Agent guardrails, action gates, harnesses, and governance: four layers, four different jobs

Abhishek Tiwari

Citation: *A. Tiwari*, ”Agent guardrails, action gates, harnesses, and governance: four layers, four different jobs”, Abhishek Tiwari, 2026.
doi:10.59350/ryy8g-0qp38

Published on: May 02, 2026

There's a category error spreading quietly through enterprise AI teams: treating content filtering as agent safety. Guardrails are tangible. You can demo them, test them, point to a vendor. They produce outputs that look like security. And they do real work - blocking prompt injection, catching PII, filtering off-policy content at the LLM boundary. Nobody serious argues against them.

The problem is what they can't see. A guardrail operates on text. It has no view of what that text is about to do - which API it will call, which database it will write to, which downstream agent it will spin up. An output can pass every content check and still trigger an action that causes serious harm. That gap has a name. Most production deployments have nothing in it.

This post maps four distinct layers - guardrails, action gate, harness, and governance - where each one intervenes in the agent lifecycle, and why the gaps between them are where things actually go wrong.

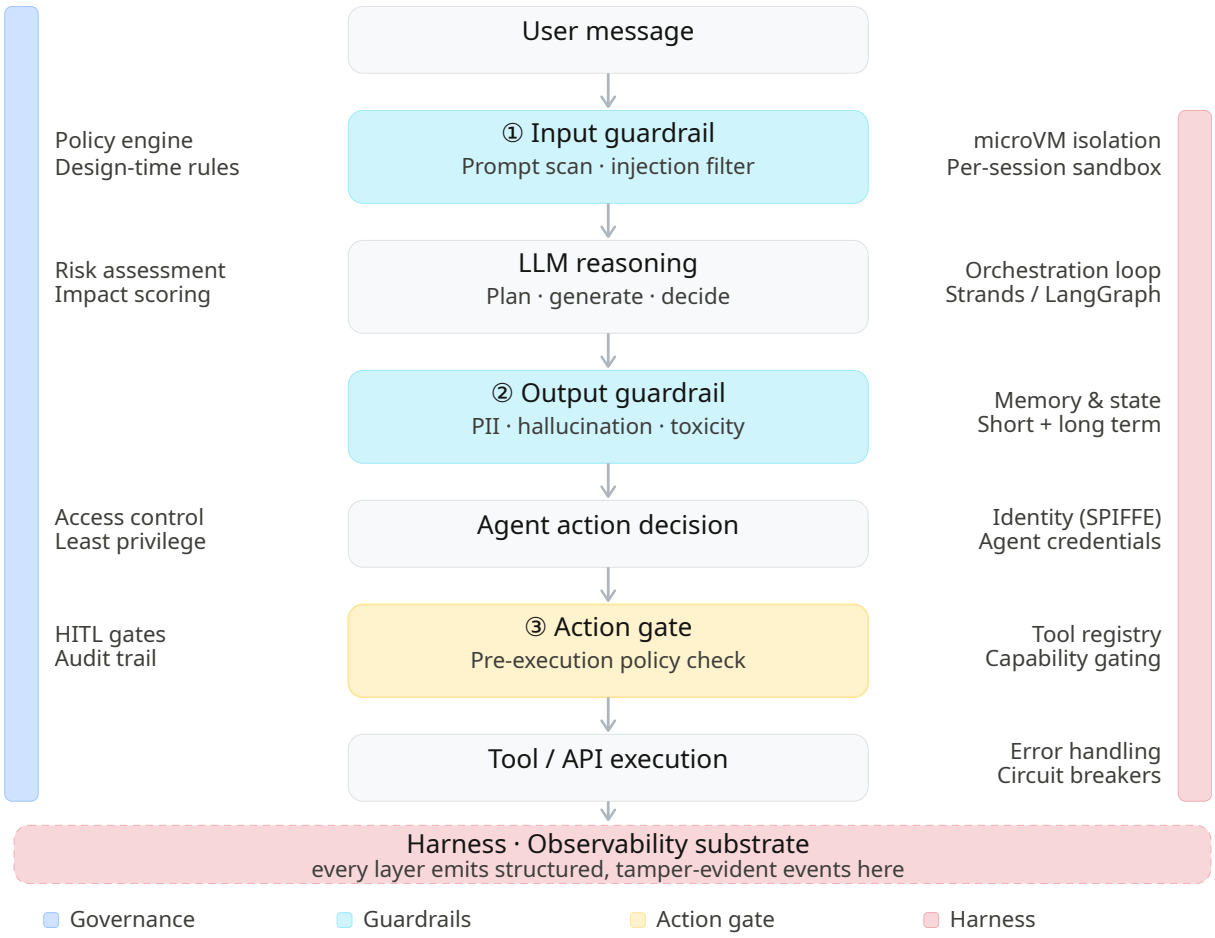

Figure 1: Agent guardrails, action gates, harnesses, and governance. One agent action, four layers: Input guardrails screen the prompt → LLM reasons and responds → output guardrails screen the response → action gate evaluates the tool call → harness executes it in an isolated environment. Governance isn't a step in this sequence — it's the policy that determines what each step is allowed to do.

Layer 1 — Guardrails: filters on the LLM boundary

Guardrails are runtime content filters. They sit at the boundary of the language model — screening what goes in and what comes out.

Input guardrails intercept user prompts and tool results before the LLM processes them. Their primary job is blocking prompt injection: adversarial content in user inputs or retrieved documents that attempts to hijack the agent's instructions. They also enforce topic constraints, validate input schemas, and sanitise data from external sources before it can poison the model's context.

Output guardrails process the model's response before it reaches the user or triggers downstream

actions. They catch PII that the model has inadvertently included in its output, flag hallucinated citations or fabricated facts, filter toxic or off-policy content, and validate that structured outputs conform to expected schemas. Tools like NVIDIA's NeMo Guardrails and Guardrails AI live primarily in this space, as do the content policy adapters in Microsoft's Agent Governance Toolkit.

What guardrails cannot do is equally important to understand: they only see text. A guardrail can pass a perfectly clean, well-formed output that describes a tool call which will cause serious downstream harm. The output looks fine. The action it triggers is not. This is the gap that the *action gate* closes — and it's worth treating as a distinct concept rather than a subcategory of guardrails, because the tools, mechanisms, and threat models are genuinely different.

Layer 2 — The action gate: the checkpoint most teams skip

The action gate sits between the agent's decision and the actual execution of a tool call. Where output guardrails ask "is what the model said acceptable?", the action gate asks "should this agent be allowed to do this right now, given what we know about its identity, permissions, and current task?"

This is pre-execution policy enforcement, not content filtering. It's the layer that maps to OWASP's Agentic Top 10 risk category ASI-02 (excessive capabilities) and ASI-03 (identity and privilege abuse). Microsoft's Agent Governance Toolkit is built almost entirely around this problem: deterministic policy evaluation running in under 0.1 milliseconds before each tool invocation, with cryptographic agent identity via Ed25519 and SPIFFE, and a four-tier execution ring model that limits what actions are even available to a given agent instance.

The reason most teams don't have this layer is partly that it's newer and less discussed than content guardrails, and partly that implementing it properly requires solving identity and permissions at the agent level — which is a harder infrastructure problem than wrapping an LLM call in a content filter. It also requires answering uncomfortable organisational questions about who owns agent permissions, how they're reviewed, and who gets paged when a policy violation fires at 2am.

Layer 3 — The harness: infrastructure, not safety

The harness is the runtime environment the agent operates inside. It is not, strictly speaking, a safety feature — it is the substrate that makes safety features reliable and production-grade.

A minimal harness has three components: an orchestration loop that drives the agent's plan-execute-observe cycle, a sandboxed compute environment that isolates execution and prevents one session from affecting another, and a state management system that tracks context, memory, and progress across a multi-step task.

AWS AgentCore, which entered public preview in April 2025, is currently the most concrete production example of what a managed harness looks like at scale. Its key design decisions illuminate what "harness" actually means in practice:

Isolation: every agent session runs in a Firecracker microVM with its own CPU, memory, filesystem, and shell. Sessions cannot bleed into each other. A compromised session cannot access another session's state.

Orchestration: the loop is powered by Strands Agents, AWS's open-source framework. It handles the reasoning loop, tool execution, context window management, error recovery, and streaming — the plumbing that would otherwise take days to build correctly per agent.

Memory: two-tier by design. Short-term memory captures raw events within a session. Long-term memory extracts durable knowledge via configurable strategies — semantic, summarisation, episodic, user preference — and makes it retrievable across sessions. The agent doesn't lose context when the underlying microVM expires.

Identity: each session can be scoped to an actor ID, which gates what memory and tools a given invocation can access. Combined with IAM execution roles, this gives you a meaningful access control model without building one from scratch.

Escape hatch: when you outgrow configuration-driven setup, you can export the harness as Strands code and continue on the same infrastructure. This is the right design choice — config-first but not config-locked.

The critical framing: without a properly built harness, your guardrails and governance are attached to an unreliable runtime. You can have excellent content filters on an agent with no session isolation, no error recovery, and no persistent memory, and still end up with cascading failures, state corruption, and security incidents that have nothing to do with what the LLM said.

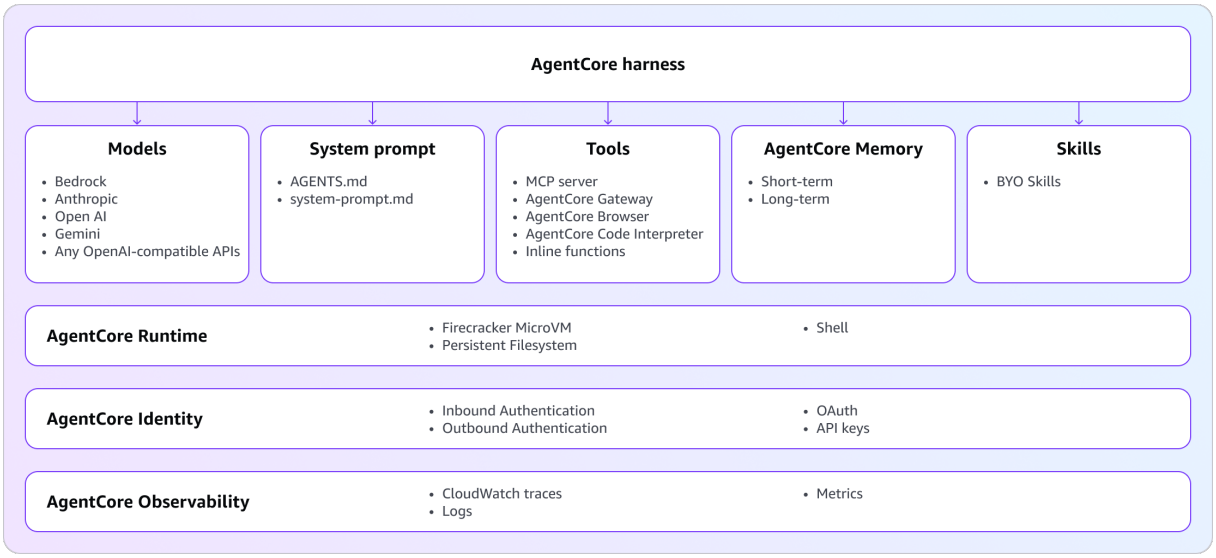

Figure 2: Amazon Bedrock AgentCore allows agent builders to declare the model, tools, and instructions. It handles the rest: isolated microVMs per session, persistent memory, MCP connectivity, VPC networking, full observability — and model-agnostic routing across Claude, GPT-4o, or Gemini, switchable mid-session without losing context.

The observability substrate

There is one more job the harness performs that most teams treat as an afterthought: providing the shared collection substrate into which every layer emits structured, tamper-evident events.

This is not observability as a separate layer sitting alongside the others. It is a horizontal capability the harness enables, and the other layers feed it. Without the harness providing a reliable, centralised collection point, each layer logs in isolation — guardrail events in one system, action gate violations in another, session traces somewhere else — and reconstructing what an agent actually did across an incident becomes an exercise in correlating four disconnected log streams.

What each layer emits, and why it matters:

Governance emits policy audit logs and HITL decision records — the compliance-grade evidence that a policy was in force, was evaluated, and produced a recorded outcome. This is what satisfies a regulator asking “how do you know your agent was operating within approved parameters?”

Guardrails emit filter events and PII detection records — a timestamped trace of every input blocked, every output flagged, every schema violation caught. Not primarily for compliance, but for safety analysis: understanding what adversarial inputs are reaching your agents and how the filters are performing over time.

The action gate emits denied tool calls and permission check records — the most operationally significant signal in the stack. A spike in denied tool calls is an early indicator of either a misconfigured policy, a misbehaving agent, or an active attempt to exceed permitted capabilities. Without this signal, those three scenarios are indistinguishable.

The harness itself emits session traces, latency metrics, error events, and resource consumption — the operational substrate that tells you whether the agent is running correctly, independently of whether it is running safely.

The flight recorder analogy is apt here. A flight recorder doesn't ask the pilot for permission to record. It captures everything regardless of whether the flight goes smoothly. The value is highest when things go wrong — which is precisely when you need a complete, unaltered, tamper-evident record of what happened and in what sequence.

Runtime vs. sandbox vs. harness

These three terms appear constantly in agent infrastructure discussions and are used interchangeably often enough that the distinctions have started to erode. They are not the same thing.

The **runtime** is the software layer that drives agent execution — the orchestration loop, context management, tool dispatch, error handling, memory access, streaming. It is code. Strands Agents is a runtime. LangGraph is a runtime. The runtime answers one question: how does the agent reason, act, and observe across a multi-step task?

The **sandbox** is the isolation boundary around the execution environment — the compute context in which the runtime operates. It is infrastructure, not code. A Firecracker microVM is a sandbox. A container is a weaker sandbox. A process is weaker still. The sandbox answers a different question entirely: what can the running code reach, and what can it affect? A runtime bug or a compromised agent cannot escape the sandbox regardless of what the runtime logic says, because the sandbox is enforced at a lower level by the hypervisor or kernel, not by the runtime itself. The runtime has no say in its own containment.

The **harness** contains both. It is the complete runtime environment an agent operates inside — the orchestration loop, the sandboxed compute, the memory system, the tool connections, the identity layer. The naming varies, but the concept is the same: everything that turns a language model into a running agent, short of the model itself.

In AgentCore this layering is explicit: Strands Agents is the runtime, Firecracker microVM is the sandbox, and together they constitute the harness. Each session gets its own microVM, so a compromised session cannot read another session's memory even if both are running identical runtime code. The sandbox enforces that isolation structurally, independently of anything the runtime does.

The distinction matters for security design. A well-designed runtime without a proper sandbox is exploitable at the infrastructure level — the runtime may behave correctly while the underlying compute context leaks state between sessions. A proper sandbox without a well-designed runtime produces an isolated but unreliable agent — contained but prone to failures the sandbox cannot prevent. Both are required, and treating "runtime" as a catch-all for all three concepts obscures which layer is responsible for which failure mode. When something goes wrong in production, the first diagnostic question is whether the failure is in the runtime logic, the sandbox boundary, or the harness configuration. Those require different fixes from different teams.

Layer 4 — Governance: the policy layer nobody owns

Governance is the layer that answers the question your guardrails and harness cannot: what should agents be permitted to do at all?

It's a design-time and organisational concern that produces policies enforced by the other layers at runtime. In practice it covers: risk assessment and tiering of agent capabilities, access control policies mapping agent identities to permitted actions, human-in-the-loop thresholds that define when an agent must pause and request approval, and audit trails that satisfy compliance, legal, and security requirements.

The reason governance is the most underinvested layer in most organisations isn't technical — it's organisational. Implementing guardrails is an engineering task with a clear deliverable. Building a harness is a platform infrastructure project. Governance requires cross-functional alignment between engineering, legal, security, and compliance teams, produces artefacts that are harder to demo than a content filter, and involves hard questions about accountability that organisations are often not ready to answer.

Microsoft's Agent Governance Toolkit covers the technical side of this problem thoroughly: policy engines using OPA/Rego/Cedar, delegation chains, compliance verification, OWASP Agentic Top 10 mappings. But the tools are only useful if the organisational processes exist to define the policies, review them regularly, and act on violations.

The most honest statement about governance: it is not a product you buy. It is a discipline you build. The tools help, but they cannot substitute for the human decisions about what your agents should and shouldn't do.

The overall landscape

These four layers correspond to distinct vendor categories, and understanding which vendor lives where prevents the common mistake of treating any one product as a complete solution.

Guardrails (LLM boundary): NeMo Guardrails, Guardrails AI. Strong on content filtering, schema validation, and output coercion. Weak on agent-level concerns like identity, sandboxing, and pre-execution policy.

Action gate (pre-execution policy): Microsoft Agent Governance Toolkit. Purpose-built for this problem, with cryptographic identity, execution rings, and OWASP Agentic Top 10 coverage. Less focused on LLM content filtering.

LLM gateway (routing and observability): LiteLLM, Portkey. Handle multi-provider routing, spend tracking, and LLM-level observability. Not agent governance tools — they don't see tool calls, they see model calls.

Harness (runtime infrastructure): AWS AgentCore, Strands Agents, LangGraph. Provide the orchestration loop, sandboxing, memory, and identity substrate. Not safety products — infrastructure products that safety products attach to.

Governance (policy and compliance): Microsoft AGT, your own platform team, and organisational process. No single vendor owns this completely.

A production-grade agent deployment typically needs components from all five categories. They are complementary, not competing.

What failure looks like at each layer

Missing input guardrails: the agent gets prompt-injected via a retrieved document, follows adversarial instructions, and exfiltrates data or performs unintended actions. Detectable in logs, but often only after the damage is done.

Missing output guardrails: the agent returns PII to the wrong user, hallucinates a citation that makes it into a customer-facing document, or produces content that violates policy. Embarrassing and sometimes legally significant.

Missing action gate: the agent decides to call an API it wasn't intended to use, because the LLM reasoned its way to a plausible justification. The output that triggered the call passed content review. The action itself did not go through any policy check. This is the most common failure mode in teams that consider themselves "safe" because they have content filters.

Missing harness: session state bleeds between users, a failed tool call crashes the agent rather than triggering recovery, or a long-running task loses context mid-execution and restarts from scratch. These failures feel like reliability problems, not safety problems — but they become safety problems quickly when the agent is operating on production systems.

Missing observability substrate: the incident has happened, the post-mortem is under way, and nobody can reconstruct what the agent actually did. Guardrail events are in one log, action gate denials

in another, session traces in a third. The question “why did this agent do that?” has no clean answer. This failure mode is invisible until the moment it matters most — which is why teams consistently underinvest in it until after their first serious incident.

Missing governance: no one knows which agents have access to which systems, approvals happen informally or not at all, audit trails don’t exist or aren’t reviewed, and the first time anyone asks “why did this agent do that?” the answer is “we’re not sure.”

Conclusion

The first three layers are, at bottom, engineering problems. You can buy a guardrail product, implement an action gate, provision a harness. There are vendors, frameworks, and reference architectures for all of it. A sufficiently resourced team can build the technical stack in a quarter.

Governance is different. No vendor ships it. No framework installs it. It requires your legal, security, compliance, and engineering teams to agree on what your agents should be permitted to do — and then to review that agreement when the agents change, when the threat model changes, when the regulatory environment changes. It requires someone to own the policy, someone to get paged when it fires, and someone with authority to say no.

Most organisations will underinvest here precisely because it’s the hardest layer to demo. A content filter produces visible output. A governance framework produces accountability, audit trails, and organisational clarity — none of which show up in a sprint review

Guardrails filter what agents hear and say. The action gate controls what they do. The harness is the environment they run in. Governance is the answer to the question all three of those layers assume someone has already answered: what should these agents be allowed to do at all? Until that question has a real owner, the other three layers are protecting a perimeter that nobody has defined.