On this page >

← Blog (https://runcycles.io/blog/)

March 20, 2026 · Albert Mavashev · 7 min read

runtime-authority agents concepts guide

Copy link Share Share PDF (https://runcycles.io/pdfs/what-is-runtime-authority-for-ai-agents.pdf)

What Is Runtime Authority for AI Agents?

AI agents are moving beyond chat. They call tools. They update records. They send messages. They trigger downstream work. They retry on failure. They fan out across subtasks. They run in loops until a goal is met — or until something stops them.

The control problem is no longer just what the model says. It is what the system is allowed to do next.

In a request-response system, the blast radius of a bad output is a bad output. In an agentic system, the blast radius includes money spent, emails sent, records modified, APIs called, and downstream systems triggered.

That is what runtime authority ↗ is for.

Runtime authority governs whether an agent may consume resources, take actions, or create exposure ↗ — before the next step executes.

What runtime authority is

Runtime authority is the layer that makes pre-execution decisions over agent behavior: whether an action is allowed, under what limits, in which scope, and with what consequences if a limit is reached.

It is not a dashboard. It is not a log. It is not a counter incremented after the fact.

It is an enforcement point that sits in the execution path and can make one of three decisions before the next model call, tool invocation, or side effect happens:

- **ALLOW** — proceed normally
- **ALLOW_WITH_CAPS** — proceed, but with constraints (use a cheaper model, skip an optional step, reduce output length)
- **DENY** — stop this action; the budget or policy does not permit it

That three-way decision ↗ is what distinguishes runtime authority from binary on/off switches. It enables graceful degradation ↗ — an agent that runs out of budget does not crash. It adjusts.

The key properties of a runtime authority are:

- **Pre-execution.** The decision happens before the action, not after.
- **Enforcement.** The system can block or constrain, not just observe and report.
- **Scoped.** Decisions apply at the right level — per tenant ↗, per workflow, per agent, per run — not just globally.
- **Concurrency-safe.** Correct even when multiple agents share the same budget and act simultaneously.
- **Reconciled.** Estimated cost is reserved before execution; actual cost is committed after. The difference is released.

Why runtime authority matters for autonomous systems

In a traditional LLM call, cost is predictable: input tokens ↗ plus output tokens, one round trip. Agents break that model. A single user request can trigger chains of model calls, tool invocations, retries, and fan-out ↗ — each with its own cost and side effects.

Four things make agent costs fundamentally harder to predict and control:

Loops. An agent iterating on a task might call the model 50 times or 500. The count depends on problem difficulty and stopping criteria — neither known in advance.

Retries. When a tool call fails, most frameworks retry automatically. Each retry is a new model call with a new cost. Silent retries are especially dangerous because nothing logs them as unusual.

Fan-out. A task like "process these 200 documents" expands into 200 independent subtasks, each with its own model calls and tool invocations. Branches can themselves branch.

Tool invocations. Agents do not only consume tokens. They call APIs, send messages, and trigger external systems. A model call that costs $0.03 in tokens might trigger a tool call that costs $2.00 — or sends an email that cannot be unsent.

Without runtime authority, you discover overspend, unintended side effects, and policy violations only after they already happened.

That is observability. It is valuable. But it is not control.

The three-layer agent stack: routing, visibility, authority

Most teams building on LLMs assemble a stack that addresses three concerns. Each answers a different question at a different point in the execution lifecycle.

Layer	Question	When it acts	Examples
Routing	*Which* model handles this?	Before execution (model selection)	LiteLLM, Portkey, Manifest
Visibility	*What* happened?	After execution (logging, tracing)	Helicone, Langfuse, LangSmith
Authority	*Should* this happen at all?	Before execution (permission, limits, and policy check)	Cycles

Routing is well-understood. Visibility is well-understood.

Authority — the pre-execution enforcement decision — is the layer most teams are missing.

It is also the only layer that can **prevent** overspend and uncontrolled side effects rather than **report** them.

These three layers compose — they are not alternatives. Remove any one and a gap appears.

What runtime authority is not

Runtime authority is not observability. Observability tells you what happened. Authority decides what is allowed to happen. A dashboard that shows you Monday's $2,800 weekend spike is valuable for the post-mortem. It did not stop the agent at call number 50, when the damage was still $30. See From Observability to Enforcement ↗ for the full maturity curve.

Runtime authority is not rate limiting. Rate limiting controls velocity — how fast a system can act. Authority controls total bounded exposure ↗ — how much a system is allowed to consume in aggregate. An agent can stay perfectly within its requests-per-second limit and still burn through its entire budget over time.

Runtime authority is not billing. Billing is retrospective: what to charge, what invoice to generate. Authority is pre-execution: whether this action may proceed given the remaining budget. The two work together, but they answer different questions at different times.

Runtime authority is not orchestration. An orchestrator decides what should happen next — task sequencing, dependencies, fan-out. Authority decides whether the next thing is allowed to happen at all. One manages workflow. The other manages permission.

Runtime authority is not a soft guardrail in application code. A counter incremented after each call is a checker, not an authority ↗. Under concurrency, two agents can both read the same counter, both decide they have room, and both proceed — exceeding the budget without either one seeing the violation. A real authority makes atomic decisions: this budget is now reserved, and no concurrent actor can also claim it.

Runtime authority is not prompt-level safety. Content filters and guardrails govern what a model says. Runtime authority governs what the system around the model is allowed to do — reserve resources, invoke tools, trigger side effects. One shapes output. The other shapes execution.

The reserve/commit lifecycle

The mechanism behind runtime authority is the reserve/commit lifecycle ↗. Instead of tracking spend after the fact, budget is reserved before execution and actual cost is committed after.

1. **Reserve** — before work starts, the agent declares an estimated cost. The authority checks all applicable scopes (tenant, workflow, run) atomically and either reserves the budget or denies the request.
2. **Execute** — work proceeds only if the reservation ↗ succeeded. The reserved amount is held against the budget, visible to all concurrent actors.
3. **Commit** — after work completes, the agent reports the actual cost. If actual cost was lower than the estimate, the unused remainder is released automatically.
4. **Release** — if work is canceled before completion, the reservation is explicitly released.

Consider a document-processing agent with a $50 budget per run. Before calling the model, the agent reserves $0.15. The authority checks: the run has $50 allocated, $12.40 committed so far, $0.60 held in other active reservations — $37.00 available. The reservation succeeds. The model call completes and uses $0.09 in actual tokens. The agent commits $0.09; the remaining $0.06 is released back to the budget. If a second agent is running concurrently for the same tenant, it sees the $0.15 held — not available — and cannot double-claim it.

This lifecycle solves the problems that simple counters cannot:

- **Concurrency.** Two agents cannot both claim the same $20 of remaining budget. The reservation is atomic.
- **Retries.** A retried operation uses the same reservation, preventing double-spend.
- **Partial failure.** If work fails halfway, the uncommitted portion is released — not lost.
- **Uncertainty.** You rarely know the exact cost of a model call before it happens. Reserve/commit handles the gap between estimated and actual cost cleanly.

The lifecycle also enables **hierarchical scopes**. A single reservation checks multiple levels — organization, tenant, workspace, workflow, run — in one atomic operation. If any scope is exhausted, the reservation is denied. Per-tenant limits, per-workflow caps, and per-run budgets compose without custom enforcement logic in the application.

How Cycles approaches runtime authority

Cycles is not a dashboard for agent costs. It is a protocol for runtime permissioning over spend, risk, and actions.

Before an agent takes its next action, Cycles answers whether that action is allowed, under what constraints, and what happens if limits are reached. That decision is made by a protocol ↗ — not by a proxy, not by application code, not by a dashboard with alerts. The protocol defines the reserve/commit lifecycle, hierarchical scopes, three-way decisions, and idempotency guarantees that make runtime authority operational.

Because it is protocol-based, Cycles works across frameworks, languages, and providers. It does not care whether the agent is built with LangGraph, CrewAI, a custom loop, or a coding agent like Claude Code. It cares whether the next action is permitted.

Next steps

- Runtime Authority vs Runtime Authorization ↗ — how this term differs from identity-based authorization (AWS Bedrock AgentCore Policy, Akeyless, agent IAM)
- Cycles vs LLM Proxies and Observability Tools ↗ — where budget enforcement fits in a production LLM stack
- From Observability to Enforcement ↗ — the maturity curve from dashboards to pre-execution budget decisions
- Coding Agents Need Runtime Authority ↗ — the specific case for coding agents
- What Cycles Is Not ↗ — deeper exploration of category boundaries
- Budget Wrapper vs Runtime Authority for AI Agents ↗ — why building a prototype is easy but owning a runtime authority is not
- How Reserve/Commit Works ↗ — the protocol mechanics behind the lifecycle
- End-to-End Tutorial ↗ — set up Cycles with a working agent in under 30 minutes

MORE FROM THE BLOG

Why Local-First Agent Runtimes Need Runtime Authority
May 7, 2026
(https://runcycles.io/blog/every-local-first-agent-runtime-needs-budget-authority)

AI Agent Kill Switches Should Be Scoped
May 6, 2026
(https://runcycles.io/blog/ai-agent-kill-switches-should-be-scoped)

Python AI Agent Control: Cost, Risk, and Audit by Layer
May 6, 2026
(https://runcycles.io/blog/python-ai-agent-control-cost-risk-audit-layers)

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