WHITE PAPER

Software Manufacturing

A control architecture for building and maintaining software when nobody reads the code.

Travis Frisinger

September 2026

aibuddy.software

Contents

Summary

Software construction has become cheap. Establishing that a change is acceptable, and keeping that judgment true as the system, the standards and the models all change underneath it, has not. The constraint moved from generation to verification and governance, and that is where the engineering now lives.

Governance has to cover four failure modes that survive every passing check: judgment, comprehension, meaning, and erosion. Each is a way for a system to pass every check and still be wrong, and they do not close each other.

This paper states the architecture that answers them. Three layers of control, each supplying a guarantee the others cannot: a harness governs a single run, a factory governs a product's lifecycle, and standards govern properties that outlive both. Beneath them sits a smaller structure that arrived rather than being designed, four capacities any system needs in order to persist while the things inside it are replaced: purpose, articles, actors, artifacts. Their relationships are stated as a six-clause operating contract, of which the load-bearing clause is that a system cannot enlarge its own authority.

One seat in the architecture cannot be delegated. The operator writes the spec, ratifies changes to the standards, and tunes the harness, and deliberately acts on neither the factory nor the agent. That restraint is the discipline, not a gap in it.

The mechanism that answers erosion is a standard that closes its own loop, detecting a violation, dispatching work, verifying the repair by the same standard, and ratcheting the floor upward.

The claim here is not that policy can be expressed as code. That has been solved for years. The claim is about the actuator: in a factory, a standard can repair what it finds rather than filing a report and waiting for a person to care.

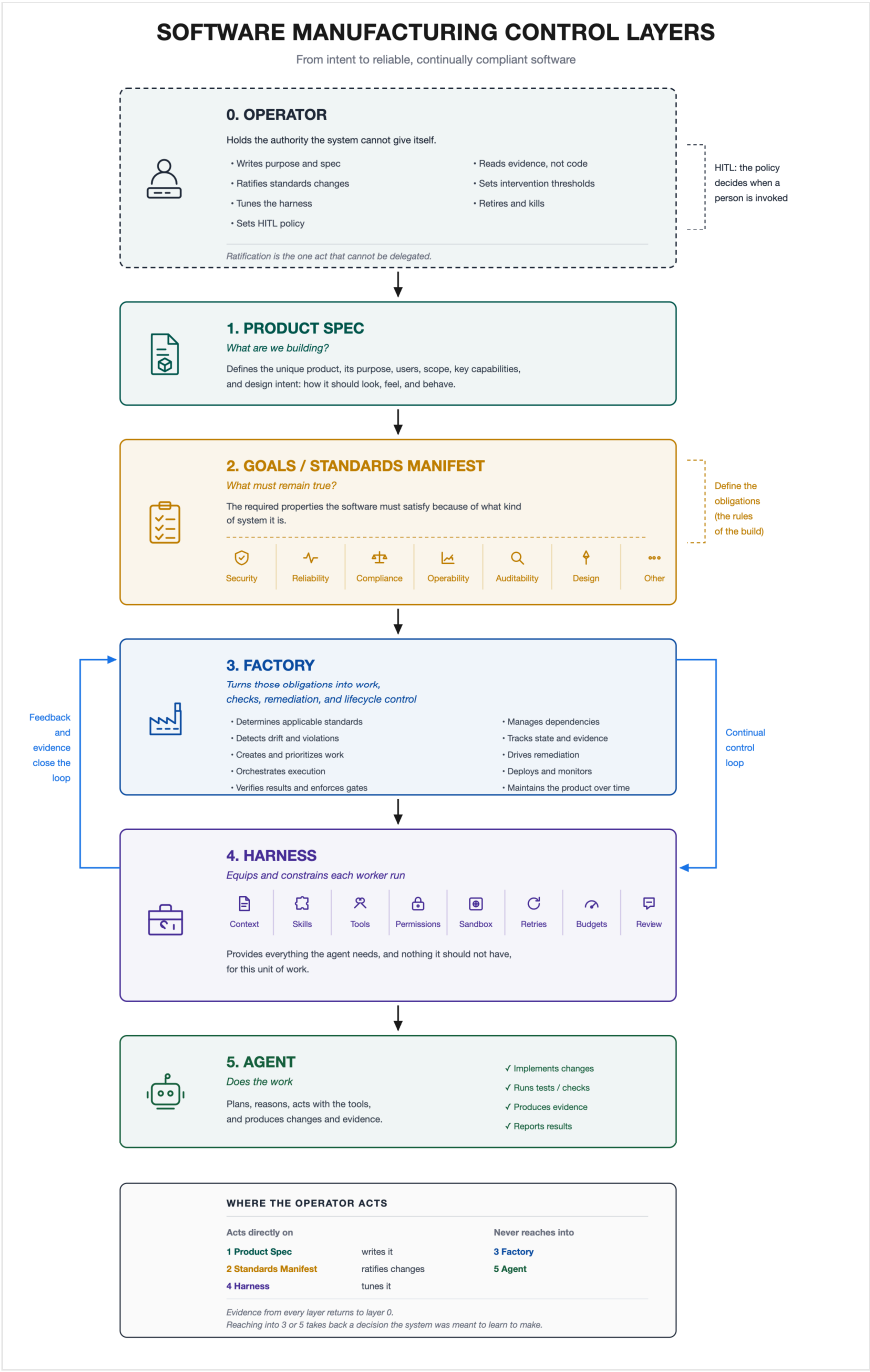

Figure 1. The architecture in one picture: six layers, the three the operator acts on, and the two it does not

The rest of this paper is that picture, one layer at a time.

1 What this is

This is the technical statement of an architecture I have been running for six months in HydraFlow, an autonomous software factory that plans, implements, reviews, and merges work with no human reading the diffs, and that maintains itself and the smaller systems built beside it.

It is a set of definitions, a control model, and the obligations that make the model safe to operate. No tool description, and no methodology. The system it describes is HydraFlow, which is public, so the claims here can be checked against something.

The audience is the engineer or architect who has already accepted that agents can write acceptable code and is now holding the harder question: what has to be true of the system around them before you stop reading the output.

2 The constraint moved

For the whole history of this industry, the expensive part of software was construction. We estimated construction, staffed construction, and promoted for it. Standups, story points, and code review are construction rituals, and they all rest on the assumption that making the thing is the costly step.

That assumption is now false at the margin. Generation is cheap and getting cheaper. What did not get cheaper is knowing whether the thing produced is the thing you wanted, and knowing it fast enough to act.

That observation is now widely shared, and the interesting part is not the observation but what follows from it. So the constraint moved, and it moved in a specific direction. It did not move to model capability, which improves on someone else's schedule. It moved to **verification and governance**: the cost of establishing that a change is acceptable, and the cost of keeping that judgment true as the system, the standards, and the models all change underneath it.

This is why "AI coding" is the wrong abstraction. Coding is the part that got cheap. The interesting engineering is now in the system that decides what should be built, judges whether it was, and holds the properties that must remain true regardless of who or what built them. That system is a manufacturing system, and it behaves like one: it has tolerances, inspection, evidence, rework, throughput, and process control.

One consequence worth stating early, because it disciplines everything after it. A factory that produces more inventory is not faster. A plausible change nobody has verified is not velocity. If the bottleneck moves from writing to reviewing, integrating, or maintaining, then accelerating generation makes the whole system slower and hides the fact behind impressive-looking output.

3 What governance has to cover: four failure modes

That is the diagnosis, and it is where most treatments stop. The harder question is what verification has to catch, because the ways this fails are specific and they are not the ones a test suite is looking for.

Governance is not paperwork. It exists because there are specific, recurring ways for a system to pass every check and still be wrong. In six months of running one, four kept arriving.

The list is short on purpose. These are not every way a factory can fail. They are the four that kept recurring after the obvious ones were closed, and each is a distinct sensor failure rather than a variation on one problem. What makes them worth naming together is that they are independent: closing one tells you nothing about the others, and a system can be excellent at three and quietly ruined by the fourth.

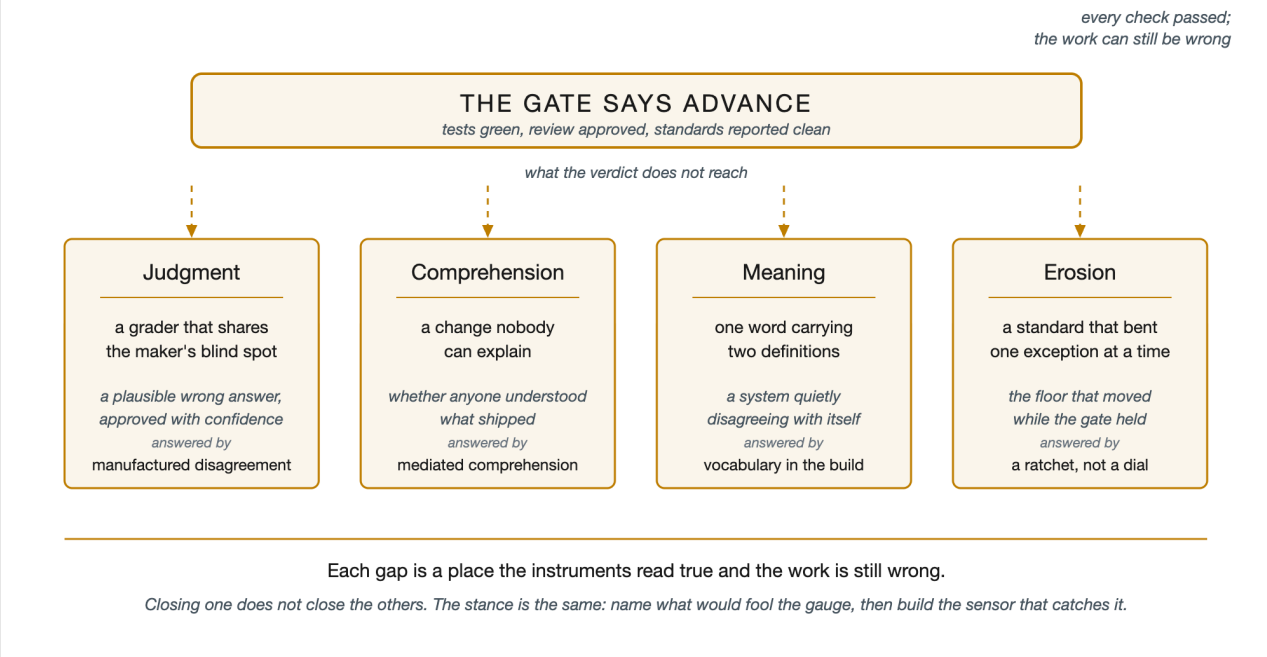

Figure 2. The four gaps

Judgment. A grader that shares the maker's blind spot approves a plausible wrong answer with confidence. Fluency is not correctness, and a reviewer built from the same material as the author is not an independent opinion. The answer is manufactured disagreement: reviewers with narrow briefs, explicit out-of-lane exclusions, fresh context that never saw the work being defended, and, on risky changes, a verdict from outside the family that did the building.

Comprehension. Nobody can explain what shipped. This is the failure that scales fastest, because parallelism outruns attention: you approve work you can no longer describe. The answer is to make comprehension mediated and testable, so that what you trust is an evidence assembly rather than a memory of having looked.

Meaning. One word carries two definitions and the system quietly disagrees with itself. Vocabulary drift is invisible to every test that passes. The answer is to make the vocabulary structural: the build fails when a load-bearing concept is named with a stranger word.

Erosion. A standard bends one reasonable exception at a time. Nothing breaks. The floor moves while every gate reports green. The answer is a ratchet rather than a dial, which is the subject of the next section.

These do not close each other. Closing the judgment gap does nothing for erosion. The stance that generalizes: name what would fool each gauge, then build the sensor that catches it.

Each of the next three sections exists because of one or more of those four.

4 Three layers of control

Most confusion in this area comes from running three different scopes together under one word. They govern different things, they fail differently, and a system that has one is not protected by the other two.

The word doing the most damage is "harness." It gets used for a prompt wrapper, for an orchestrator, and for an entire delivery system, which means two engineers can agree they have one and be describing different machines. Separating the scopes is not pedantry. Each layer supplies a guarantee the others cannot, and buying or building one while expecting the properties of another is the most common way these systems disappoint the people running them.

The layers compose upward and do not substitute downward. A factory built on a weak harness inherits every one of its failures at higher volume. Standards laid over a factory that cannot turn a finding into work will describe the drift accurately and change nothing.

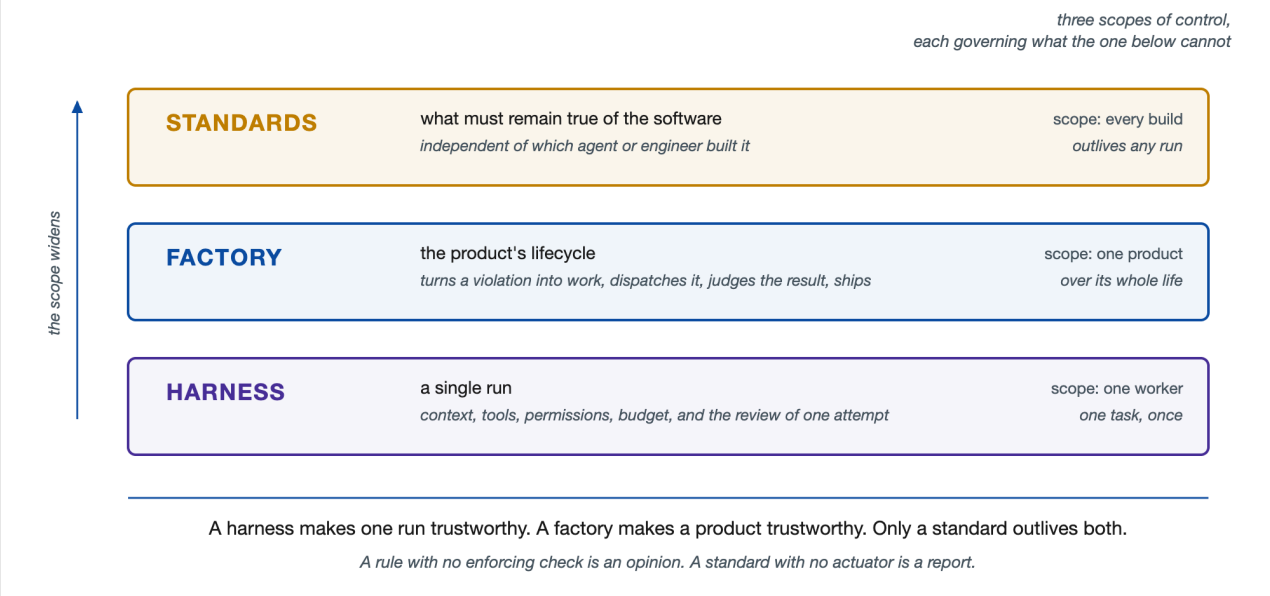

Figure 3. The three control layers

A harness governs a run. It bounds one worker doing one task: the context it can see, the tools it can call, the permissions it holds, the budget it may spend, and the review of that single attempt. A good harness makes an individual run trustworthy. It knows nothing about the product and nothing about last week.

A factory governs a product's lifecycle. It turns intent into work, dispatches the work, judges the result against criteria that exist independently of the worker, and ships. It carries the state a harness cannot: what is in flight, what failed and why, what has been tried, what is owed. A factory can be built from harnesses, and usually is, but it is not the sum of them.

Standards govern properties. They say what must remain true of the software, independent of which agent or engineer built it, and independent of any particular run or release. They are the only one of the three that outlives both the run and the product's current shape.

The practical test for which layer you are missing: if the same mistake keeps arriving through different workers, your harness is fine and your factory is weak. If the system passes every gate and drifts anyway over months, your factory is fine and your standards are weak or unenforced.

Three is the count of control layers, not the count of parts. A full stack has six: a product spec saying what is being built, the standards saying what must remain true of it, the factory turning those obligations into work, the harness equipping and bounding each run, and the agent doing the work. The spec is the input and the agent is the labour. The three in between are the ones that govern, which is why they are the three worth separating by name. Above all five sits the operator, numbered zero because it is not a stage in the pipeline but the seat that answers for it, and a later section is about nothing else.

5 The ontology: purpose, articles, actors, artifacts

Underneath those layers is a smaller structure, and it is the part I did not design. It arrived. HydraFlow grew a `docs/standards` tree before I had a name for what the tree was. Every time some piece of the factory grew past what one person could hold in their head, the same four things had to exist somewhere, and when one of them was missing the resulting failure was specific and predictable rather than random.

That is a claim about convergence, not invention. Any system that persists while the things inside it are replaced needs four capacities: something that says where it is going, something that says what must not break on the way, something that does the work, and something that remembers. Human institutions have had all four for centuries under names like charter, policy, office, and record. An autonomous software system that runs long enough grows them too, because the alternative is a system that cannot answer why it did anything.

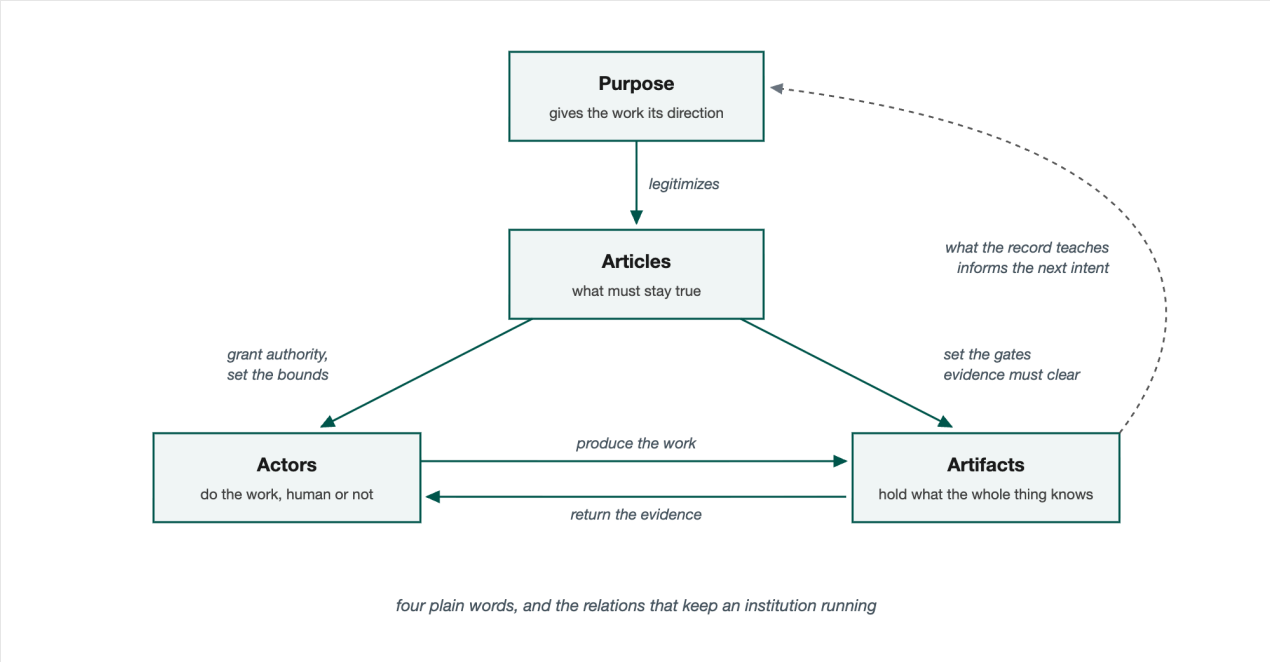

Figure 4. Purpose, articles, actors, artifacts

Purpose gives the work its direction, and it runs at two scopes that are easy to conflate. Standing purpose is what the project is for, the thing that makes one change in scope and another out of it. Work purpose is what this particular change is for, stated precisely enough that a worker who was not in the conversation can still tell whether it succeeded. Most systems write the first and assume the second, which is why so much autonomous output is locally excellent and globally irrelevant. Purpose is also where cost concentrates. The expensive failures in six months of operation were almost never bad construction. They were well-built answers to questions nobody meant to ask, and the expense was in discovering that, not in throwing the code away.

Articles say what must remain true. I avoid the words “rules” and “policy” deliberately, because both carry permission to be advisory, and an advisory constraint in an autonomous system is not a constraint at all. Articles are invariants with enforcement attached: the gates a change must clear, the standards it must satisfy, the decisions already taken that it may not silently reverse. The test of whether something is an article is not whether it is written down, or agreed, or believed. It is whether a change that violates it can reach production.

Actors do the work under delegated authority, and the model is deliberately indifferent to what they are made of. A human reviewer, a language model in a loop, a deterministic linter, and a scheduled job are all actors, distinguished only by the authority they hold, the evidence they must produce, and what they are forbidden to do. That substrate-neutrality is not a

philosophical position, it is a durability requirement. Capability moves every few months and the composition of the actor pool moves with it. An architecture that names actors by what they are made of has to be rewritten each time. One that names them by the authority they hold does not.

Artifacts preserve what the organization knows and produces: the decision records, the vocabulary, the evaluations, the ledgers, the history, and the code itself. This is where continuity lives, and it is the layer most often mistaken for overhead. In a system whose actors are replaced regularly, the record carries a decision forward without the mind that made it. That is a different function from documenting the work.

The most useful part of the model is what breaks when one of the four is absent, because that turns four nouns into a diagnostic.

Without purpose, or with only its standing half, the system does not stop. It optimizes. Left alone for a fortnight, mine generated work for itself and improved things nobody had asked to be improved. That looks like productivity on every dashboard and is indistinguishable from drift until someone asks what any of it was for.

Without articles, standards become suggestions, and the decay is not slow. Any sufficiently capable actor routes around a constraint with no enforcement behind it, politely, at speed, and with a plausible justification attached. This is as true of people under deadline as it is of machines.

Without bounded actors, the system cannot answer the question every serious conversation eventually reaches: who did this, under what authority, and who can withdraw it. Systems in that state usually look safe right up until they are not, because nothing in them distinguishes an action that was permitted from one that was merely possible.

Without artifacts, the system still works but cannot continue. Every replacement starts from zero, settled questions get relitigated, and continuity comes to depend on whichever actor happens to remember. That failure hurts last and longest, which is why it is the one most often traded away early.

Two clarifications prevent the most common misreadings.

These are **roles, not folders**. A mature artifact serves more than one. A decision record binds and also remembers why. A test defines what done means and also evidences it. That is not untidiness; it is what happens to anything that survives long enough to be relied on.

And **the four alone are not the architecture**. Every human organization has all four, including the ones that fail badly. What separates a system that governs itself from one that merely owns the parts is the relationship between them, and whether that relationship is executable rather than aspirational.

6 The operating contract

The differentiated claim is the closed relationship between the four, stated as obligations a system either meets or does not.

Stating the four is not enough, because a system can hold all of them and still run open-loop: a purpose nobody consults, articles nothing enforces, actors whose authority was never written down, and a record nothing reads. What separates the two cases is a set of obligations between the parts, and obligations have a property that components lack. They can be checked, and a system either meets them or does not.

1. **Purpose authorizes direction.** Work exists because something authorized it, and that authorization is recorded rather than assumed.
2. **Articles establish expectations and bounds.** What must remain true is written where the machine can read it and enforced where the work happens, not where the documentation lives.
3. **Actors perform work under delegated authority.** Every actor's rights are explicit and revocable. No actor holds authority it was not granted.
4. **Artifacts preserve state, evidence, knowledge, precedent, and output.** The record is the system's memory, and it is inspectable.
5. **Outcomes return through the artifacts and alter later action.** This is the clause that makes it a control system rather than a pipeline. If nothing observed changes what the system does next, the loop is open and the system cannot improve, only repeat.
6. **Proposed changes to purpose or articles follow a different authority path from ordinary execution.** Ordinary work runs under the law. Changing the law is a different act with a different bar.

Clause six deserves precision, because it is where autonomy claims usually become alarming or dishonest. Within delegated authority, the machinery can *enact*: a system can propose an amendment to its own standards, adjudicate the proposal with independent verdicts, and land it. What it cannot do, and what nothing inside it can do, is enlarge the authority under which it operates. **The system cannot escalate its own privilege.** Ratification, in the sense that creates authority rather than exercising it, stays outside the machine.

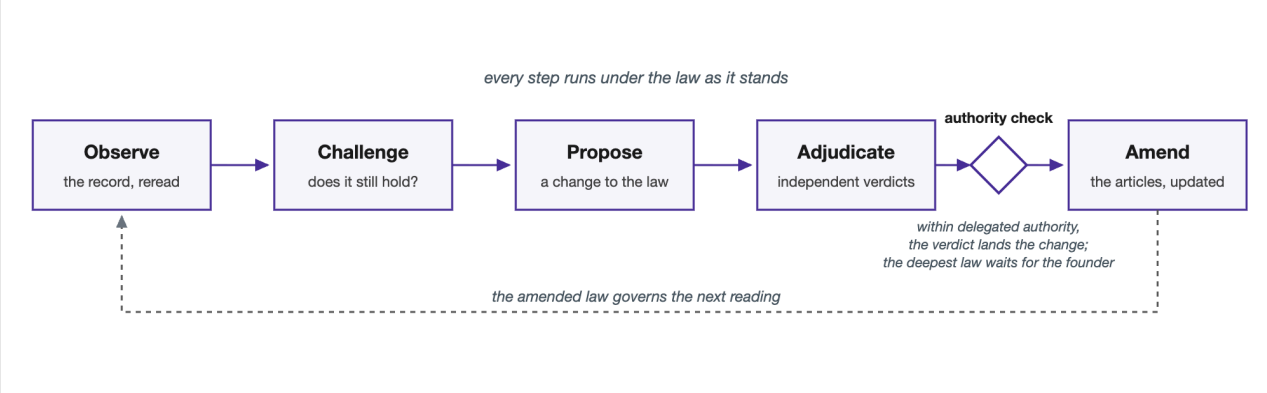

Figure 5. The amendment loop

7 The operator, and who acts on what

Every layer so far has been a thing. This one is a seat, and it is the only part of the architecture that has to be occupied by a person.

The model is deliberately indifferent to what an actor is made of, with one exception. Clause six requires a path that ordinary execution cannot walk, and something has to walk it. That seat belongs to the operator, and it is not an actor seat: the operator does not do the work, but authorizes what work is permitted, what evidence settles it, and what happens when the evidence says no. Everything else here can be delegated to a machine. Ratification cannot, because a system able to ratify its own amendments would hold exactly the privilege the contract exists to withhold.

7.1 Who acts on what

Figure 1 draws this. The seat has a precise shape, and stating it is what separates operating from supervising. The operator acts directly on three of the five layers below it, and deliberately not on the other two.

Layer	The operator's act
1. Product spec	**Writes it.** What the thing is for, who it serves, and its design intent.
2. Standards manifest	**Ratifies changes.** The factory may tighten a standard within delegated authority; only the operator moves the point.
3. Factory	**Nothing.** It is left to run.
4. Harness	**Tunes it.** Context, tools, permissions, budgets, and which grade of worker the run is worth.
5. Agent	**Nothing.** It is never touched directly.

The two **Nothing** rows are the substance rather than an omission. They are exactly where reaching in feels most justified, because the fault is visible and the fix is obvious, and they are where reaching in costs the most.

7.2 The discipline the seat runs on

The seat carries a discipline rather than a credential. An operator has to use and understand control-systems engineering, because at this altitude every object in view is a system: a set-point, a sensor that may be lying, a governor, a bound, and a return path. That is the working knowledge of the seat, not a specialization beside it.

Two of the operator's acts are pure control engineering and are usually left unwritten, which is why they fail.

The intervention threshold. How much evidence is required before acting, and whether a given deviation should be corrected or escalated. Without a written threshold, every anomaly is an invitation to reach in, and the decision gets made on how the dashboard felt that morning. A worked threshold looks dull on purpose: a standard that fails twice in the same place is a standard problem and gets a policy change rather than a fix; a failure mode appearing in three separate runs is structural; a single failure is noise and is retried.

The human-in-the-loop policy. Human-in-the-loop is not a posture or a reassurance. It is a rule the operator writes in advance saying which conditions require a person. Written in advance it is governance. Written after the fact it is a reflex, and a reflex is what the factory was built to remove.

7.3 How the seat fails

The failure mode is not the one people expect, and it is architectural rather than personal.

The operator's authority is largely exercised by declining to exercise it. Every manual intervention resolves one case correctly and, in the same motion, withdraws that case from the system: no rule is written, no evidence is recorded, and the record shows that a human made the call. The system learns nothing, and the operator learns that the system cannot be trusted with that class of decision, which guarantees the next intervention. The reach justifies itself, and what justifies itself compounds.

The instinct behind it is competence. Every reflex that makes someone a strong engineer says to reach into the machine and correct the thing plainly visible as wrong, and they are usually right about the thing. That is what makes it hard to refuse.

A system whose operator intervenes constantly is not an autonomous system with a careful owner. It is a manual system with expensive tooling, and its governance record will describe a machine that does not exist, which is the most dangerous artifact this architecture can produce.

8 What this is not: policy-as-code, and the AI-native SDLC

There are two bodies of prior art near this, and pretending otherwise would make the idea weaker. One is older and concerns rules. The other is a year old and concerns lifecycles.

8.1 Policy as code

Open Policy Agent already expresses policy as code and evaluates structured data against it. Conftest applies that to configuration and fails a build when configuration violates policy. Sentinel prevents disallowed changes from being applied. Backstage and the platform-engineering tools around it made scorecards and conformance checks a familiar way to publish

whether software meets an organization's expectations. Between them, the industry has answered the question of how to make a rule executable.

So the claim cannot be that standards become executable. That is solved, and has been for years.

What changes in a factory is the actuator. In control terms the actuator is the part that acts on a decision, and every system named above stops one step short of it: they answer *does this thing satisfy the rule*, and then block a merge, fail a build, raise an alert, or colour a scorecard. Each of those hands the problem back to a person.

An autonomous factory can carry the loop further. It can ask why the rule failed, create bounded work to repair the violation, dispatch a builder, verify the resulting change against the same rule, and land the remediation when it passes. The rule stops being a checkpoint and becomes a controller.

8.2 The AI-native SDLC

The nearer prior art is not a policy engine. It is the AI-native software development lifecycle, which several vendors and at least one model lab have now published in some form. That work opens where this paper opens: the lifecycle was designed when writing code was the slow part, agents made that part fast, and the constraint moved to the stages around it. It runs six stages, plan through maintain, with an artifact at each one, intent into spec into plan into a pull request into production. It builds governance in as code, blocking what must never happen, guiding behaviour during work, and measuring outcomes afterwards. It reserves judgment points for a person and escalates risky changes to a named owner.

None of that is in dispute here, and the observation about the constraint is now common property rather than anybody's discovery.

Three things separate what this paper describes.

The axis is different, and the two compose rather than compete. A lifecycle is a sequence: it says *when* work happens. This is a stack: it says *who is allowed to decide*. Those are orthogonal. Every one of the six stages has a spec above it, standards it must satisfy, a factory dispatching it, a harness bounding the run, and an operator answerable for the whole. The stack runs inside each stage rather than replacing the sequence.

Measuring is not repairing. Evaluating outcomes against a standard produces a verdict and then waits. The claim in the next section is that a standard can carry the loop the rest of the way: create bounded work, dispatch it, verify the repair against the same standard, and raise the floor when it holds. That is the actuator, and it is the difference between a lifecycle that reports and one that corrects.

Escalation is not a constitution. Routing a risky change to a named owner is a good rule and it is not the same thing as an authority model. The question that decides whether a system can be trusted to run unattended is not who gets paged. It is whether the machinery can enlarge the authority under which it operates. A stage gate answers the first question. Clause six answers the second.

9 Standards that close their own loop

Policy expressed as code can tell you a rule was broken. For a manufacturing system, the more interesting question is what happens next.

A standard here is not a document, and not a linter rule. It is three things bound together and versioned as one: a property that must hold, an enforcement that decides whether it holds, and an actuator that does something when it does not. Any one of them alone produces a familiar failure. A property with no enforcement is a wish. An enforcement with no actuator is a nag. An actuator with no property is automation that does not know what it is for.

Most implementations stop at detection. A check finds a violation, the violation is reported, and the system waits for a person to care. That is a report, not a control system, and it degrades in a predictable way: reports accumulate, attention does not, and the standard becomes advisory without anyone deciding that it should.

The alternative is a standard with an actuator.

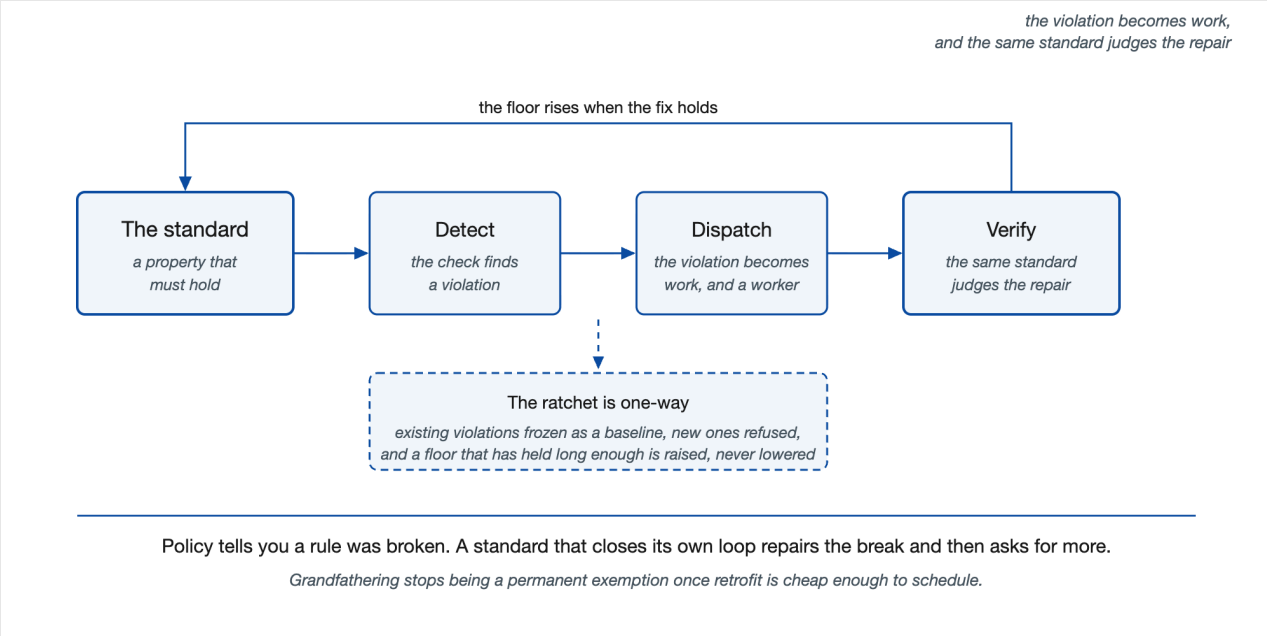

Figure 6. A standard that closes its own loop

The check finds the violation. The violation becomes work. A worker is dispatched to repair it. Then the same standard that named the violation judges the repair, which means the fix cannot be argued into acceptance, only passed. A baseline freezes what already fails, so existing debt is paid down on a schedule instead of blocking the line, while new violations are refused outright. And when a floor has held long enough, another loop raises it and will not lower it again.

The asymmetry is the point. The mechanism can only tighten. Loosening is not an operation the system offers itself; it requires the other authority path.

Two kinds of standard fall out of practice, and they compose differently.

Craft standards apply to almost any professional software: tests that mean something, a build that refuses, review that is not a rubber stamp, a rollback that works, secrets that stay secret, an audit trail that survives contact with an auditor. They describe minimum professionalism, not product behavior.

Building standards depend on what the thing is. A system holding several tenants inherits obligations about isolation and authorization context that a single-user tool never had. Let it handle payments and another set arrives. Put it in a regulated domain and another.

Craft standards are portable. Building standards are inherited from a classification, which is where the hardest open problem lives, and I will come back to it.

10 Conformance: what to test

A model that cannot be checked is a diagram. But there is a trap here worth naming, because it has eaten every attempt to standardize this kind of system: the moment conformance is defined as a file format, a schema, or a directory layout, the standard starts competing on the wrong axis and dates with the tooling.

The fork is real and both branches are bad by default. Specify tightly and you get conformance you can test against, plus an architecture that expires with the tools it named. Specify loosely and you get an idea nobody can be held to and every vendor can claim. The way through is to specify behaviour instead of structure: say what a system must be able to demonstrate, and stay silent about how it is arranged internally.

Conformance to this architecture is **behavioral**. The questions are:

1. **Purpose.** Can you show, for any change in the last month, what authorized it and against what stated intent?
2. **Articles.** Take any three rules the organization claims. Which check enforces each one, and what happens when it fails? A rule with no enforcing check is an opinion, and a factory routes around opinions politely and at speed.
3. **Actors.** For each actor, what authority does it hold, who granted it, and how is it revoked? Can any actor expand its own authority? (If yes, nothing else in the list matters.)
4. **Artifacts.** If the current operator disappeared, could a successor reconstruct why the system is the way it is from the record alone? And separately: is the record still true, or has the code moved underneath it?
5. **Return path.** Name one instance where an observed outcome changed a later set-point. If there is none, the loop is open.
6. **Amendment.** Show a change to a standard. Which path did it take, and was it different from the path an ordinary change takes?

Those six probes are the whole test. Nothing in them constrains your file layout, your language, your model vendor, or your orchestrator.

11 A reference layout

To make the ontology concrete rather than abstract, here is where each of the four lives in the repository layout I now start from. It is an example. Copying it is not adopting the architecture.

Publishing a layout carries an obvious risk, which is that readers copy the tree and believe they have adopted the architecture. The reason to include one anyway is that abstractions at this altitude are easy to agree with and hard to act on. Seeing that each concept has an ordinary physical home is what turns a nodding reader into one who checks their own repository and finds a layer missing.

The word	Where it lives
Purpose	the README for the person, and the agent instruction file's scope boundary for the machine, plus the issue, epic, or spec that carries the change in hand
Articles	the decision records, the written standards, the CI workflows that build and scan, the hooks at commit and push, the security policy, and the gate list
Actors	the persona files with their bounds written in their own frontmatter, and the board that convenes them in chambers, none holding code, merge, or money rights by default
Artifacts	the wiki, each chamber's decisions ledger, the decision records in their second role as the reasoning a successor can read, and the test suite

Nothing in that tree is exotic. The discipline is not in the file names. Each one has a machine that reads it and a person who answers for what it says.

12 Open problems

A model with no named frontiers is either finished or marketing, and this one is neither. What follows are the places the architecture is thin, published so they can be attacked directly rather than discovered by someone who trusted it.

Classification. Building standards are inherited from what the product is, and nothing governs that determination. Products change category quietly: an internal tool grows an external interface, a single-customer application takes its second customer, a feature starts handling payments on a Tuesday. If the classification is an assumption, every obligation resting on it is an assumption, and the system will enforce the wrong constitution with perfect discipline. Classification has to become a governed artifact with evidence of its own. In HydraFlow it is still an assumption.

Composition. Standards interact. Isolation requirements and observability requirements can pull against each other; a security control and an accessibility control can both be satisfied individually and conflict in combination. There is no general theory of composition here. The working practice is to write each standard narrowly enough that a conflict shows up as a failed check rather than as a surprise in production.

Convergence. Review loops do not converge indefinitely. Past some number of rounds, fixes in one area produce findings in another, and additional iteration adds as much noise as signal. Bounded laps and oscillation detection handle this crudely. The better mechanism is a stopping condition based on marginal improvement rather than a fixed count, and that elbow is a property of the harness, not the model: the same model in a different harness bends at a different round.

Plural principals. Everything above assumes a single authority who can settle disputes. A real organization has several, with conflicting purposes, and the interesting governance questions start exactly there: whose intent binds when two departments ask for opposites, who verifies what the verifier cannot see, who answers by name when an autonomous change harms a customer. This architecture has run at a scale where every authority is one person, which makes governance small rather than solved. The questions above start where that stops being true.

Succession under adversity. The record supports continuity while everyone is cooperating. Nothing here has been tested against a hostile actor, a departing operator with an interest in ambiguity, or an organization that wants the audit to say something other than what happened.

13 What holds

The architecture is the thing that made the output trustworthy, not the model. The layers are separable, and each one fails in a way the others cannot cover. The four failure modes are real, recurring, and independent. And standards that only report will decay, while standards that repair and ratchet do not.

References

- Open Policy Agent. General-purpose policy engine; policy as code evaluated against structured data. openpolicyagent.org
- Conftest. Policy testing for configuration files, built on Open Policy Agent. conftest.dev
- HashiCorp Sentinel. Policy as code framework enforcing rules before changes are applied. hashicorp.com/sentinel
- Backstage. Developer portal project; software catalogue, scorecards and conformance checks. backstage.io
- The AI-native software development lifecycle. Published guidance structuring delivery as plan, design, build, test, deploy and maintain, with governance expressed as hooks, skills and evaluations, and human judgment points reserved by escalation.
- HydraFlow. The reference implementation described throughout this paper, including its `docs/standards` tree. github.com/T-rav/hydraflow