



The Four Layers of Agent Immunization

A Practical Guide to Coding Agent Enforcement

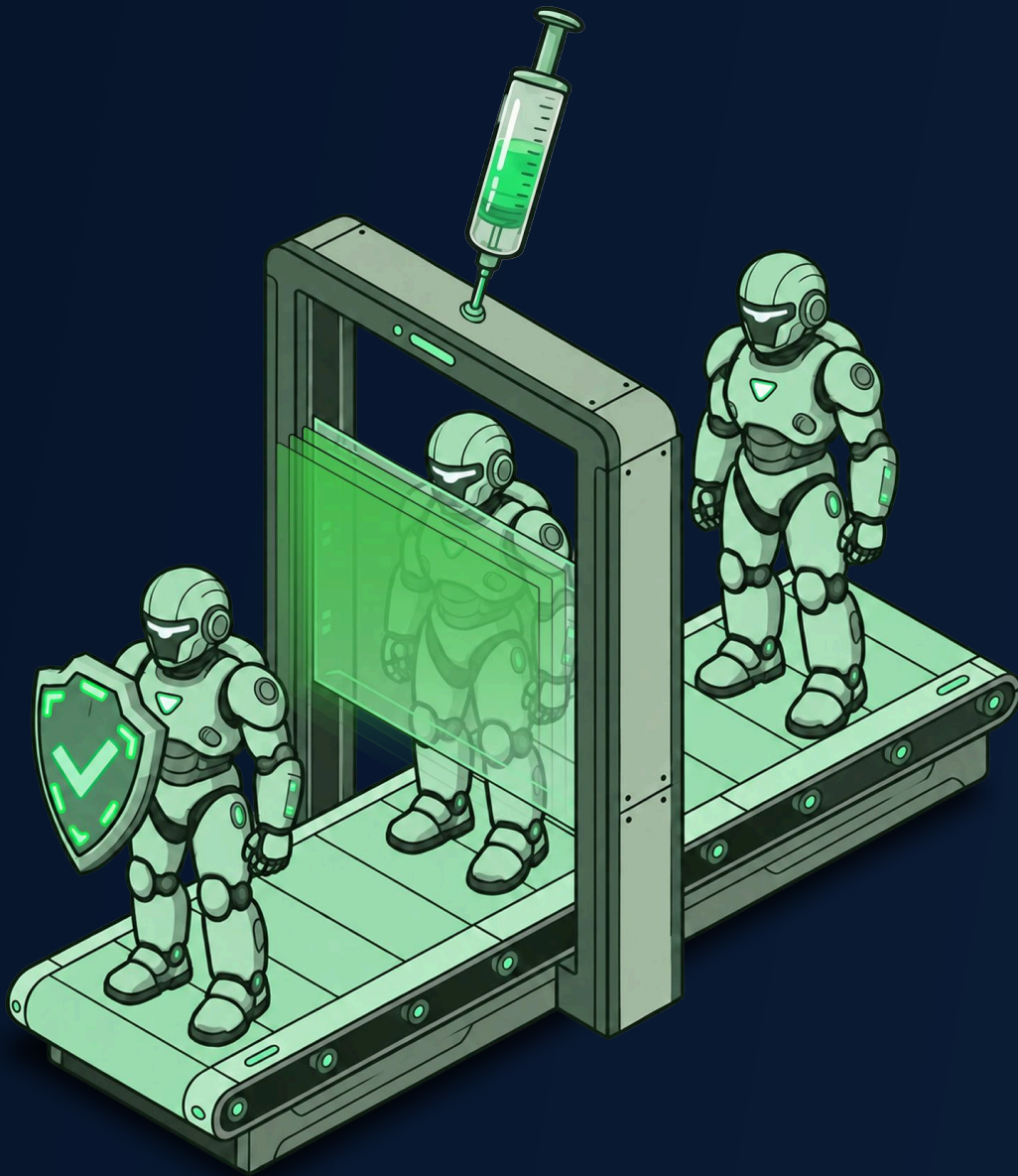


Table of Contents

You Can't Onboard an Agent 3

Layer 1: Scoped Identity 4

Layer 2: In-Agent Enforcement 5

Layer 3: Vet Before Pull 6

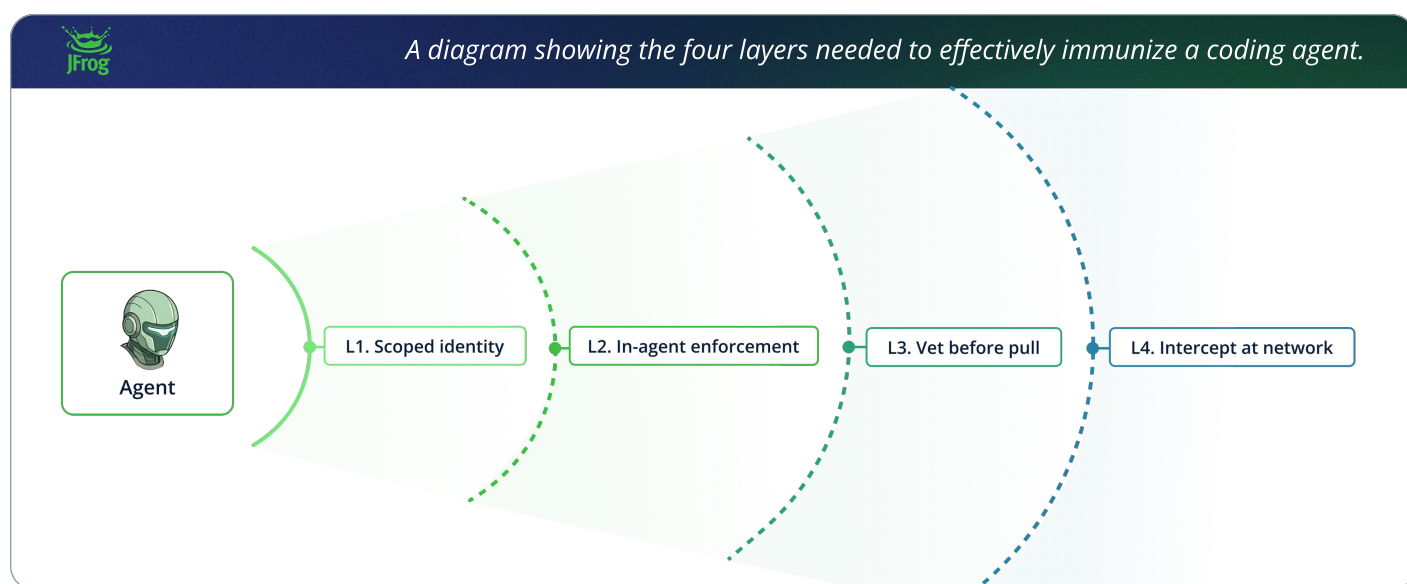
Layer 4: Intercept at Network 8

Immunize and Control Your Agents 10

You Can't Onboard an Agent

When a human developer starts a job, they're onboarded. There's a preconfigured laptop, security check and application permission checks built into the process, normally a training path and often a human buddy who can guide them in the ways of institutional knowledge. Now, ask yourself: do you do the same for every agent that is currently running in your organization?

When we use coding agents, they arrive with no onboarding and no memory of the security incident three years ago that made headlines and kept everyone paranoid about a particular package. The agent has credentials, a task, but by default, many coding agents operate permissively without personalized policies. Why? Because nobody told it otherwise in a language it understands.



Add to this that agents are relentlessly task-driven and will try any package, skill, MCP server, or action to finish a job, including things a human developer would reject without a second thought. Malware is still part of the risk, but it doesn't have to evade anyone's defenses: a compromised or malicious package gets in because the agent chose it, using borrowed credentials and optimizing for task completion with no concept of what's off limits.

The bottom line: AI coding agents need to be onboarded, but the human processes we have aren't the right fit. We need to build layers of trust with these agents. How do you do that? You need to invest in a solution JFrog calls "agent immunization," a framework designed to deliver control to the user and build trust in our agents.

Immunization means the controls live inside the supply chain the agent already works in: the same package resolution (developers pulling OSS packages from remote repositories), the same registries, the same tool calls it was going to make regardless. Not a wall built around the agent, but a structure built into the path every request already travels. That's the shift this guide is about, and it takes four cumulative layers. Why? Because a single control, no matter how strict, only protects a certain aspect of the agent's behavior, leaving it exposed to exploits by bad actors.

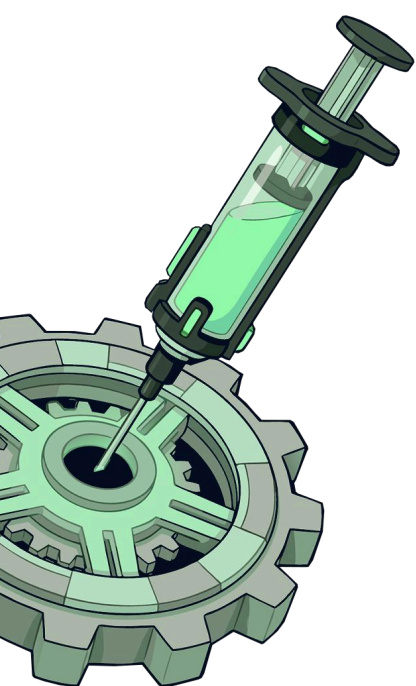
LAYER 1:

Scoped Identity

In many environments today, an agent doesn't have its own identity. It borrows one: a human's API key, a service account nobody remembers provisioning, credentials sitting in an environment variable because that was the fastest way to get the task running. From the system's point of view, a request signed with that key is often assumed to be privileged. However, most systems cannot determine was this the human explicitly approving or was this an agent. This naturally creates exposure. Not that the agent has too much access, but that whoever issued the key has no idea how much access they just handed to a script, because the request looks exactly like the person's own. A single leaked key, at that point, is total and unsupervised access to whatever that person could touch, running unattended for as long as the task takes.



The fix starts with giving the agent its own identity, so that a request made by an agent can always be told apart from one made by a person. An agent gets its own credential, one that's labeled as an agent's from the moment it's issued, not a human's borrowed for the occasion. That credential needs to carry restrictions built in at issuance: the exact scope the task requires, an expiration tied to the task itself rather than a standing grant that outlives the work, and an explicit limit on which resources it can access. Every action the agent takes under that identity also needs to carry a link back to the person or process that started it, so the chain of responsibility and auditability survives even when the agent, not a person, made the specific call.



This is immunization at the point of exposure itself. If the credential is compromised, or the agent is manipulated into acting against its own task, the damage is bounded by what that identity was ever permitted to touch, not by what the human it used to borrow from could reach. A rogue or hijacked agent running under a scoped, expiring identity of its own can do only as much damage as that identity allows, which is typically far less than a broadly scoped human credential would expose.

What identity cannot do is evaluate any specific thing the agent asks for once it's acting. Knowing precisely who is making a request, and exactly how much power they were issued, says nothing about whether the package they're requesting is vulnerable, whether the skill they're consuming was ever vetted, or whether the tool call they're making is reasonable in the moment. Identity establishes who's acting and with what authority. What happens with that authority once the agent starts acting is where the next layer picks up.

LAYER 2:

In-Agent Enforcement

Once an agent is acting under its own scoped identity, the next question is what it does with that scope in the moment. This is where the agent is actually working: mid-task, calling tools, moving data, making decisions about what to do next based on what the task requires rather than what's wise. A tool that reads a file is fine. The same tool reading a credentials file because it happened to be in scope for the task is not fine, and the difference only shows up at the moment of the call, not from anything identity alone could have flagged in advance.

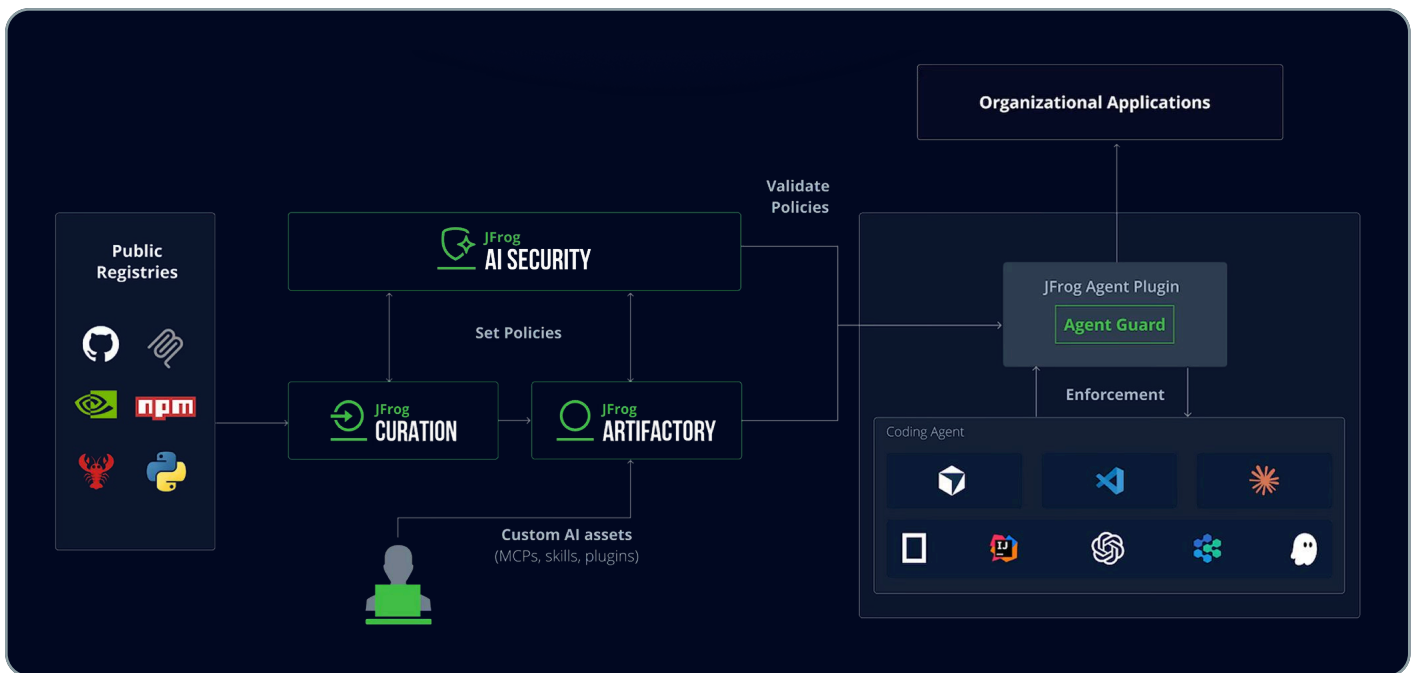


JFrog Agent Guard is built to enforce at this level, with tool-level policy enforcement for MCP servers and skill-level enforcement for libraries, not at the level of the server or library they came from. A server can be entirely permitted while specific tools within it are restricted or disabled for certain tasks, and a skill library can be entirely permitted while one specific skill inside it is blocked for a given project. Both cases require conditions a scoped identity has no mechanism to express, because identity governs who is acting and how much authority they carry, not what a specific call does with that authority in real time. This is where the immunization metaphor is most apt: the exposure is contained to the moment and the specific action, rather than spreading to whatever else the agent's credentials could reach.

The same enforcement holds even when the agent never pulled the asset from anywhere to begin with. An agent can write its own MCP server, or assemble its own skill from scratch, mid-task rather than requesting one that already exists, which means there was never an external pull for a vetting gate to catch. JFrog Agent Guard is designed to work without one.

It is designed to validate the specific asset or tool against policy at the moment the agent tries to use it, whether that asset arrived through an approved channel or was generated on the spot for the task at hand.

For example, **a self-written skill** that attempts to send data to an unapproved destination is evaluated against the same policy as a vetted skill: at the point of the invocation, not at the point of origin.



Agent Guard is also designed to bound the blast radius directly. If a tool call or skill invocation is going to touch something out of scope, the interception happens before it completes, not after the agent has already acted on borrowed trust. Two calls that are each individually reasonable can still add up to something nobody would sanction: querying a customer database is fine, similarly using a skill that drafts an external email is also fine, but the sequence of one followed immediately by the other is the kind of combination that only becomes visible at the moment it's attempted. That's sequence-based risk control, and it's both layer 2's job and the limit of its job: it governs the moment of action. Whether the AI assets behind that action were ever vetted, and whether the packages underlying the project were caught anywhere upstream, is what comes next.

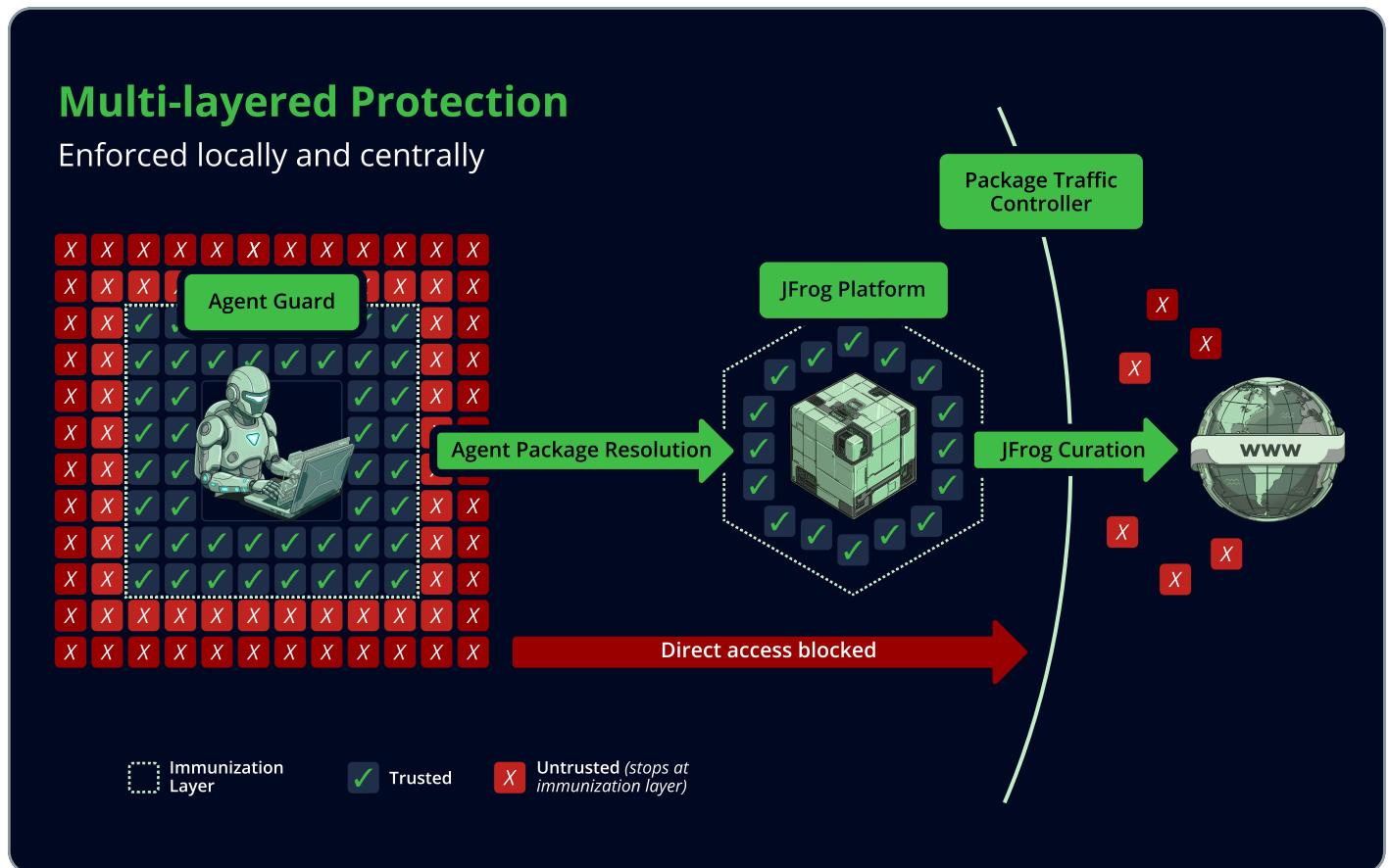
LAYER 3:

Vet Before Pull

Identity establishes who's acting, and in-agent enforcement watches what a call does in the moment. What neither one addresses is the asset the agent is allowed to reach for in the first place: models, MCP servers, skills, plugins. This is where layer 3 comes in, and it starts with one non-negotiable: every one of those requests, without exception, resolves through a system of record rather than directly against a public registry or a public model hub. This is precisely what JFrog Artifactory is meant to address.

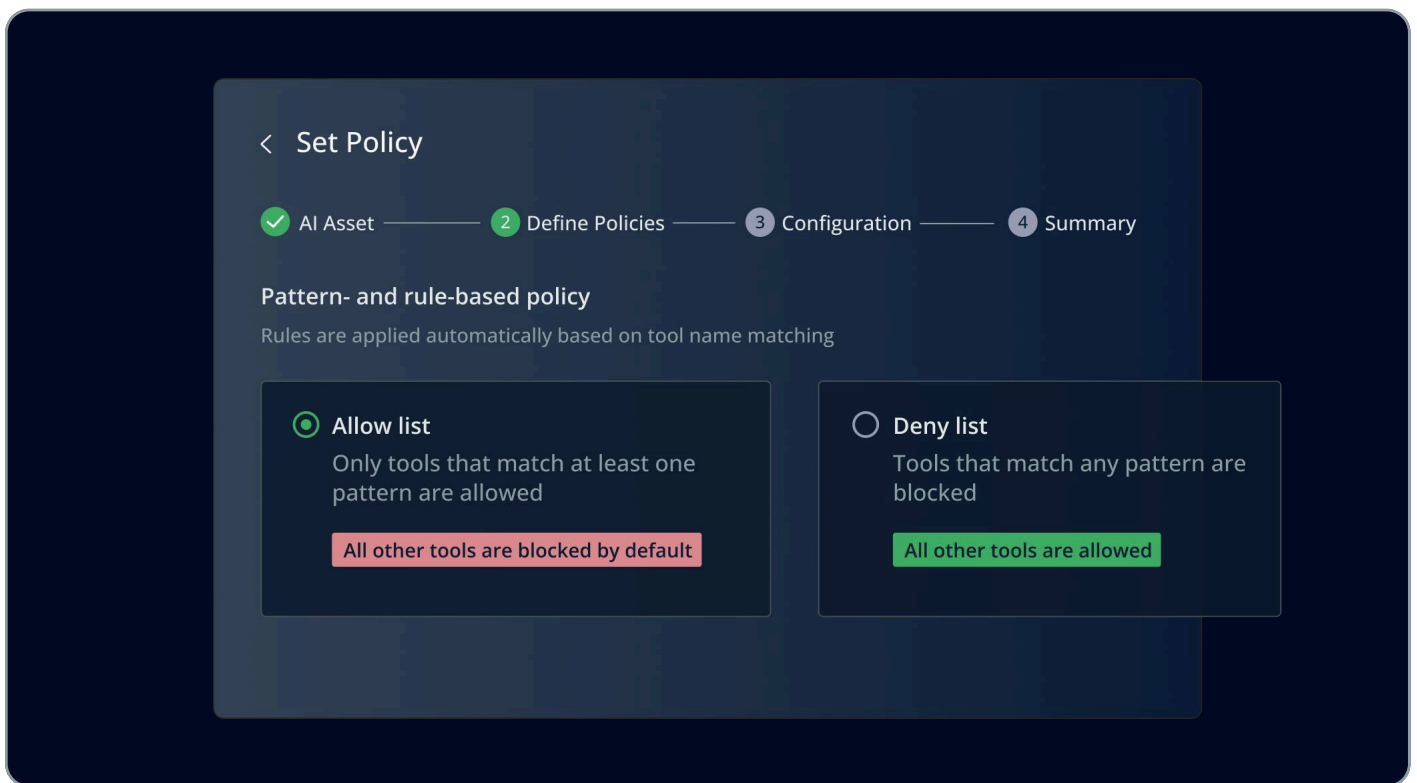
JFrog's Agent Package Resolution is what makes that non-negotiable. Whether a request is typed by a person or issued by an agent mid-task, it resolves against JFrog rather than the open internet, which is the only way policy has anything to apply to before the request has left the building.

Once resolution is forced through that single point, two policies decide what's allowed to come back. **JFrog Curation** governs the open source side: a package's vulnerability status, license terms, and known-malicious signals are evaluated before it ever reaches the agent's working directory, and only packages that satisfy your configured policy, based on currently available vulnerability, license and malicious package data, resolve.



JFrog AI Security governs the other side of the same decision: AI assets can be registered and evaluated in the AI Security before agents use them, helping keep models of unclear provenance or unmaintained MCP servers out of agent workflows. A skill or plugin gets the same treatment: its content is inspected before it's trusted, not after.

This is the layer built for a question layers 1 and 2 were never meant to answer: whether the asset itself, independent of who's calling it or what they do with it, was ever evaluated. An AI asset isn't requested the way a package is, either. It's configured into the environment, often once, often by a person who isn't thinking about it as a supply chain decision at all. The AI Security treats it as one anyway, because functionally, it is one: one supply chain, no AI exception.



This layer also solves a problem of scale that a one-off review never could. An organization running agents across dozens of teams will accumulate AI assets faster than any manual approval process can keep pace with. Every approval gate costs development velocity, and the instinct to clear each request individually, once, to keep agents moving is exactly how an ungoverned asset gets waved through. Policy applied once, at the point of resolution, makes the approval reusable, subject to continuous re-evaluation as new vulnerability and malicious package information becomes available: every agent that needs the same package or asset draws from the same decision instead of triggering a fresh one. From there, the chain keeps going: an approved model or server can carry package dependencies of its own, and the next layer picks those up the same way it would any other package request, caught by policy rather than left to resolve on their own.

LAYER 4:

Intercept at Network

By the time an agent is mid-task, its identity is scoped, its tool calls are being watched, and what it's allowed to pull has already been decided upstream, by JFrog Curation for packages and JFrog AI Security for assets. What's left is making those decisions stick against persistent workarounds. An agent that gets told no doesn't treat it as final. If a package is blocked, it will often try an adjacent name, an older version, a different registry entirely, because none of those feel like a violation to something with no concept of intent.

This is immunization in its most literal form: exposure happens inside a controlled path, not after the fact through a scan that finds what already got in. When all package traffic is routed through the **JFrog Package Traffic Controller** (PTC), requests for packages that do not meet policy are blocked before they resolve, substantially reducing the chance of a disallowed install.



The same mechanism handles a subtler failure mode: an agent given a task doesn't stop at one attempt. If a package is blocked, it will often try an adjacent name, an older version, a different registry entirely, because none of those feel like a violation to something with no concept of intent. PTC evaluates each retry as a separate request against the same policy, making persistence far less effective as a way around it.

This is where the chain closes rather than starts: the network layer knows a name and a version, not the identity behind the request, the asset context it's running in, or the tool call it's part of. Those questions were already answered three layers prior, which is exactly why the chain has to close here rather than start here: a package decision made without everything upstream of it is a decision made on a name alone. What it can see is every other package request moving through the same registries, human and agent alike, because it's built into the JFrog Platform rather than bolted on beside it. All of it resolves against the identical curation policy, with no separate rule set carved out for automation.

Immunize and Control Your Agents

It's tempting to read four layers and pick a favorite. However, the four layers address distinct failure modes. Least-privilege identity cannot replace runtime enforcement, and runtime enforcement cannot compensate for overly broad or long-lived credentials. Effective control depends on how the layers work together, with each addressing the gaps the others leave:

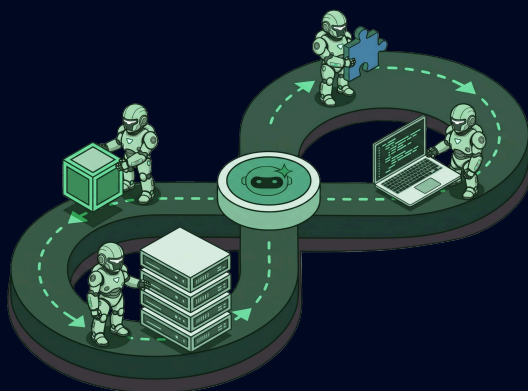
- 1 **Scoped Identity** establishes who's acting
- 2 **In-Agent Enforcement** governs what a call does with that authority in the moment
- 3 **Vetting** decides which assets are even available to call in the first place
- 4 **Network Interception** resolves the packages underneath all of it against one policy

Layered controls reduce the risk without completely eliminating it.

With regulators raising the bar on software supply chain accountability, even the possibility of a rogue or poisoned agent is no longer acceptable. Every package resolved, every asset admitted, every tool call permitted or blocked, becomes part of the same evidence trail a human developer's work already produces, so an agent's contribution to a codebase is never a special category requiring its own separate audit. One supply chain, no AI exception, means the question "what did this agent actually touch" has a single answer instead of four partial ones.

An agent cannot be onboarded the way a person is: there is no set of years or accumulated judgment to lean on. But an agent can be immunized at four distinct points, by building judgment into the path it already runs on, so that it can operate within defined, auditable guardrails.

To enforce your coding agents using the JFrog Platform,
[schedule a demo](#) or [start a free trial](#) today.



This ebook is provided for general informational and illustrative purposes. It reflects information available as of the publication date shown below and may not be updated. Examples and outcomes are illustrative; actual results may vary depending on your environment and use case. JFrog makes no representation or warranty that the content is complete, accurate, current or suitable for your particular circumstances. You are responsible for evaluating the content before relying on it. This ebook does not modify any agreement governing your use of JFrog products or services. Product use remains subject to the applicable JFrog Subscription Agreement or other written agreement between you and JFrog.