Nick Clark · Published April 25, 2026 · PDF

How to Run Untrusted Third-Party AI Agents Safely on Your Platform

If you host AI agents that other people wrote, you carry the blast radius when one of them mutates its own permissions, delegates work it should not, or propagates into a tenant it should never touch. This guide describes an architectural approach to that problem: bind each agent's permissions to a signed policy the substrate checks before any state change, and make self-modification trip a separate meta-policy gate. It is an architecture disclosed in United States Patent Application 19/230,933, not a shipping library. The home inventive step is the Execution Platform inventive step.

What You Are Building

You operate a platform that runs AI agents authored by people you do not control: marketplace plugins, customer-supplied automations, agents delegated to you by another tenant. Each one arrives wanting to read data, call tools, spawn sub-agents, and move between contexts. Your job is to let it do useful work without letting it exceed the scope you granted it, quietly rewrite that scope, or leak into a neighbor's data.

The goal of this guide is an architecture where an untrusted agent physically cannot take an action outside its granted scope, cannot expand its own scope, and every action it attempts leaves an auditable trace. Concretely: the agent's permissions travel with it

as a signed policy, the runtime evaluates that policy before it does anything consequential, and a violation produces a deterministic denial plus quarantine rather than a best-effort log entry after the fact.

This is a design you implement yourself. The mechanisms below trace to a filed patent disclosure. There is no package to install.

Why the Obvious Approaches Fall Short

The common approaches each solve part of the problem and leave a structural gap.

Container or VM isolation (gVisor, Firecracker microVMs, per-tenant sandboxes) bounds what code can touch at the operating-system level. That is real and worth having. But it governs syscalls and network egress, not semantics. Inside a legitimately granted sandbox with legitimate API access, an agent can still call a tool it was never supposed to call, delegate a task to a sub-agent it should not have spawned, or act on data from the wrong tenant. Isolation answers "can this process reach the disk," not "is this agent allowed to perform this particular action right now."

Prompt-level guardrails and safety filters inspect model inputs and outputs. They are probabilistic and, as the disclosure notes of prevailing systems, typically applied post hoc: the action is generated, then a filter decides whether to allow it. A determined or malfunctioning agent can phrase its way around a classifier, and the check runs after the reasoning that produced the unwanted action, not before the action commits.

Centralized authorization (an external policy service the agent must call) centralizes the decision but detaches it from the agent. The permission set lives in a server the agent queries; the binding between "this specific agent, with this history" and "this rule" is by reference, and enforcement depends on every code path remembering to make the call. It also assumes the orchestrator is always reachable and authoritative, which breaks in federated or edge deployments.

The gap common to all three: permissions are external to the agent and checked by convention. Nothing structurally prevents the agent from acting outside scope, and nothing stops an agent from being handed, or granting itself, a wider scope than it started with. Self-modification of privilege is the case none of these cleanly close.

The Architecture

The disclosed approach moves the permission and the check inside the execution path itself. Every claim below traces to United States Patent Application 19/230,933.

Give each agent a fixed structural schema. In the disclosure, a semantic agent is a memory-bearing object with six fields: an intent field, a context block, a memory field, a policy reference field, a mutation descriptor field, and a lineage field. Intent is what the agent means to do; context is its current environment and trust zone; memory is its own tamper-evident ledger of what has happened; the policy reference points at the rules it must obey; the mutation descriptor states the conditions under which it may transform itself; and lineage records its ancestry and delegation provenance. The agent carries everything needed to decide whether an action is permissible, so the runtime does not have to reconstruct that context from an external session.

Embed the policy as a signed contract, not a lookup. The policy reference field contains one or more cryptographically signed links to policy contracts that define the agent's permissible behaviors. The disclosure is explicit that these contracts are not advisory; they are enforced deterministically within the execution substrate, at execution time, without dependence on external orchestrators or third-party consensus. Because the policy reference travels in the agent and is signed, the enforcing runtime verifies the signature and scope locally rather than trusting a remote authority to still be reachable and correct.

Check policy before any mutation, delegation, or propagation. This is the load-bearing rule. The disclosed method evaluates the policy reference field at runtime prior to any mutation, delegation, or propagation of the agent, and the action is deterministically permitted or denied based on validation of that policy, without reliance on centralized authorization or post-execution filtering. In the disclosed middleware pipeline the agent passes through a structural validator, then a policy enforcement engine that verifies the signature, parses scope, and assesses mutation eligibility, and only then reaches a mutation queue where a validated change is applied and recorded. An action that fails the check never commits.

Gate self-modification through a separate meta-policy. The dangerous move for an untrusted agent is changing the rules that bind it: widening its own mutation scope, elevating its privilege tier, or overriding a zone constraint. The disclosure separates ordinary policy from meta-policy contracts that govern precisely these self-directed changes. Meta-policy enforcement is triggered when an agent attempts to mutate the fields that determine its own operational limits. In the disclosed example, an agent tries to alter its own mutation descriptor to allow downstream delegation without quorum validation; because that touches its own privileges, the meta-policy contract is invoked, and because its preconditions (scoped validator consensus or lineage-based authorization) are not satisfied, the substrate enforces a deterministic denial. An agent cannot bootstrap itself into a wider scope, because the attempt to rewrite scope is itself a governed act.

Make denial deterministic, and quarantine on violation. Rather than allowing the action and retroactively resolving a violation, the disclosed substrate blocks the mutation, isolates the agent in memory, and initiates a semantic quarantine; in some configurations it also rolls the agent back to its last verified state. The denied action is recorded in the agent's own memory trace, so the refusal is auditable and permanently encoded in the agent's history. Failure is a stop, not a warning.

Scope enforcement to zones, and escalate contested cases to quorum. Agents execute inside trust zones, which are logical governance domains each bound to signed policy. When a mutation is proposed, independent validators in the zone each evaluate it against the agent's memory, its mutation descriptor, and the zone's active policy; a quorum is required to approve. If quorum fails, the agent is rolled back or quarantined. A contested rejection can escalate to the meta-policy layer, which either authorizes an override or finalizes the quarantine. The disclosure notes this ensures no single node or external system can override zone governance.

Bind identity to behavior over time, not to a static key. The disclosure derives agent identity from a Dynamic Agent Hash computed over the agent's memory, mutation history, and lineage, entangled with a Dynamic Device Hash of its host, and validates the trajectory of that identity across execution cycles ("trust slope"). A discontinuity, for instance from an unauthorized mutation, blocks execution and can trigger quarantine. For a platform running untrusted agents this means a tampered or spliced agent fails validation on arrival rather than passing on a credential it merely holds.

How to Approach the Build

You are implementing an architecture, not wiring up a dependency. A workable order:

1. Define the agent object. Fix a schema carrying at least the six disclosed fields. The two that make untrusted execution tractable are the policy reference (a signed pointer to the agent's rules) and the mutation descriptor (the declared conditions under which the agent may transform). The memory field must be append-only and tamper-evident, because it doubles as your audit log.

2. Make policy a signed, verifiable object. Represent each policy contract so its integrity and issuer are checkable locally. The illustrative shape below is not runnable code; it sketches the check the enforcement engine performs, faithful to the disclosed

sequence:

```
# Illustrative only. Not a library. Faithful to the disclosed check order.
def admit(agent, zone, proposed_action):
    if not verify_signature(agent.policy_reference):      # signed contract
        return deny(agent, reason="unsigned_or_tampered_policy")
    if touches_own_limits(proposed_action, agent):        # self-modificatio
        if not meta_policy_preconditions_met(agent, zone): # quorum / lineag
            return quarantine(agent, reason="meta_policy_denied")
    if not within_scope(proposed_action, agent.policy_reference, zone):
        return quarantine(agent, reason="out_of_scope")
    return permit(proposed_action)   # only now does anything commit
```

The essential property is ordering: `admit` runs to completion before mutation, delegation, or propagation, and any negative branch stops the action rather than logging it afterward.

3. Put the check on the only path to state change. The architecture holds only if there is no way to mutate, delegate, or propagate that bypasses the enforcement engine. Route every consequential operation through one choke point. If an agent can reach a tool or spawn a sub-agent without passing the gate, the whole model is void.

4. Separate meta-policy from policy explicitly. Enumerate which fields and actions constitute changing the agent's own limits: editing the mutation descriptor, raising a privilege tier, overriding a zone constraint, delegating without required validation. Route those specifically through the meta-policy gate with stricter preconditions (validator quorum or lineage-based authorization). This is what stops privilege escalation, so it deserves its own code path, not a flag.

5. Implement zone validators and quorum. For actions that warrant it, have multiple independent validators vote against the zone's policy, and require a quorum. Record every vote in the agent's memory trace. Decide your escalation rule for

contested rejections up front.

6. Make denial and quarantine first-class. On any failed check, freeze the agent, block propagation, optionally roll back to the last verified state, and write the denial into the agent's memory. A quarantined agent should be inert and inspectable, never silently retried.

7. Log inside the agent. Because mutations, policy decisions, and denials are appended to the agent's own memory field, an auditor can reconstruct exactly what an untrusted agent attempted and why it was stopped, without correlating external logs.

What This Does Not Give You

This is an architecture disclosed in a patent filing, not a drop-in library, an SDK, or a benchmarked product. You implement it. Nothing here has been productized or performance-tested for your workload, and the disclosure states no throughput, latency, or scale numbers; do not assume any.

It is not a substitute for OS-level isolation. Policy enforcement governs which semantic actions an agent may take; it does not by itself stop a process from exhausting memory, reading raw disk, or opening a socket. Run it alongside container or microVM sandboxing, not instead of it.

Its guarantees are only as strong as your choke point and your policy definitions. If any code path can mutate, delegate, or propagate without passing the enforcement engine, the model does not hold. If your policy contracts under-specify scope, the runtime will faithfully enforce a permissive rule. The signature scheme, the quorum membership, and the meta-policy preconditions are trust roots you must secure; a compromised signing key or a captured validator quorum defeats the corresponding check.

The identity approach authenticates behavioral and entropic continuity over execution cycles; it is a different tool than PKI certificate validation and does not replace transport-layer authentication of the systems the agent talks to. And the whole approach presumes agents you can wrap in this schema. It does not retrofit governance onto an opaque third-party binary you cannot instrument.

Disclosure Scope

The architecture described in this guide is disclosed in United States Patent Application 19/230,933. This guide is educational: it explains an approach a developer can build themselves and traces each mechanism to that filing. It is not a warranty, a specification, or an offer of software, and it does not grant any license. No shipping product, benchmark, or production deployment is claimed. Implementations, security properties, and results are the responsibility of the party who builds them.

Execution Platform (/execution-platform)

All 40 steps → (/inventive-steps)

The complete runtime for governed, persistent agents.

U.S. 19/230,933 (/patents/19-230933)

PRIMARY TECHNICAL DISCLOSURE

- A Cognition-Native Execution Platform for Distributed, Stateful, and Governable Agents (/articles/a-cognition-native-execution-platform-for-distributed-stateful-and-governable-agents)

SECONDARY TECHNICAL

- Six-Field Canonical Agent Schema: Structural Definition of Autonomous Semantic Agents (/articles/execution-platform/canonical-schema)
- Semantic Nest Instantiation: Dynamic Execution Environments From Agent Density and Entropy (/articles/execution-platform/nest-instantiation)

- Trust Zone Overlay Governance: Logical Policy Domains Independent of Network Topology (/articles/execution-platform/trust-zone-overlay)
- Scoped Quorum Mutation Validation: Independent Validators With Meta-Policy Escalation (/articles/execution-platform/quorum-validation)
- Meta-Policy Override Resolution: Higher-Level Governance for Local Quorum Decisions (/articles/execution-platform/meta-policy-override)
- Semantic Router: Schema-Aware Propagation Replacing Address-Based Forwarding (/articles/execution-platform/semantic-router)
- Dynamic Agent Hash Derivation: Deterministic Identity From Memory and Mutation History (/articles/execution-platform/dah-derivation)
- Dynamic Device Hash Derivation: Substrate Identity From Device-Local Entropy (/articles/execution-platform/ddh-derivation)
- Content Anchor Hash Derivation: Perceptual Identity for Non-Executing Digital Content (/articles/execution-platform/cah-derivation)
- DAH-DDH Slope Entanglement: Binding Agent Identity to Host Device Lineage (/articles/execution-platform/dah-ddh-entanglement)
- Trust Slope Validation Across Zone Migration: Continuity Verification With Quarantine (/articles/execution-platform/zone-migration)
- Pseudonymous Propagation: Recognition by Slope Rather Than Global Identifier (/articles/execution-platform/pseudonymous-propagation)
- Alias Slope-Band Indexing: Symbolic Resolution Through Slope-Indexed Anchor Pathfinding (/articles/execution-platform/slope-band-indexing)
- Fallback Rehydration: Recovering Partial Agents Through Contextual Policy Inference (/articles/execution-platform/fallback-rehydration)
- Structural Validator With Fallback Routing: Schema Verification Before Execution (/articles/execution-platform/structural-validator)
- Execution Graph Manager: Structured Lineage of Agent Reasoning and Transformation (/articles/execution-platform/execution-graph)
- Full and Partial Agent Interoperability: Cross-Boundary Semantic Exchange Under Policy (/articles/execution-platform/agent-interoperability)
- Cross-Topology Substrate Deployment: Identical Agent Structure Across All Substrates (/articles/execution-platform/cross-topology)

APPLICATIONS · GENERAL

- Governable AI Agents: Auditable Reasoning, Policy-Constrained Orchestration, and Training-Artifact Traceability (/articles/execution-platform/ai-agent-governance)

- Multi-Cloud Agent Orchestration Without a Centralized Scheduler (/articles/execution-platform/multi-cloud-orchestration)
- Autonomous Fleet Coordination Through Self-Governing Agents (/articles/execution-platform/fleet-coordination)
- Enterprise Workflow Automation Without Orchestration Servers (/articles/execution-platform/enterprise-workflow-automation)
- Smart Contract Alternative Without Blockchain Latency: Governed Contract Execution (/articles/execution-platform/smart-contract-alternative)
- Reproducible Scientific Computing With Provenance-Bearing Governed Agents (/articles/execution-platform/scientific-computing)
- Supply Chain Autonomous Agents (/articles/execution-platform/supply-chain-agents)
- Distributed Energy Grid Management With Governed Autonomous Agents (/articles/execution-platform/energy-grid-management)
- Disaster Response Coordination Without Central Command (/articles/execution-platform/disaster-response-coordination)
- Sovereign Agent Runtimes: Running AI Agents Air-Gapped and On-Premises for Defense and Regulated Industries (/articles/execution-platform/sovereign-agent-runtimes)

APPLICATIONS · SPECIFIC

- Kubernetes Orchestrates Containers. It Does Not Know What They Are Doing. (/articles/execution-platform/kubernetes)
- Temporal Alternative for Governed Agent Execution: Durable Workflows Have No Semantic Identity (/articles/execution-platform/temporal-io)
- Apache Airflow vs. Governed Agent Execution: DAG Scheduling or Agent-Level Governance? (/articles/execution-platform/apache-airflow)
- Prefect Alternative for Governed Agent Execution: Beyond Python Task Scheduling (/articles/execution-platform/prefect)
- AWS Step Functions Alternative for Governed Agent Execution (/articles/execution-platform/aws-step-functions)
- Azure Durable Functions vs a Governed Execution Platform: Where Does Step Authority Live? (/articles/execution-platform/azure-durable-functions)
- HashiCorp Nomad vs. a Governance-Bearing Execution Platform: Where Does Workload Authority Live? (/articles/execution-platform/nomad)
- Docker Swarm Alternative for Governed Agent Execution: Beyond Opaque Containers (/articles/execution-platform/docker-swarm)
- Apache Mesos Managed Datacenter Resources. The Resources Had No Semantic Governance. (/articles/execution-platform/mesos)

- Argo Workflows Alternative for Governed Pipelines: Kubernetes-Native DAGs Without a Governance Substrate (/articles/execution-platform/argo-workflows)
- Dagster Alternative for Governed Pipelines: Software-Defined Assets Without a Governance Substrate (/articles/execution-platform/dagster)
- Luigi Alternative for Governed Agent Execution: Beyond Task-Dependency Pipelines (/articles/execution-platform/luigi)
- Camunda vs Governed Agent Execution: BPMN Orchestration Beyond the Process Engine (/articles/execution-platform/camunda)
- Zeebe vs Governed Agent Execution: Does Governance Scale With Throughput? (/articles/execution-platform/zeebe)
- AWS RoboMaker vs Governed Agent Execution at the Fleet Edge (/articles/execution-platform/aws-robomaker)
- NVIDIA Cosmos vs Governed Agent Execution: World Models Need a Runtime (/articles/execution-platform/nvidia-cosmos)
- NVIDIA DRIVE vs Governed Agent Execution: A Cross-Vehicle Substrate Alternative (/articles/execution-platform/nvidia-drive)
- NVIDIA Isaac vs a Governed Agent Execution Substrate (/articles/execution-platform/nvidia-isaac)
- NVIDIA Metropolis vs Governed Agent Execution: A Metropolis Alternative for Edge Cognition (/articles/execution-platform/nvidia-metropolis)
- LangGraph (LangChain) alternative: where does agent state, policy, and lineage actually live? (/articles/execution-platform/langgraph)
- Microsoft AutoGen vs a substrate-embedded execution platform: where does agent orchestration state live? (/articles/execution-platform/microsoft-autogen)
- CrewAI alternative: where does delegation and policy state live at runtime? (/articles/execution-platform/crewai)
- Ray (Anyscale) alternative: where does governance and identity live when agents move between nodes? (/articles/execution-platform/ray-anyscale)

Execution Platform overview → (/execution-platform)