix B - 22. Symbol-anchored traceability graph (derived edges) `package`
- Appendix B - 23. Synchronization model (meta-sync) `package`
- Telemetry-collection provenance (per-stream origin, landing, per-env coverage) (online) `package` · under Executable Source of Truth
- Timeout-budget ordering model (nested wall-clock budgets, checked) (online) `package` · under Executable Source of Truth
- Typed contract surfaces (the contract is a checked model, not a comment) (online) `package` · under Executable Source of Truth

- User-journey model (product-goal → implementation bridge) (online) `package` · under Executable Source of Truth

Product

Canonical models & seams

- Canonical walkers (one traversal per tree) (online) `package` · under One Door Enforced
- Office Models ({Slides,Docs,Sheets}Model) (online) `package` · under One Door Enforced
- Appendix B - 24. PdfModel (sole PDF mutation surface) `package`
- Sole raw-Redis seam (the dispatch module) (online) · under One Door Enforced
- ServiceClient (typed cross-service seam) (online) · under One Door Enforced

Validation & conformance

- Cross-source coherence lints (online) · under Drift / Parity Gate
- Appendix B - 25. ContentValidator (input ⊆ output fidelity)
- Appendix B - 26. Blocking semantic lints
- Standards / WCAG rule engine (online) · under Conformance-to-External-Spec Engine

Regression tests

- DDT pin-trailers (doc-derived characterization) (online) · under Drift / Parity Gate
- Appendix B - 27. Fuzz campaigns (+ auto-coverage)
- FsCheck property tests (online) · under Generative Validation
- Test-onion tiers (Smoke / Lite / targeted / full) (online) · under Staged Admission Gates

Provenance & attribution

- `a11y_` prefix convention (online) · under Caused-By Provenance
- `derive-changelog` (reconstruct mutations) (online) · under Caused-By Provenance
- F10 mutator-stamp-wiring lint (online) · under Caused-By Provenance
- Appendix B - 28. Per-mutator attribution stamps

Repair vocabulary

- Codemod-first threshold (N≳50 → AST transformer) (online) · under Closed Action Vocabulary
- Appendix B - 29. Closed remediation-verb sets
- Typed `ViolationCategory` / `FailureCategory` enums (online) · under Closed Action Vocabulary

Appendix D

Operator's Reference

Appendix D — Operator's Reference

Operational reference for running a Governed Engineering Environment

This appendix collects operational reference material used while running a Governed Engineering Environment. The chapters teach the method; these pages support *practicing* it. You do not read them once and move on — you come back to them mid-build, the way you keep a wiring diagram at the bench.

Each entry answers a standing operational question with a single surface you can scan. The first is the **Operator's Dashboard**: the metrics you steer by while the work is in flight and certify the result with at maturity, on one page. The second is **From Drifted Wiki to Trusted Model**: the brownfield migration drill that joins a legacy wiki to the code and walks it from drifted documentation to a governed model, projected from Chapter 4.1 onto a single tear-out card.

The appendix is built to grow. These are the first reference cards, not the last; future editions add the operational surfaces a practitioner reaches for often enough to want them collected:

- **MAGE lifecycle summary** — the stages a governed build passes through, and what each one owes the next.
- **Governance-conversion quick reference** — turning a recurring failure into a durable control, at a glance.
- **Operational decision heuristics** — the calls you make while operating a fleet, with the signal that should drive each one.

Together these make the laminated reference card for running the environment: the pages you consult while operating, kept apart from the narrative that explains why they read the way they do.

Appendix D - 1. The Operator's Dashboard

The dashboard collects the primary metrics used to operate a Governed Engineering Environment. Formative metrics steer work while it is in progress. Summative metrics certify the finished system. Some metrics serve both purposes.

Table 10.1-1 gives each metric in two bands — the formative metrics above the divider, the summative verdicts below — with what each one counts, when to read it, and its healthy direction. Scan the band you need. The 2.5 reference is the companion one level down: read it to steer a single loop iteration, read this to steer or grade the whole program.

Table 10.1-1: The Operator's Dashboard — every metric the build acts on, in two mode bands: the formative metrics you steer by while the work is in flight, then the summative verdicts you certify the result with at maturity.

Metric	Mode	What it counts	When to watch	Healthy direction	Defined in
Formative — measured during the work, to steer the next step					
Missing-Model Metric	formative	Fraction of tests whose exercised code traces to no model claim — the unmodelled surface — plus its drain curve.	After each model-loop Epic; steer the next at the biggest orphan cluster.	Drains toward 10%-or-under (56% to 7.89% over nine re-runs).	2.5
Velocity	formative	Commits per week.	Watch the dip where velocity buys hardening.	Roughly linear; a hardening dip is expected, not alarming.	5.2
Churn	formative	Lines added and deleted per week per path.	Reads which build phase you are in.	Peaks at mechanization, then collapses as the environment stabilizes.	5.2
Model-sync efficacy	formative	Whether the drift and parity gates keep model equal to code.	Watch that map-equals-territory holds.	Gates stay green.	3.8
Grammar coverage	formative	Whether the generator exercised every production of the input grammar.	Watch for corpus holes no line-coverage number reveals.	Rises toward full grammar exercise.	4.6
Model-claim coverage	formative	Whether generated inputs drove every declared invariant, transition, and edge.	The saturation oracle for generative validation.	Rises toward full claim exercise.	4.6
Summative — measured at maturity, a verdict on the result					
MBSE navigation token-savings	summative	Tokens spent to reach an answer, model on versus off.	Certify the model earned its context budget.	Lower with the model on.	3.1
Support ratio	both	Support-apparatus LoC (tests, lints, docs, infra, tooling) over production LoC.	Watch the apparatus keeps leading as feature work resumes.	Leads production; settles around 3x it at maturity.	5.2
Control growth	both	Project-specific lint files and gate scripts, per window.	Watch the environment still accrete controls.	Climbs steadily (lints 0 to 747; gates 0 to 102).	2.3
Epic-closure rate	both	Epics moved into the closed set per week — the finishing rate, not the commit rate.	Whether an operating-mode shift converts to durable throughput rather than raw output.	Rises when the environment absorbs autonomous loops; flat velocity beside a rising closure rate is the healthy anti-decay shape.	5.2

Each metric's mode and the rationale for its formative-or-summative call live in the dashboard's model file, which projects this table and holds the page equal to it.

Appendix D - 2. From Drifted Wiki to Trusted Model

Most engineering organizations already own the beginnings of a governed engineering environment — an environment where engineering knowledge and policy live in shared models, tools, and checks instead of in a few people's memory. They just call it the wiki.

A wiki is a lightweight human-facing knowledge graph: its pages are nodes, its links and backlinks are edges, its tags carry loose metadata. The graph is already there; what it lacks is a disciplined join to the code. This card is the operating drill that supplies the join, then walks the wiki from drifted documentation to a trusted — and eventually executable — model. Chapter 4.1 is the full treatment; this is the one-page projection you keep at the bench.

The minimum package

Four small additions turn the wiki from optional documentation into the start of an engineering environment. Add all four, or the wiki stays a surface nobody trusts.

Table 10.2-1: The minimum package — the four additions that make a wiki into engineering infrastructure, each with the behavior it must exhibit.

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 two-way join is *traceability* — not a heap of hyperlinks, but claims and code that each point to the other.
Agent briefs	Tell agents how to reach the wiki, walk its links and tags, cite it, and maintain it as they work.
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, read the affected pages against the current repository — do not trust them because an agent reports it updated them.

One rule governs all four: **never add metadata without adding consumers.** A tag, a backlink, an ownership field, a machine-readable mirror — each earns its place only when an audit, a brief, a retrieval step, an analysis, or a gate reads it. Metadata with no reader is one more surface free to drift.

The migration path

The migration runs in four stages. Each reaches a useful stopping point, so the work pays off long before executable models arrive. The exit criterion is what tells you a stage is done — not a calendar date.

Table 10.2-2: The migration path — Audit, Synchronize, Govern, Extend, each with the work it asks and the exit criterion that closes it.

Stage	Work	Exit criterion
Audit	Inventory the major wiki regions and implementation regions; add round-trip links in both directions; create or	Every major wiki region links to the code and tests that realize it, and every major code and test region is

Stage	Work	Exit criterion
	propose pages for large code with no explaining home; classify the orphans. Stays advisory — an audit, not a gate.	linked, marked below the wiki's grain, or recorded as owing a new entry.
Synchronize	Drain the drift with ordinary work: an agent that touches linked code reads the attached pages, checks their claims, fixes nearby errors, adds missing links, and flags disagreements for a human. A boy-scout rule, not a rewrite.	A fresh audit shows the important wiki claims and the repository substantially in agreement, with no major orphan left unclassified.
Govern	Promote the wiki from optional context to required input: briefs mandate task-specific wiki analysis before implementation, templates name affected pages, the Definition of Done requires claims checked against the repository at HEAD.	The wiki is trusted enough to be required input and verified output.
Extend	Where the value justifies the cost, selected pages add structured invariants, correctness criteria, dependencies, ownership, transitions, and links to validators — mirrored in a machine-readable form once a real consumer exists.	Every structured field added drives at least one retrieval path, analysis, generator, validator, or gate.

The promotion rule

Move the wiki up the same way you land a new lint into a legacy tree: audit first, drain the findings, then make it mandatory. Enforcing a broadly inaccurate wiki only promotes old errors to official instructions, so trust is earned before it is required.

Link the surface → drain the drift → earn trust → make synchronization mandatory → extend selected claims into executable models.

The Govern step is that promotion made durable: an obligation that used to depend on someone remembering to update the wiki becomes part of the environment's acceptance criteria. (The book calls this the Alignment Thesis applied one increment at a time.)

Orphan triage

An audit sorts every unlinked region into exactly one of three states. The discrimination is what keeps the audit honest — the goal is a map whose claimed surface agrees with the territory, not one wiki entry per function.

1. **Missing model or page** — a real subsystem with no explaining page; write or propose one.
2. **Missing anchor** — the page exists, but the link back from the code is absent; add the join.

3. **Below the grain** — code too fine or too generic to earn a page of its own (typed-id aliases, config loaders, enums); the *grain* is the level of detail the model keeps, and this code sits legitimately beneath it.

Full treatment

See Chapter 4.1 for the explanation this card compresses:

- top-down modeling from a whiteboard;
- bottom-up induction of the model from code;
- lint-and-cover preparation before induction;
- the Missing Model Metric — coverage run backwards, to point at unmodelled code;
- how much governance is enough, sized by a failure's cost times its frequency.

Appendix E

How to Write a Skill

Appendix E — How to Write a Skill

A skill is a model of a domain an agent triggers into

A skill is not a prompt, and it is not a checklist. It is a *model of a domain* — the frame an agent loads when a task in that domain arrives, so it reasons through the domain's own abstractions instead of from scratch. You met three of them earlier in this book: the skills that let the fleet write its prose, harden its own substrate, and operate itself. Here is how they were made.

All three were built the same way. A skill is not a bag of instructions that grew by accretion. Each of the book's skills started from one abstraction, layered independent facets on it, and tied them together with a governing principle. That construction is itself a reusable pattern: name it and you can write the next skill deliberately instead of by feel.

This appendix names that pattern as a three-step recipe and grounds each step in the three self-* skills you already met. Read the Skills chapter for what those skills *do* read on here for how they were *built*.

The full recipe — its three steps grounded in the three self-* skills — lives in the web edition of this book: The recipe — three steps. Open it there to read each step worked through in full.

Bibliography

"Do Llms Generate Test Oracles That Capture the Actual or the Expected Program Behaviour?." 2024. https://arxiv.org/abs/2410.21136.

"SWE-Rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents." 2025. https://arxiv.org/abs/2505.20411.

Abdurrahman. "We Adopted Spec-Driven Development, Then the Specs Started Eating Our Context Window." Medium, July 2026. https://abdurrahman5.medium.com/we-adopted-spec-driven-development-then-the-specs-started-eating-our-context-window-6cb9476dfedc.

Ait, Adem, Gwendal Jouneaux, Javier Luis Cánovas Izquierdo, and Jordi Cabot. "Towards Automated Governance: A DSL for Human-Agent Collaboration in Software Projects." In "Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE), New Ideas and Emerging Results Track." Special issue, *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE), New Ideas and Emerging Results Track*, 2025. https://arxiv.org/abs/2510.14465.

Alur, Rajeev, and David L. Dill. "A Theory of Timed Automata." *Theoretical Computer Science* 126, no. 2 (1994): 183–235.

Anthropic. "How Anthropic Runs Large-Scale Code Migrations with Claude Code." Anthropic, July 16, 2026. https://claude.com/blog/ai-code-migration.

Argote, Linda, and Paul Ingram. "Knowledge Transfer: A Basis for Competitive Advantage in Firms." *Organizational Behavior and Human Decision Processes* 82, no. 1 (2000): 150–69.

Baier, Christel, and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 3rd ed. Addison-Wesley, 2012.

Beach, Derek, and Rasmus Brun Pedersen. *Process-Tracing Methods: Foundations and Guidelines*. 2nd ed. University of Michigan Press, 2019.

Beck, Kent, Mike Beedle, Arie van Bennekum, et al. "Manifesto for Agile Software Development." 2001. https://agilemanifesto.org/.

Becker, Joel, Nate Rush, Elizabeth Barnes, and David Rein. "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity." 2025. https://arxiv.org/abs/2507.09089.

B. Beyer, C. Jones, J. Petoff, N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

Bollinger, Terry B., and Clement L. McGowan. "A Critical Look at Software Capability Evaluations." *IEEE Software* 8, no. 4 (1991): 25–41. https://doi.org/10.1109/52.300034.

Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. 2nd ed. Addison-Wesley, 2005.

Brambilla, Marco, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. 2nd ed. Morgan & Claypool, 2017.

Brooks, Frederick P. "No Silver Bullet: Essence and Accidents of Software Engineering." *Computer* 20, no. 4 (1987): 10–19.

Brooks, Frederick P. *The Mythical Man-Month: Essays on Software Engineering*. Anniversary. Addison-Wesley, 1995.

Carlini, Nicholas. “Building a C Compiler with a Team of Parallel Claudes.” Anthropic, February 5, 2026. https://www.anthropic.com/engineering/building-c-compiler.

Cui, Zheyuan (Kevin), Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. “The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers.” *Management Science*, ahead of print, 2025. https://doi.org/10.1287/mnsc.2025.00535.

Czarnecki, Krzysztof, and Simon Helsen. “Feature-Based Survey of Model Transformation Approaches.” *IBM Systems Journal* 45, no. 3 (2006): 621–45. https://doi.org/10.1147/sj.453.0621.

Davis, James C., Paschal C. Amusuo, Tanmay Singla, Berk Çakar, and Kirsten A. Davis. “Cheap Code, Costly Judgment: A Case Study on Governable Agentic Software Engineering.” 2026. https://arxiv.org/abs/2607.01087.

Dhanorkar, Shipi, Samir Passi, and Mihaela Vorvoreanu. “Human Oversight of Agentic Systems in Practice: Examining the Oversight Work, Challenges, And Heuristics of Developers Using Software Agents.” 2026. https://arxiv.org/abs/2606.05391.

DocAble Research Group. “Deterministic Workflows with Bounded Model Delegation for Software Verification.” 2026. https://arxiv.org/abs/2605.10712.

Faulkner, Steve. “How We Rebuilt Next.js with AI in One Week.” Cloudflare, February 24, 2026. https://blog.cloudflare.com/vinext/.

Foster, J. Nathan, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. “Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem.” *ACM Transactions on Programming Languages and Systems* 29, no. 3 (2007): 17:1–17:65.

Friedenthal, Sanford, Alan Moore, and Rick Steiner. *A Practical Guide to Sysml: The Systems Modeling Language*. 3rd ed. Morgan Kaufmann, 2014.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

Gill, Gurbinder. “The Rise of the Agent OS: Comparing Harnesses, Runtimes, And Orchestration Layers for LLM Agents.” LinkedIn, July 15, 2026. https://www.linkedin.com/pulse/rise-agent-os-comparing-harnesses-runtimes-layers-llm-gurbinder-gill-72c9c/.

E. Hollnagel, D. D. Woods, N. Leveson. *Resilience Engineering: Concepts and Precepts*. Ashgate, 2006.

Hora, Andre, and Romain Robbes. “Are Coding Agents Generating over-Mocked Tests? An Empirical Study.” 2026. https://arxiv.org/abs/2602.00409.

Hou, Xinyi, Yanjie Zhao, Yue Liu, et al. “Large Language Models for Software Engineering: A Systematic Literature Review.” *ACM Transactions on Software Engineering and Methodology* 33, no. 8 (2024). https://doi.org/10.1145/3695988.

Hutchins, Edwin. *Cognition in the Wild*. MIT Press, 1995.

Jackson, Victoria, Susannah Liu, and André van der Hoek. “Using Generative AI in Software Design Education: An Experience Report.” 2025. https://arxiv.org/abs/2506.21703.

Kruchten, Philippe. “The 4+1 View Model of Architecture.” *IEEE Software* 12, no. 6 (1995): 42–50.

Lamport, Leslie. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

Lee, Han. “Hidden Technical Debt of AI Systems: Agent Harness.” May 8, 2026. https://leehanchung.github.io/blogs/2026/05/08/hidden-technical-debt-agent-harness/.

Lee, Yoonho, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. “Meta-Harness: End-to-End Optimization of Model Harnesses.” 2026. https://arxiv.org/abs/2603.28052.

Leveson, Nancy G. *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press, 2011.

Li, Kenneth, Aspen K. Hopkins, David Bau, Fernanda Viégas, Hanspeter Pfister, and Martin Wattenberg. “Emergent World Representations: Exploring a Sequence Model Trained on a Synthetic Task.” 2022. https://arxiv.org/abs/2210.13382.

Liang, Shanchao, Spandan Garg, and Roshanak Zilouchian Moghaddam. “The SWE-Bench Illusion: When State-of-the-Art Llms Remember Instead of Reason.” In “Proceedings of the IEEE/ACM 48th International Conference on Software Engineering, Software Engineering in Practice (ICSE-Seip).” Special issue, *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering, Software Engineering in Practice (ICSE-SEIP)*, 2026. https://doi.org/10.1145/3786583.3786882.

Lin, Jiahang, Shichun Liu, Chengjun Pan, et al. “Agentic Harness Engineering: Observability-Driven Automatic Evolution of Coding-Agent Harnesses.” 2026. https://arxiv.org/abs/2604.25850.

Liu, Xiangyan, Bo Lan, Zhiyuan Hu, et al. “Codexgraph: Bridging Large Language Models and Code Repositories via Code Graph Databases.” 2024. https://arxiv.org/abs/2408.03910.

March, James G. “Exploration and Exploitation in Organizational Learning.” *Organization Science* 2, no. 1 (1991): 71–87.

Meadows, Donella H. *Thinking in Systems: A Primer*. Chelsea Green Publishing, 2008.

Meyer, Bertrand. “From Probable to Provable.” *Communications of the ACM*, ahead of print, 2025. https://doi.org/10.1145/3773295.

Murphy, Gail C., David Notkin, and Kevin Sullivan. “Software Reflexion Models: Bridging the Gap between Source and High-Level Models.” In “Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-3).” Special issue, *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-3)*, 1995, 18–28.

OpenAI. “Why SWE-Bench Verified No Longer Measures Frontier Coding Capabilities.” OpenAI, February 2026. https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/.

Ousterhout, John. *A Philosophy of Software Design*. Yaknyam Press, 2018.

Ouyang, Siru, Wenhao Yu, Kaixin Ma, et al. “Repograph: Enhancing AI Software Engineering with Repository-Level Code Graph.” 2024. https://arxiv.org/abs/2410.14684.

Peng, Sida, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. “The Impact of AI on Developer Productivity: Evidence from Github Copilot.” 2023. https://arxiv.org/abs/2302.06590.

Reganti, Aishwarya Naresh. “The AI Agent Stack in 2026.” The Nuanced Perspective, April 29, 2026. https://thenuancedperspective.substack.com/p/the-ai-agent-stack-in-2026.

Ringer, Talia, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. “QED at Large: A Survey of Engineering of Formally Verified Software.” *Foundations and Trends in Programming Languages* 5, nos. 2–3 (2019): 102–281. https://arxiv.org/abs/2003.06458.

Runeson, Per, and Martin Höst. “Guidelines for Conducting and Reporting Case Study Research in Software Engineering.” *Empirical Software Engineering* 14, no. 2 (2009): 131–64. https://doi.org/10.1007/s10664-008-9102-8.

Scale AI. “SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?.” 2025. https://arxiv.org/abs/2509.16941.

Shah, Pratik, Rajat Ghosh, Aryan Singhal, and Debojyoti Dutta. “RANGER: Repository-Level Agent for Graph-Enhanced Retrieval.” 2025. https://arxiv.org/abs/2509.25257.

Shingo, Shigeo. *Zero Quality Control: Source Inspection and the Poka-Yoke System*. Translated by Andrew P. Dillon. Productivity Press, 1986.

Smith, Edward K., Earl T. Barr, Claire Le Goues, and Yuriy Brun. “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).” Special issue, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015, 532–43. https://doi.org/10.1145/2786805.2786825.

Stevens, Perdita. “Maintaining Consistency in Networks of Models: Bidirectional Transformations in the Large.” *Software and Systems Modeling*, ahead of print, 2020. https://doi.org/10.1007/s10270-019-00736-x.

Tan, Garry. “Thin Harness, Fat Skills.” April 9, 2026. https://github.com/garrytan/gbrain/blob/master/docs/ethos/THIN_HARNESS_FAT_SKILLS.md.

Tu, Haoxin, Huan Zhao, Yahui Song, Mehtab Zafar, Ruijie Meng, and Abhik Roychoudhury. “Agentic Verification of Software Systems.” 2025. https://arxiv.org/abs/2511.17330.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, et al. “Attention Is All You Need.” In “Advances in Neural Information Processing Systems 30 (NIPS 2017).” Special issue, *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, 2017, 5998–6008.

Vella, Annie, and Kelly Blincoe. “The Impact of AI Coding Assistants on Software Engineering: A Longitudinal Study.” 2026. https://arxiv.org/abs/2605.23135.

Vogel, Martin, Falk Meyer-Eschenbach, Severin Kohler, Elias Grünewald, and Felix Balzer. “Codebase-Memory: Tree-Sitter-Based Knowledge Graphs for LLM Code Exploration via Mcp.” 2026. https://arxiv.org/abs/2603.27277.

Wikipedia. “Computer-Aided Software Engineering.” 2026. https://en.wikipedia.org/wiki/Computer-aided_software_engineering.

Wikipedia. “Hybrid System.” 2026. https://en.wikipedia.org/wiki/Hybrid_system.

Wikipedia. “Model-Driven Architecture.” 2026. https://en.wikipedia.org/wiki/Model-driven_architecture.

Wikipedia. “Systems Modeling Language.” 2026. https://en.wikipedia.org/wiki/Systems_modeling_language.

Wikipedia. “Timed Automaton.” 2026. https://en.wikipedia.org/wiki/Timed_automaton.

Winters, Titus, Tom Manshreck, and Hyrum Wright. *Software Engineering at Google: Lessons Learned from Programming over Time*. O'Reilly Media, 2020.

Young, Justin. “Effective Harnesses for Long-Running Agents.” Anthropic, November 26, 2025. https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents.

Zaharia, Matei, Kasey Uhlenhuth, and Corey Zumar. “Introducing Omnigent: A Meta-Harness to Combine, Control and Share Your Agents.” Databricks, June 13, 2026. https://www.databricks.com/blog/introducing-omnigent-meta-harness-combine-control-and-share-your-agents.

Zhang, Linghao, Shilin He, Chaoyun Zhang, et al. “SWE-Bench Goes Live!.” 2025. https://arxiv.org/abs/2505.23419.

Zhong, Hailin, and Shengxin Zhu. “AI Harness Engineering: A Runtime Substrate for Foundation-Model Software Agents.” 2026. https://arxiv.org/abs/2605.13357.