On this page >

← Blog (https://runcycles.io/blog/)

April 11, 2026 · Albert Mavashev · 9 min read

runtime-authority governance architecture comparisons multi-tenant multi-agent observability

Copy link Share Share PDF (https://runcycles.io/pdfs/agents-are-cross-cutting-your-controls-arent.pdf)

Agents Are Cross-Cutting. Your Controls Aren't.

A SaaS platform runs an AI feature for fifty customers. Each customer's agent calls three LLM providers — OpenAI for reasoning, Anthropic for long context, a local model for cheap embedding — and eight tools — search, a payments API, a CRM client, an outbound mailer, a vector store, a document parser, a code interpreter, and a web fetcher. The agents run on a pool of stateless workers, each handling concurrent runs across multiple customers.

The platform team is asked a reasonable question by their head of engineering: **where do we put the budget cap?**

They walk through the options. OpenAI has organization-level usage limits. Anthropic has workspace caps. Their observability tool, Langfuse, gives them per-call cost attribution. LangChain's `AgentExecutor` has a `max_iterations` parameter. They also have a small Redis-backed counter the team wrote one afternoon for per-customer spend tracking.

Each of those controls exists. Each one solves a real problem. None of them, individually, and few combinations of them in practice, can answer the question that was actually asked: *can this agent — across all providers, all tools, this specific customer, on whichever worker happens to pick up the job — proceed with the next action?*

The shape of the problem is not a missing feature in any one of these tools. It is a structural mismatch. **Agents are cross-cutting by construction. The controls people reach for first are tool-local by construction.** That gap does not close by adding features. It closes by adding a layer.

This post is the *span* companion to Runtime Authority vs Guardrails vs Observability ↗, which makes the *lifecycle* version of the same argument: enforcement has to happen before the action runs. Here the question is different. How far does the enforcement reach?

The cross-cutting thesis

Autonomous agents ↗ are cross-cutting along at least four axes:

- **Providers.** A single agent run routes between OpenAI, Anthropic, Google, an embedding service, and one or two paid third-party APIs. Cost arbitrage, capability fit, fallbacks, and latency tuning all push agents toward multi-provider architectures.
- **Tools.** Beyond LLM calls, agents invoke search APIs, payment processors, mailers, databases, file stores, and arbitrary HTTP. Many of these have non-trivial per-call cost or non-trivial blast radius.
- Tenants ↗. Production agents in SaaS systems serve many customers from the same infrastructure. Per-customer isolation is a service requirement, not a nice-to-have.
- **Workers.** Agents do not run in one process. They run on pools of stateless workers, with retries, fan-out ↗, and concurrent runs hitting the same shared budget.

Each of these is independently true. Combined, they are multiplicative. A platform with three providers, eight tools, fifty tenants, and twelve worker processes has well over ten thousand distinct (provider × tool × tenant × worker) combinations to reason about. Any control that lives inside one of those dimensions is, by construction, blind to the other three.

Effective governance has to span the same surface the agent spans.

Where each "obvious" control falls short

The four categories of control most teams reach for first each fall short for the same structural reason: they govern themselves, not the agent.

Provider spending caps

OpenAI and Anthropic offer organization or workspace-level spending limits. Google Cloud and AWS provide budget alerts and service quotas that play a similar role, though as alerts and throughput limits rather than hard spend caps. All of them are useful safety nets against catastrophic monthly bills.

But a provider cap lives inside that provider's billing system. It can see what your account spent on its product. It cannot see what you spent on any other provider, on any tool, or on behalf of which customer.

A monthly OpenAI cap of $10,000 does not stop a runaway loop that burns $4,000 in three hours, because the cap measures against the calendar month, not against the run. It does not distinguish your fifty tenants from each other, because it does not know what a tenant is. It does not include the search API call, the payment processor charge, or the outbound mailer that happen between the model calls. And it does not see Anthropic at all.

The deeper point is not that any of those gaps is a feature request. Provider caps exist to protect the provider from your account. They were never designed to govern your application's cross-provider, cross-tool, cross-customer surface. The full granularity, scope, and timing analysis is in Cycles vs Provider Spending Caps ↗.

Observability platforms

Langfuse, Helicone, LangSmith, and OpenLLMetry are excellent at what they do. They trace, attribute cost, measure latency, surface anomalies, and give teams the visibility to diagnose what an agent did after it did it.

That last phrase is the constraint. Observability platforms read from provider APIs and write to dashboards. They are, by design, retrospective. The trace exists because the action already happened.

An observability tool watching a runaway agent burn $1,900 over a weekend will produce a faithful and beautifully detailed record of the disaster. It will not stop the agent at call number 50, when the damage was still $1.50. Alerts shorten the reaction window but they do not create pre-execution control — autonomous agents do not pause to wait for a human.

It is sometimes suggested that observability tools could "just add" enforcement. They could in principle. But it would require them to become a different category of system — a transactional, multi-tenant, atomic decision service in the critical path of every agent action. That is not an extension of an observability product. It is a different product at a different layer. The longer treatment is in Cycles vs LLM Proxies and Observability Tools ↗.

Framework limits

LangChain has `max_iterations` and `max_execution_time`. CrewAI has step caps. The OpenAI Agents SDK has its own loop and tool-call ceilings. These are sensible defaults for the simplest class of runaway behavior — an agent that gets stuck calling the same tool forever.

The ceiling they enforce is per orchestrator instance. Two parallel runs on two workers, each respecting `max_iterations=50`, will produce one hundred iterations against the shared budget. Five workers running the same agent during a fan-out will produce two hundred and fifty. A retry that re-enters the orchestrator after a transient failure resets the counter. A worker crash drops the counter entirely.

These limits also speak in iteration counts, not in money or risk. They do not know what an iteration costs. They do not distinguish a $0.001 tool call from a $4 model call. They do not attribute spend to a specific customer. They are local circuit breakers, not budget authority ↗.

In-process counters and DIY wrappers

The pattern is familiar: a small Redis-backed counter, one increment per call, a check before each request. It works for a sprint. Then it doesn't.

The walls show up in a predictable order — TOCTOU race conditions under concurrency, multi-provider attribution drift, retry storms ↗ that double-count or under-count, lost state on worker crashes, the slow realization that what started as a counter is becoming a distributed transactional system. The full post-mortem of that journey, including the specific failure modes and the production data that triggered each rebuild, is in We Built a Custom Agent Rate Limiter. Here's Why We Stopped. ↗.

Why these tools cannot evolve to fill the gap

It is tempting to assume the gap closes with one more release. A cap that sees more than one provider. An observability tool that adds enforcement. A framework that adds distributed counters. None of these are technically impossible. All of them are architectural reorientations that turn the tool into a different category of system.

Each of these categories was designed to govern itself.

Tool type	What it was designed to govern	What that means structurally
Provider caps	The provider's billing exposure ↗ to your account	Single-product, single-org, calendar-grained, embedded in billing
Observability	Its own ingestion pipeline and dashboards	Read-only by design, optimized for trace volume, not transactional decisions
Framework limits	The orchestrator's own loop	Per-process, per-instance, in-memory, no distributed view
In-process counters	One worker's view of one budget	Local state, fragile under concurrency, no shared truth across workers

A provider cap cannot become a cross-provider authority without becoming an external service that the provider's billing system does not control. An observability tool cannot become an enforcement layer without rewriting its data path from "ingest after the fact" to "decide before the fact" — which is a transactional decision system, not an analytics product. A framework limit cannot become a distributed authority without becoming a service the framework calls into rather than a parameter the framework sets. A homegrown counter cannot become a multi-tenant, multi-provider authority without being rebuilt as exactly that — at which point the team is no longer building a counter.

This is not a feature gap. It is a layer gap. The enforcement surface the agent actually lives on is **outside any one of the systems it uses**, and the only thing that can govern it is a separate system at that outside layer.

What the cross-cutting layer requires

If the layer has to live outside any one tool, the requirements follow from that:

- **External authority.** Lives outside any provider, tool, framework, or worker process, so it can see across all of them.
- **Atomic, distributed** reservations ↗**.** Concurrency-safe by construction. Two agents on two workers cannot both claim the same remaining budget. The race condition that breaks the in-process counter cannot exist by design.
- **Hierarchical** scope ↗**.** Tenant → workspace → app → workflow → agent → toolset, with budgets enforced at every level. The same primitive answers "how much can this customer spend?" and "how much can this single run spend?"
- **Reserve, commit, release.** Budget is held before the action runs, finalized with the actual cost after, and any unused estimate is returned. This is what makes pre-execution enforcement accurate over time — estimates can be conservative without permanently locking budget the agent never spent.
- **A** three-way decision ↗**.** ALLOW, ALLOW_WITH_CAPS, DENY — so an agent that is running low can degrade gracefully (cheaper model, smaller context, skip optional steps) instead of hard-failing.
- **Provider-, tool-, and framework-agnostic.** The same primitive applies regardless of which slice the action lives in. A reservation ↗ against the agent's budget is the same protocol call whether the spend is an OpenAI token, a Stripe charge, or an outbound email.

These properties are not arbitrary. Each one falls out of "the agent is cross-cutting, so the controls have to be too." Drop any one of them and the layer collapses back into something with a local view — a per-provider cap, a per-process counter, a per-orchestrator limit.

The layer, not the feature

Provider caps are not going to grow into cross-provider authorities. Observability tools are not going to grow into transactional enforcement layers. Framework limits are not going to grow into distributed governance services. None of those evolutions is impossible. They are just different products at a different layer, and the tools that exist today have their architectural assumptions baked into the wrong layer for this job.

Cycles sits at that outside layer. Keep the provider cap. Keep the observability tool. Keep the framework limit. But add the cross-cutting authority for the question none of those can answer: **may this agent, for this customer, on this worker, take the next action right now?**

If your agent spans N providers, M tools, K tenants, and W workers, your governance has to span the same N × M × K × W. Anything that lives inside one of those dimensions is a partial view. A partial view is not governance.

Next steps

- Runtime Authority vs Guardrails vs Observability ↗ — the *lifecycle* companion to this post: why enforcement has to happen before the action, not during or after.
- Cycles vs LLM Proxies and Observability Tools ↗ — deeper treatment of where proxies and observability fit, and where authority is the missing layer.
- Cycles vs Provider Spending Caps ↗ — the granularity, scope, and delay analysis of provider caps in detail.
- We Built a Custom Agent Rate Limiter. Here's Why We Stopped. ↗ — the production post-mortem of the in-process counter failure mode.
- Multi-Tenant AI Cost Control ↗ — per-tenant isolation ↗ and the noisy-neighbor problem.
- Idempotency, Retries, and Concurrency ↗ — the failure modes the reserve-commit primitive is designed to survive.

MORE FROM THE BLOG

Agent Delegation Chains Need Authority Attenuation, Not Trust Propagation
April 4, 2026
(https://runcycles.io/blog/agent-delegation-chains-authority-attenuation-not-trust-propagation)

Runtime Authority vs Guardrails vs Observability
March 20, 2026
(https://runcycles.io/blog/runtime-authority-vs-guardrails-vs-observability)

The State of AI Agent Incidents (2026): Failures, Costs, and What Would Have Prevented Them
April 3, 2026
(https://runcycles.io/blog/state-of-ai-agent-incidents-2026)

← Back to all posts (https://runcycles.io/blog/)