crawdad.

REFERENCE ARCHITECTURE

# A Reference Architecture for Runtime Agent Security

*How to deploy inspection, enforcement, and proof at the point where agents act.*

CRAWDAD SECURITY · WHITE PAPER

SEPTEMBER 2026

## Executive summary

The security industry has reached rough consensus on *what* is needed to secure autonomous AI agents: an identity layer that scopes what each agent may reach, a runtime layer that inspects what it actually does, and a governance layer that produces an audit trail. What remains underspecified is *how to build the runtime layer* — where it sits, what it inspects, what it must never do, and how it fits alongside the identity and governance controls an organization already has.

This paper is a reference architecture for that runtime layer. It is written for security architects, platform engineers, and technical leaders who have accepted that agent identity is necessary but insufficient, and now have to decide where enforcement lives and how to deploy it without breaking the systems it protects.

The architecture rests on four principles, each a direct response to a failure mode the last eighteen months made concrete. Inspection must happen **on the wire**, at the point where an agent's action becomes a request to the outside world, because that is where the action is real and still stoppable. Enforcement must be **local and on-device**, because the tool that inspects everything an agent does must not itself become the thing that leaks it. Policy must be **legible and testable** — expressed as rules a human can read and simulate before they run — because a black-box control cannot be governed or trusted. And every decision must be written to a **tamper-evident record**, because regulation and incident response both increasingly demand provable history.

The paper works through each principle, then walks two deployment topologies — a single developer or workload, and a managed fleet across many machines or clients — and closes with an integration model for placing this layer beside agent identity, existing security tooling, and compliance obligations. Throughout, the goal is the same: enforcement at the point of action, without a new single point of failure and without moving sensitive data off the machine to secure it.


# 1. The gap this architecture fills

The three-layer model for agent security — identity, runtime, governance — is now widely shared. The difficulty is that the middle layer is the least standardized and the hardest to place. Identity has mature anchors: non-human identity management, scoped credentials, agent identity objects. Governance has mature anchors too: policy registries, audit systems, compliance frameworks. Runtime enforcement sits between them, and the architectural questions it raises are genuinely open.

Where does it intercept — at the model API, at the tool call, at the network egress, or all three? Does it run in the cloud as a service, or locally on the machine where the agent runs? What does it inspect, and just as importantly, what does it capture and where does that capture go? What can an operator change, and what must remain fixed

even against a valid administrative command? These are not questions identity or governance answer, and getting them wrong produces either a control that is trivially bypassed or one that becomes a liability of its own.

The documented incidents of 2025 and 2026 are, read closely, an argument about exactly these questions. EchoLeak succeeded against a system whose identity and permissions were never violated — the failure was the absence of enforcement on the *action*. The Replit database deletion happened because a capable agent had standing power and no enforced boundary at the point of the destructive operation. The MCP tool-poisoning class works because tool trust is assumed rather than inspected at the moment of use. Each points to the same architectural conclusion: **the enforcement point has to be as close as possible to where the agent actually acts, and it has to be structurally incapable of the failure modes it exists to prevent.**

## 2. Principle one — inspect on the wire, at the point of action

An agent's intent is not observable. Its *actions* are. Every meaningful thing an agent does — call a model, invoke a tool, read a resource, reach the network, move data — eventually becomes a request that crosses a boundary. That boundary is where the action is both real and still reversible. It is the right place to inspect.

Concretely, this means placing the enforcement point in the request path between the agent and the services it calls — the model providers, the tool endpoints, the network. In a runtime architecture, the agent's outbound traffic is routed through a local inspection point that sees each request as it is made, evaluates it, and then allows, holds, or blocks it before it reaches its destination. This is a transparent-proxy pattern applied to agent traffic: to the agent, nothing changes except the address it talks to; to the operator, every action becomes visible and governable at the moment it happens.

Inspecting on the wire has three properties that inspection anywhere else lacks. It is **complete** for the actions that matter — anything that reaches an external system passes through it, including the tool calls and egress that carry exfiltration. It is **timely** — the request can be held or blocked before the action completes, not merely logged after. And it is **agent-agnostic** — it does not depend on cooperation from the agent framework, the model, or the tool; it sees the traffic regardless of what produced it. That last property matters because the agent ecosystem is heterogeneous and changing; an enforcement point that requires every agent, model, or tool to integrate with it will never achieve coverage.

The wire is also where the newest attack surface is visible. MCP tool traffic — the calls an agent makes to its tools, and the responses those tools return — passes through the same request path. An enforcement point on the wire sees tool invocation and tool response as the concrete actions they are, rather than trusting a tool description that may have been poisoned.

## 3. Principle two — enforce locally, on the device

Here is the principle that most sharply distinguishes a runtime architecture that can be trusted from one that cannot: **to protect an agent, the enforcement point has to see everything the agent does — so it must be structurally incapable of becoming the thing that leaks it.**

The naive approach routes agent traffic through a cloud inspection service. This creates exactly the exposure the control is meant to prevent. A cloud inspector, by construction, receives the full content of every request an agent makes — the prompts, the data, the file contents, the tool arguments. It has now centralized, off the customer's machine, the most sensitive material in the environment. The control has become a breach waiting to happen, and a compliance problem in its own right.

The architecture this paper recommends inverts that. **Inspection happens locally, on the device where the agent runs.** The raw content of an agent's requests never leaves the machine as a condition of being inspected. What may leave — if the operator chooses to enable it, for fleet visibility or a security operations center — is *metadata about the decision*: the category of what was detected, the severity, the verdict, the identity of the agent. Never the prompt. Never the data. Never the file.

This is not a policy promise; it should be an architectural guarantee. The distinction matters. A promise is a sentence in a data-processing agreement. A guarantee is a property of the system — for instance, an egress path that is structurally constructed so that raw content cannot be placed on it, and an elevated-disclosure mode that, if it exists at all, requires explicit, multi-party consent to enable and is off by default. When "the content stays local" is enforced by the shape of the code rather than the goodwill of the operator, the tool that watches everything cannot quietly become the tool that exposes everything.

Local enforcement has a second benefit that has become newly relevant. The MCP research of 2026 established that agents frequently run in developer environments with developer-level operating-system privileges and no process isolation. An enforcement point that runs locally, on that same machine, is positioned exactly where those privileged actions occur — at the boundary between the agent and the operating system, the network, and the tools. It is enforcing in the right place because it is running in the right place.

## 4. Principle three — make policy legible and testable

A control that cannot be read cannot be trusted, and a control that cannot be tested cannot be governed. Much of the enterprise investment in agent security has gone into policy engines; far less attention has gone to whether the policy those engines enforce is something a human can actually read, reason about, and verify before it takes effect.

The architecture treats policy as a first-class, legible artifact. Enforcement decisions should be expressed as explicit rules over agent actions — at minimum a graduated set of dispositions such as *allow*, *ask* (hold for human decision), *deny*, and *terminate* — evaluated against every action the agent attempts. The rules should be readable: a security engineer should be able to open the policy and see, in plain terms, that reading the source tree is allowed, that reading a credentials file is denied, that a destructive system command is terminated outright. Nothing consequential should be buried in a model weight or a vendor's opaque scoring service.

Legibility enables the property that actually builds trust: **testability before the fact.** An operator should be able to pose a hypothetical action to the policy — "what happens if an agent tries to read the cloud credentials file?" — and get the decision the live policy would make, *before* any agent ever attempts it. This turns policy from an article of faith into something an organization can validate, review, and evolve with confidence. It is also the honest answer to the reasonable fear that a security control might silently block legitimate work: if you can test exactly what it will do, you can tune it before it does it.

There is a corollary about *floors*. Some protections should not be reducible even by a valid administrative action — a destructive-command block, a credential-exfiltration block. In a well-designed architecture these exist as hard floors that a routine policy change cannot lower, and that even a cryptographically valid remote command from a management console cannot override. The Replit incident is the argument for this: the failure was not a missing rule but the absence of an enforced boundary that a capable, authorized actor could not cross at the moment of the destructive operation. Floors are how an architecture protects against not just attackers but also mistakes and over-broad authority.

Finally, the direction of change should be asymmetric. Strengthening protection should be frictionless — a single action. *Reducing* protection should require a deliberate, confirmed, auditable ceremony. An operator should never be able to lower a fleet's guard by accident; only on purpose, with the consequence stated and the change recorded.

---

## 5. Principle four — write a record that holds up

The fourth principle is the one regulation is now making non-negotiable. The EU AI Act's high-risk obligations, in force from August 2026, include automatic recording of events over a system's lifetime. Incident response has the same requirement from a different direction: after something goes wrong, you have to be able to reconstruct exactly what an agent did, and prove the reconstruction was not altered.

The architecture writes every enforcement decision into a **tamper-evident record**. The mechanism should be standard and verifiable rather than proprietary and opaque: each entry cryptographically linked to the one before it, so that any change to a past record breaks the chain in a detectable way, and each entry signed so that the record's origin is provable. The important properties are two. First, the record must be **independently verifiable** — an auditor or an operator should be able to check the integrity of the entire chain themselves, offline, using standard

cryptography, without trusting the vendor. Second, the verification must be **honest about failure** — when a record has been altered, the system must report the break rather than paper over it. A log that always says "verified" is not evidence; a chain that can prove it was never touched is.

This record is what turns the runtime layer into the governance layer's source of truth. Policy defines what should happen; the tamper-evident chain proves what did.

## 6. Deployment topology one — the single workload

The simplest deployment protects a single machine: a developer running coding agents, or a server running an autonomous workload. The pattern is deliberately minimal, because adoption friction is itself a security problem — a control that is hard to deploy does not get deployed.

The enforcement point installs locally as a lightweight, hardened gateway. Routing an agent's traffic through it is a single configuration step — in practice, pointing the agent at the local inspection address, so that nothing about the agent's own setup changes except the endpoint it calls. From that moment, every action the agent takes is inspected on the wire, evaluated against the local policy, and recorded.

Two operational choices make this safe to adopt. First, the layer should start in an **observe-only posture** — inspecting and recording, showing the operator exactly what it *would* enforce, without yet blocking. This lets a team see their agents' real behavior and validate the policy against it before turning enforcement on. Second, moving from observe to enforce should be a deliberate, reversible action the operator takes when ready. The result is a control a single engineer can adopt in minutes, run alongside everything else on the machine, and arm on their own timeline — with the raw content of their work never leaving the device.

## 7. Deployment topology two — the managed fleet

The second topology is the one that matters at organizational scale, and for managed service providers securing many clients: one control point per machine, many machines under one management plane.

The shape is a **two-plane architecture**. On each device, the local enforcement point does the work described above — inspection on the wire, local policy evaluation, content staying on the machine, decisions written to that device's tamper-evident chain. Above them, a **management console** provides centralized policy, visibility, and response across the fleet — without ever pulling raw content off the individual machines. What flows up from each device is decision metadata; what flows down is policy.

This separation is what lets the architecture scale without becoming the centralized-exposure liability of principle two. The console can push a policy to every machine at once, change enforcement posture across a whole fleet, and surface the aggregate security picture — while the sensitive material of each agent's work stays local to each agent's machine. For a managed service provider, the model extends cleanly to multi-tenancy: each client's devices under one console, with the ability to seal a client's data so completely that it is invisible even to the provider — enforced as a boundary the console itself cannot cross, not as a setting the provider is trusted to honor.

The fleet plane is also where the governance story completes. Because each device writes a tamper-evident record of its own decisions, the organization — or the provider, on behalf of each client — can produce a cryptographically verifiable account of every protection decision made across the fleet. Policy pushed from the center; proof gathered from the edge.

Three properties keep the fleet plane trustworthy. Actions that *reduce* protection across many machines should carry a confirmation ceremony that states the blast radius — how many devices, across how many scopes — before anything happens. Protection floors defined centrally should be ones the console itself cannot lower below the hard limits. And the management commands the console issues should be signed and verified at each device, so that the device enforces its floors even against a command that is otherwise cryptographically valid. The console governs the fleet; it does not get to override the safety guarantees the architecture makes at each edge.

## 8. Fitting beside identity, tooling, and compliance

A runtime architecture is a layer, not a replacement. Its value depends on fitting cleanly beside what an organization already has.

**Beside identity.** This layer does not replace agent identity, and should not try to. Identity — non-human identity management, scoped credentials, agent identity objects such as those emerging in the enterprise identity ecosystem — answers what an agent *may reach*, and remains the necessary first control. The runtime layer answers the question identity structurally cannot: given what an agent is allowed to reach, is what it is doing right now actually safe? The two compose. An agent scoped to least privilege by identity, and inspected on the wire by runtime enforcement, is covered on both the *boundary* and the *action* — and the incidents of the last eighteen months are, almost without exception, failures of the second in the presence of a perfectly intact first.

**Beside existing security tooling.** The enforcement point runs alongside endpoint protection, data-loss prevention, and network controls, not instead of them. It is a hardened, local gateway that bolts on in one step and adds a capability those tools do not have — inspection and enforcement of agent *actions* on the wire — without requiring their removal or replacement. Adoption should not be rip-and-replace; it should be additive, and reversible.

**Beside compliance.** The tamper-evident record is designed to be the artifact an audit needs. As the EU AI Act, the Colorado AI Act, and frameworks such as the NIST AI Risk Management Framework and ISO/IEC 42001

increasingly require organizations to record and prove agent behavior, a runtime layer that produces an independently verifiable account of every enforcement decision is producing exactly the evidence those obligations demand — as a byproduct of doing its job, rather than as a separate compliance exercise.

## 9. Summary: the shape of a runtime layer that works

The industry agrees agents need a runtime security layer. The open question has been how to build one that is both effective and trustworthy. This architecture's answer is four properties, each a response to a real failure:

- **On the wire.** Inspect at the point where an agent's action becomes a request, because that is where the action is real, complete, and still stoppable — and where poisoned tools and exfiltration are visible.
- **On the device.** Enforce locally so the raw content never has to leave the machine to be inspected; the tool that watches everything must be structurally unable to leak it.
- **Legible and testable.** Express policy as rules a human can read and simulate before they run, with hard floors that neither a routine change nor a valid remote command can lower.
- **Provable.** Write every decision to a tamper-evident, independently verifiable record that is honest about tampering.

Deployed on a single machine, this is a control one engineer can adopt in minutes and arm on their own timeline. Deployed as a fleet, it is a two-plane architecture that pushes policy from the center and gathers proof from the edge, without ever centralizing the sensitive material it exists to protect. In both cases it sits beside agent identity, complements the tooling already in place, and produces the evidence compliance is beginning to require.

The agent will act. The only question this architecture answers — and the one identity and governance cannot — is whether the action itself is inspected, enforced, and provable at the moment it happens.

*Crawdad Security implements the reference architecture described here: a local, on-device gateway that inspects every agent action on the wire, keeps raw content on the machine by default, enforces a legible policy with hard floors, and writes a tamper-evident record of every decision — with a fleet console that pushes policy from the center and gathers proof from the edge. Crawdad publishes a public, reproducible benchmark corpus of contemporary agent attacks under an open license. Learn more at getcrawdad.dev.*

*This paper references public research and frameworks, including the OWASP Top 10 for Agentic Applications 2026 and publicly documented incidents, and describes an architecture rather than benchmarking vendors.*