Chapter 3

Why observability, harnesses, and evals don’t catch it

A reasonable reader of the first two chapters might reasonably ask: are you sure the existing tools don’t already catch this? The industry has invested enormous engineering effort in observability, agent harnesses, evaluation frameworks, policy-as-code, model monitoring, and a dozen other tools. Each of these catches real failures. Are you sure that between them they do not already catch decision-basis failure?

I am sure, and this chapter explains why. Each of these tools was designed to catch a class of failure that is structurally different from decision-basis failure. They do their job well. Decision-basis failure is simply not in the set of failures they are built to catch. What follows takes each major category of tooling in turn: what it catches, what it leaves open, and why the gap it leaves is where decision-basis failure lives.

Observability

What it catches. Observability — the modern umbrella term covering logs, traces, metrics, and the tooling built on top of them — catches *what happened*. It reconstructs the sequence of events: which service was called, with what arguments, at what time, with what response, with what latency, with what error, if any. In the agent context, observability tools render agent traces as a navigable timeline: this prompt went to the model, this response came back, this tool was called with these parameters, this result was returned, this next action followed.

Observability is extraordinary at diagnosing after the fact. When an agent system misbehaves and you need to understand *how*, observability is the first and often sufficient tool. The modern agent observability stack — Langfuse, Helicone, LangSmith, Datadog’s AI monitoring, and others — gives engineers a view into agent reasoning that was simply not available two years ago.

What it leaves open. Observability is descriptive. It tells you what happened. It does not tell you whether what happened was *right*. If every tool call succeeded, every response came back as expected, every step followed the plan, observability renders a healthy trace. A healthy trace of a decision-basis failure looks identical to a healthy trace of a correct decision. You cannot distinguish

them by looking at the trace, because the thing that was wrong — a missing consideration — leaves no footprint in the trace.

An observability tool that could flag decision-basis failures would need to compare the trace not just to its own prior patterns but to a specification of what the decision *should have* considered, and no such specification exists in the observability layer. The specification lives one layer up, in whatever document the responsible party for this domain has in their head or in Confluence. Observability does not reach there.

Agent harnesses

What they catch. Agent harnesses — the frameworks that structure how agents call tools, maintain state, and sequence actions — enforce *process compliance*. They ensure the agent follows the designed workflow: the correct tools are called in the correct order, with the correct inputs, within the correct scope. Harnesses catch when an agent tries to take an action it was not authorized to take, when it attempts a tool call outside its permitted set, when it fails to follow the designed state machine.

Modern harnesses are good at this. They catch most of the obvious ways agents can go off the rails. They are the operational equivalent of a well-designed turnstile.

What they leave open. Harnesses enforce that the designed workflow was followed. They do not have an opinion about whether the designed workflow was *complete*. If the workflow was designed to check A, B, and C, the harness ensures A, B, and C are checked. If the organization also has an unwritten expectation that D should be checked, the harness does not know about D. The harness enforces what was specified. The decision-basis failure lives in what was not specified.

This is why “patching the workflow” is the dominant response to a surfaced decision-basis failure, and why it does not scale. Each patch adds a specific D to the specified checks. The next decision-basis failure, caused by a different unspecified E, is treated as a new case. The harness layer, by its nature, cannot anticipate everything the organization implicitly expects. The implicit expectations live elsewhere.

Evaluations

What they catch. Evaluations — the benchmarks, test sets, and scoring frameworks used to measure how well an agent performs — catch *systematic performance gaps against a specified standard*. If the agent is supposed to achieve 95% accuracy on a benchmark and it achieves 83%, evals catch that. If

the agent's behavior drifts over time, drift evals catch that. If the agent fails on a specific class of input, targeted evals catch that.

Evals are essential for responsible deployment. Without them, you cannot ship with any confidence.

What they leave open. Evals score the agent against the eval set. The eval set is constructed by someone — usually a developer or product manager — enumerating what success looks like for the categories of input they considered. The categories they did not consider are not in the eval set. Decision-basis failure almost always lives in categories that were not considered, because if they had been considered they would be in the specified checks the harness enforces.

Some evaluation tools now compare an agent's execution trajectory, the sequence of messages and tool calls it produces, against a reference trajectory written by engineers. That is close in form and different in substance. The reference is a sequence of tool calls, not the organization's endorsed judgment about what a decision must weigh, what conditions matter, and when someone must step in.

The newest evaluation tools use a model to do the scoring. An LLM-as-a-judge grades outputs against a rubric; an agent-as-a-judge decomposes the task, gathers evidence and assesses the intermediate steps, not only the final answer (You et al., 2026). This is close to what Part V proposes, and the studies of how model judges fail are the reason it is not the same thing. Stress tests find prompt sensitivity, verbosity bias, self-preference, miscalibration and hallucinated verdicts (Dev et al., 2026), and the simplest rubric-based grader often matches the elaborate ones. A judge holds its own rubric, and its verdict ends the story. The verifier in this book holds no opinion of its own: it measures the distance between an agent's trajectory and a reference the organization wrote and endorsed, its response is the one the owner chose, and its own error rate is recorded in the Ledger (Chapter 18). The standard has to be external to the model, because the model is the thing that drifts.

An eval set is a kind of snapshot of organizational judgment at the moment the eval was written. The problem with institutional judgment is that it is distributed across the organization, mostly tacit, and updated continuously as new incidents happen. A static eval set captures a snapshot. It does not capture the current state of the organization's judgment. The gap between the eval and current institutional judgment is where decision-basis failure accumulates.

Policy-as-code and governance tooling

What it catches. Policy-as-code frameworks — tools like Open Policy Agent, HashiCorp Sentinel, custom policy engines — enforce *explicit policies* that have been written down in a machine-readable form. If the policy says "no deployments to production between 17:00 and 09:00," the engine enforces that. If the policy says "all claims above $50,000 require human approval," the engine enforces that.

Policy-as-code is the closest existing tool to what Governance Blueprints will do, and the two will coexist. For explicit, enumerable policies, policy-as-code works well. Its research cousin is runtime verification (Leucker & Schallhart, 2009), and that tradition has now reached agents: AgentSpec (Wang, Poskitt & Sun, 2026) and ShieldAgent (Chen, Kang & Li, 2025) both enforce rules against an agent's actions as it runs. They are the strongest existing work in this direction, and the next paragraph applies to them as much as to the policy engines.

What it leaves open. Policy-as-code evaluates the action. A rule inspects the properties of what the agent proposes to do — the amount, the time, the target — and allows or denies it. When no rule matches, the action proceeds. That default is correct for an access-control system and wrong for a judgment system, because the failures this book is about are precisely the cases no rule anticipated.

The Governance Blueprint evaluates the basis. Its nodes do not say "block claims above X." They say "a claims decision in this class must have considered the household's recent claims history." When the agent's basis lacks that consideration, the check fails, whatever the claim looks like. The Blueprint fails closed on coverage where policy-as-code fails open on action.

	Policy-as-code	Governance Blueprint
What it checks	Attributes of the proposed action	Coverage of the decision's basis
Form of a rule	If the action has property X, do Y	A decision of class D must have considered C
Unanticipated case	No rule matches, so the action passes	A required consideration is missing, so the basis fails
Who maintains it	Whoever wrote the rule	A named signer, with an expiry date
Response	Allow or deny	Seven graded interventions

Nothing stops an organization from compiling the bright-line nodes of a Blueprint — the boundaries and failures — into a policy engine, and a good implementation will. The engine is not the contribution. The contribution is what is checked, which default applies, who signs, and the process — Chapter 26 — that surfaces the considerations in the first place.

Scope, skills and permissions

Three further answers come from the configuration layer, and they fail the same way. *Scope definitions* say what the agent may touch; they do not say under what judgment the scope was approved, who signed it, or whether it reflects the organization's current judgment or a snapshot from months ago. *Skill specifications* say what the agent can do; a capability inventory is not an authority grant, and it says nothing about what the agent should do under this organization's policies or where it should escalate. *MCP permissions* say which external systems the agent can reach; they bound the operational surface without recording the judgment that was supposed to shape behavior inside it. Configuration is operator-controlled. The responsible-party signature behind it is a different artifact, and in most deployments it does not exist.

The common missing piece

Each of the tools I have described catches a specific class of failure. Together they catch most of the obvious ways agent systems can go wrong. The common missing piece — the thing none of them was designed to catch, and the thing decision-basis failure specifically exploits — is a mechanism for verifying that the agent's basis for a decision matches what the responsible party for that class of decision in this specific organization would have required.

The missing piece has four properties that no existing tool combines.

It is per-organization. What matters for a release decision at Company A is not what matters at Company B. The verification cannot be a generic standard; it has to be this organization's judgment.

It is per-domain. What matters for a release decision is different from what matters for a claims decision or a procurement decision. The verification is domain-specific.

It is per-responsibility. Within a domain, specific aspects of judgment are owned by specific parties. Data engineering owns the schema rule. Compliance owns the beneficial-ownership rule. Fraud owns the pattern-of-claims rule. The verification mechanism must respect this division of authority, so that each party signs the aspect they own and no one is accountable for aspects outside their domain.

It is real-time and architectural. The verification cannot be a quarterly audit or an end-of-day batch. It has to happen before the agent acts, every time, at machine speed, consistently.

No tool in the current stack has all four properties. The Governance Blueprint architecture, described in Part V, is designed around them.

What needs to be built

The next step, if you accept that decision-basis failure is real and that existing tools do not catch it, is to specify what needs to be built. The specification has two sides — an organizational side and a technical side — and they must be developed together.

On the organizational side: a process for extracting institutional judgment from where it currently lives (people, documents, prior incidents) and encoding it in a form that can be signed by the responsible party for each piece. This is the direction designer role described in Part VI. It is a new professional practice. It does not exist in most organizations today in any formalized way.

On the technical side: an architecture that stores the signed judgment as a graph, evaluates each agent decision against the relevant subgraph at an appropriate scrutiny level, and responds to detected failures through a graduated intervention ladder. This is the real-time counterweight architecture described in Part V.

The two sides are inseparable. A Governance Blueprint without the organizational process to populate it is an empty database. An organizational process to extract judgment without the technical architecture to enforce it produces documents that no one consults. The two together are what close the gap.

Before we specify the architecture — which is Part V's job — we need to understand *why* the inherited management framework cannot do this job, and *what* theoretical reconstruction is required. That is Parts II, III, and IV. We turn to them now.

The full pre-release edition of *The Decision* and every chapter are available from MeaningStack. Contact: luciana@meaningstack.com