Nick Clark · Published April 25, 2026 · PDF

How to Scope What a Delegated AI Subagent Is Allowed to Do

If you build agent systems, you have hit the problem: a parent agent spawns a subagent, and now you need that child to be strictly less privileged than its parent, provably, without trusting an external orchestrator to enforce it. This guide describes an architecture for making a delegated subagent's permitted actions travel with the agent object itself and be checked before the child can act. It describes 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 are building a delegation model where a parent agent can spawn a child (a subagent) and the child is constrained to a subset of what the parent could do, where that constraint is carried inside the child agent object and checked at runtime before the child mutates, delegates further, or propagates to another environment.

The search-intent problem is concrete. A planning agent decomposes a task and hands a piece to a worker subagent. You want the worker to be able to, say, read a specific data scope and nothing else, to be forbidden from spawning its own subagents, and to be

unable to quietly widen its own permissions. And you want that to hold even when the subagent runs on a different node, in a different process, with no live connection back to the parent or to a central authorization service.

This is a least-privilege and provenance problem for autonomous software, not a prompt-engineering problem. The guide teaches an architecture disclosed in United States Patent Application 19/230,933; you implement it yourself.

Why the Obvious Approaches Fall Short

The common approaches are real and each solves part of the problem, but each leaves a structural gap for autonomous, migrating subagents.

The first approach is an external policy service. The orchestrator asks a central authorization system "may this subagent do X?" before each action. This works while every action funnels through one enforcement point with a live connection. It degrades when subagents run across disconnected, federated, or edge environments, because the check depends on reaching the central authority at action time, and the child's authority is defined outside the child rather than inside it.

The second approach is passing a scoped credential to the child, such as a signed token narrowing a capability. Public-key infrastructure and bearer tokens describe permissions accurately and are widely used. The gap for agent delegation is that the token typically sits alongside the agent as ambient session state, and the child's actual behavior, mutation history, and further delegations are not intrinsically bound to it. A static credential attests to what was granted at issue time; it does not, by itself, carry a tamper-evident record of what the agent has since become or spawned.

The third approach is a post-hoc filter or guardrail that inspects outputs after the agent acts. This catches some violations but is reactive: the action already happened, and there is no structural guarantee the child stayed in scope during execution.

The structural gap common to all three: the child's permitted-action set, its lineage back to the parent, and the enforcement point are separable from the agent object. When any of the three lives somewhere the agent is not, a migrating or disconnected subagent can outrun its own governance.

The Architecture

The disclosed approach closes that gap by making policy scope and lineage fields of the agent object, and by enforcing them inside the substrate before the agent is allowed to act. The filing calls the unit of execution a memory-bearing semantic agent.

The agent carries its own scope. Each agent is a structured object with a fixed set of fields, including a policy reference field, a mutation descriptor field, and a lineage field (the filing lists six fields in total: intent, context, memory, policy reference, mutation descriptor, and lineage). The policy reference field holds one or more cryptographically signed links to policy contracts that define the agent's permissible behaviors. Because the scope is a field of the object, it travels with the agent across nodes rather than living in an external service.

Enforcement happens before the action, deterministically. The filing's 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 that validation, without reliance on centralized authorization or post-execution filtering. Enforcement is done by substrate-local validators, not by a remote orchestrator. This is the pivot from the reactive filter model: the check is a gate in front of the action, not an audit after it.

Delegation creates a structurally distinct, scope-inheriting child. When an agent delegates, the operation creates a new agent that is structurally distinct but inherits semantic context and policy scope from its parent, and the child remains cryptographically linked to the originating chain through the lineage field. The child

may carry its own mutation descriptor and policy reference and continue executing autonomously, while the lineage field records the directed parent-to-child link. So a subagent is not a thread the parent supervises; it is its own governed object whose scope descends from, and is traceable to, the parent.

Policy can explicitly narrow what a child may do. The filing states that policy references embedded in the agent object may declare propagation constraints such as limiting execution to a particular zone, disallowing delegation, or requiring trust slope entanglement for mutation validity. Those three examples map directly onto the search intent: you can scope a subagent to a governance boundary, forbid it from spawning further subagents, and require a validated identity-continuity condition before it is allowed to mutate.

A child cannot silently widen its own scope. The filing separates ordinary policy from meta-policy contracts, which govern whether an agent may alter its own operational limits, for example modifying its own mutation descriptor, elevating its privilege tier, or overriding zone-scoped constraints. If an agent attempts to mutate the fields that define its own limits, meta-policy enforcement is triggered. In the worked example in the filing, an agent tries to alter its own mutation descriptor to allow downstream delegation without quorum validation; because the meta-policy's preconditions are not satisfied, the substrate deterministically denies the mutation, blocks it, and quarantines the agent, recording the denied attempt in the agent's trace. Self-escalation is therefore a governed mutation subject to its own contract, not an unchecked write.

Everything is recorded in tamper-evident lineage. Mutations and delegations are appended to the agent's memory field and lineage field as a cryptographically linked record, each entry referencing a prior state, the mutation descriptor invoked, and the policy reference governing the transition. This produces a verifiable history of how a

subagent came to exist and what it was permitted to do, which is what lets a downstream node or auditor confirm a child's scope descends legitimately from its parent.

Optional: zone-scoped consensus for contested actions. For higher-assurance settings, the filing describes trust zones in which a mutation request is evaluated by multiple independent validators, approved on quorum, and escalated to a meta-policy layer when contested. This is an enforcement topology you can adopt where a single local validator is not enough.

How to Approach the Build

The following is an ordered way to implement the architecture yourself. The interface sketch below is illustrative and simplified to match the filing's structure; it is not a library and not runnable as shown.

1. **Define the agent object schema.** Give every agent explicit fields for policy reference, mutation descriptor, and lineage, alongside its task fields. Treat these as first-class structure, not metadata bolted on later, because enforcement reads them directly.

```
Agent {
  intent, context, memory,        // task-side fields
  policyRef:        [signed link to policy contract(s)],
  mutationDescriptor: {permitted transforms, delegation rules},
  lineage:          [parent hash refs, prior states, delegation records]
}
```

2. **Make policy contracts signed, referenced objects.** Do not inline raw permission booleans. The policy reference field points to cryptographically signed contracts, so a child cannot forge a broader policy without breaking the signature.

Model at least the constraints the filing names: allowed zone of execution, whether delegation is permitted, and any condition required before mutation.

3. **Put the enforcement gate before every state-changing operation.** Before an agent mutates, delegates, or propagates, validate its policy reference against the active scope and permit or deny deterministically. Enforce locally so the check does not depend on reaching a central service. The gate, not the caller, decides.
4. **Implement delegation as child construction, not permission-copying.** When a parent delegates, construct a new, structurally distinct agent that inherits the parent's semantic context and policy scope, write the parent-to-child link into the child's lineage field, and give the child its own (equal-or-narrower) policy reference. The child is independently governed thereafter.
5. **Separate meta-policy from ordinary policy.** Route any attempt by an agent to modify its own limit-defining fields (mutation descriptor, privilege tier, zone-override) through a distinct meta-policy check with its own preconditions. Default that check to deny. This is what prevents a scoped child from scoping itself back up.
6. **Append to lineage on every mutation and delegation.** Record each transition with a reference to the prior state, the descriptor invoked, and the governing policy reference, cryptographically linked, so the chain from parent to child is later verifiable by a third party.
7. **Choose your enforcement topology.** For low-stakes flows, a single substrate-local validator per node may suffice. For contested or high-assurance mutations, adopt the zone quorum and meta-policy escalation pattern the filing describes.

A useful test of your implementation: take a scoped child, disconnect it from the parent and any central service, and try to make it delegate when its policy disallows delegation. If the local gate denies the attempt and records it in lineage, the scope is traveling with the object as intended.

What This Does Not Give You

This is an architecture, not a drop-in library. There is no package to install and nothing here "just works" out of the box; you design the schema, the signed policy contracts, the local validators, and the lineage store yourself, in your own stack.

It is disclosed in a patent filing, not benchmarked or shipped as a product. The filing does not state performance numbers, and this guide claims none. Do not treat the described mechanisms as production-proven; treat them as a design you can build and evaluate.

Scope is enforced by field validation and, optionally, validator consensus. The strength of the guarantee depends on the correctness of your validators, the integrity of your signing keys, and the discipline of always gating actions behind the check. The architecture removes reliance on a central authorizer at action time; it does not remove your responsibility to implement the gate soundly.

It fits autonomous, memory-bearing, potentially migrating agents. If your subagents are short-lived calls that always route through one trusted in-process orchestrator you fully control, a conventional scoped-token or in-process capability check may be simpler and sufficient. The architecture earns its cost when children must carry their own governance across disconnected or federated environments.

The filing also describes machinery beyond this guide's scope, including entropy-derived identity, trust slope validation, and distributed semantic indexing. This guide draws only the delegation-scoping subset and does not attempt to reproduce those systems.

Disclosure Scope

The architecture described here is disclosed in United States Patent Application 19/230,933. This guide is educational: it explains an approach a developer can implement independently, and it is not a warranty, a specification, or an offer of software. Every mechanism described above is drawn from that filing's disclosure; where the filing is silent, this guide makes no claim. References to public-key infrastructure and other established technologies are for context only and are described neutrally.

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)