Nick Clark · Published April 25, 2026 · PDF

How to Validate an AI Agent's State Before It Is Allowed to Run

If you run autonomous agents across queues, edge nodes, or federated services, you eventually hit the same problem: an agent payload arrives and you have to decide whether it is safe and coherent enough to act on before you let it act. This guide describes an architectural approach to answering that question through structural validation of the agent object itself, so admission is a property of what the agent is, not what it does at runtime. The approach described here is disclosed in United States Patent Application 19/452,651 and is not a shipping library; it is the Agent Schema inventive step, which you implement yourself.

What You Are Building

You are building an admission gate for AI agents. Before an incoming agent is allowed to reason, mutate itself, delegate, or propagate to another node, something has to decide whether that agent is coherent, governed, and safe to admit. The goal here is to make that decision from the agent's own structure, ahead of any execution, so a malformed or ungoverned agent never reaches your reasoning loop in the first place.

This is a common need for anyone running agents across a message queue, serverless functions, edge devices, or multiple trust domains. In those settings you receive an agent as a payload, often from a node you do not control, sometimes after the

originating session is long gone. You cannot fall back on "the process that created it is still holding its state," because there is no such process. You need to validate the thing in front of you.

The architecture described here treats a semantic agent as a first-class, serializable data object with a canonical structure, and performs validation on that structure prior to execution. This is the Agent Schema inventive step disclosed in United States Patent Application 19/452,651. What follows is how the approach works and how you would build it. You implement it; there is no package to install.

Why the Obvious Approaches Fall Short

The usual way agent state is handled is procedural. An agent is a runtime process, session, or control loop, and its intent, memory, trust context, and governance constraints live outside the agent itself, in application logic, a workflow engine, or session-scoped state. That works while everything stays in one place. It breaks down the moment an agent has to be paused, transferred, rehydrated, or run across a stateless or federated system, because the state that gave the agent meaning was never part of the agent.

Some systems try to patch this by attaching memory or metadata to the agent payload. That helps, but without a schema that defines what a valid agent looks like, a partial or degraded payload tends to be either rejected outright or repaired with ad hoc logic that differs from node to node. The result is fragile and inconsistent behavior and poor interoperability across asynchronous environments.

There is also a validation-timing problem. If admissibility depends on runtime behavior, execution history, or shared session state, then two nodes can reach different conclusions about the same agent, and you have not really validated anything before running it; you have observed it while running. For governance that needs to be deterministic and enforced independently at each node, that is the wrong order of

operations. None of this is a knock on message queues or workflow engines, which do their jobs well. The gap is specifically that agent identity and governance were never made structural, so they cannot be checked structurally.

The Architecture

The core move is to represent the agent as a structurally self-describing data object rather than a runtime process, and to validate that object before any execution, mutation, delegation, or propagation. Eligibility to participate becomes a consequence of structural coherence, not a result of running the agent.

The schema defines six canonical semantic fields that an agent object may embed:

- **Intent**: the agent's goal, objective, or semantic direction. A declarative anchor for evaluating permissible behavior, not a set of execution steps.
- **Context**: environmental, trust, identity, and domain metadata, such as origin identifiers, trust scope, role indicators, and deployment constraints, used to interpret policy applicability locally.
- **Memory**: embedded trace outcomes from prior evaluations, mutation events, delegation records, and scaffolding resolutions, appended in a traceable manner and carried with the agent.
- **Policy reference**: a linkage to one or more governing policies constraining mutation, delegation, propagation scope, and trust thresholds. References may point to internal objects, external identifiers, or decentralized aliases, provided they are resolvable and verifiable at validation time.
- **Mutation descriptor**: the authorized transformation pathways under which the agent's intent or structure may evolve, evaluated together with the policy reference and context.
- **Lineage**: references to one or more semantic ancestors, forming a directed graph that preserves provenance and trust inheritance across generations.

Every field is embedded directly in the object and is individually addressable and machine-readable. Critically, whether the object is structurally coherent, and whether the present fields are compatible, is determined based only on information embedded within the object. No external session, registry, or execution history is consulted.

Validation proceeds in a defined order. First, confirm the object contains at least two of the six canonical fields; this minimum threshold ensures there is enough structure to interpret and govern the agent deterministically. Then evaluate the logical compatibility of the fields that are present: alignment between intent and the referenced policy, consistency between memory traces and lineage anchors, and coherence between intent and contextual constraints. An object that fails the minimum threshold, or that shows irreconcilable conflicts among its fields, is deemed structurally non-compliant and may be rejected, quarantined, or deferred.

An agent does not need all six fields to be valid. A partial agent with a subset of fields remains valid provided minimum presence and coherence thresholds are met. Where a field is absent, the schema permits resolution through **structural scaffolding**: a deterministic, schema-defined process that infers, reconstructs, or defaults a missing field from available context, policy references, and lineage anchors. Scaffolding is bounded. It does not introduce authority beyond what existing fields imply, and every inferred or defaulted value is recorded as a trace outcome in the memory field, marked as scaffolded so downstream nodes can tell inferred state from inherited state. The spec is explicit that fallback inference here is rule-bound, not probabilistic or model-driven, unless a governing policy explicitly authorizes otherwise.

Two safety defaults matter. If a memory field is absent, the agent is treated as a first-instance actor and a blank trace structure is initialized rather than fabricated. If a mutation descriptor is absent, the resolved agent is treated as immutable until mutation is explicitly authorized. That last default is what prevents a partially instantiated agent from drifting.

Because validation reads only the object, the object has to be serializable so its field boundaries, references, and validation metadata survive transport and reconstruction. The schema is defined so a receiving node can reconstruct and validate an agent from the serialized contents alone, with no prior knowledge of its history or transport path. Validation is therefore deterministic in the schema sense: identical object structures under identical policy references and context yield identical outcomes, independent of runtime, scheduler, or transport medium. The method prescribes no execution order, scheduling, or runtime control; it resolves structural admissibility only, and never initiates the agent's actions.

How to Approach the Build

The following is an ordered path. The interface sketch below is illustrative and faithful to the spec, not runnable code you can drop in.

1. **Define the canonical object.** Fix a serializable representation in which each of the six fields is independently addressable and parseable, and field boundaries survive round-tripping. An extensible, hierarchical format works. Reserve room for validation metadata alongside the fields.

```
AgentObject {
  intent?        // declarative goal
  context?       // trust scope, origin, role, environment
  memory?        // append-only trace outcomes
  policyRef?     // resolvable governing-policy references
  mutation?      // authorized transformation pathways
  lineage?       // ancestor references (directed graph)
}
```

2. **Implement the presence check.** Reject or defer any object with fewer than two canonical fields. This is your cheapest gate; run it first.

3. **Implement the coherence rules.** Encode the compatibility checks as declarative rules: intent-to-policy alignment, memory-to-lineage consistency, mutation-within-policy. Treat an irreconcilable conflict as non-compliance, not as something to silently paper over.
4. **Resolve policy references.** At validation time, resolve each policy reference and confirm it is verifiable. Discrepancies between declared policy and memory-recorded behavior should fail validation. Decide up front how unresolvable references are handled.
5. **Add scaffolding for partial agents, carefully.** For each absent field, define the deterministic default or inference: intent from context or lineage, memory as a fresh blank trace, immutability when no mutation descriptor is present. Record every resolution as a scaffolded trace outcome. If a field cannot be resolved deterministically, do not guess; reject, quarantine, or defer.
6. **Make outcomes deterministic and node-local.** Ensure two independent nodes validating the same serialized object reach the same verdict. Do not let validation depend on local session state or execution history.
7. **Write outcomes back into memory.** Record validation decisions, scaffolding, and (later) mutation and delegation events as trace outcomes inside the agent's own memory field, so the record travels with the agent and survives serialization.
8. **Gate execution on the verdict.** Only admit an agent to your reasoning, mutation, or delegation path after it passes. Keep validation strictly ahead of, and separate from, execution.

A useful design test: if you deleted your orchestration layer, could a bare node still decide whether to admit a given serialized agent? If yes, you have put the state in the right place.

What This Does Not Give You

This is an architecture, not a drop-in library, and not a benchmarked or productized system. Nothing here has been measured for throughput or latency, and this guide makes no such claims; you build the validator and own its behavior and performance.

Structural validation answers admissibility, not correctness of outcomes. It confirms an agent is coherent, governed, and traceable enough to admit. It does not verify that the agent's intent is a good idea, that a referenced policy encodes the rules you meant, or that the agent will produce a useful result. The schema itself is explicit that scaffolding resolves structural completeness only and does not initiate, schedule, or perform any semantic action.

The model leans on the policy references being resolvable and verifiable at validation time; if your policy resolution is weak, your governance is weak, regardless of how clean the structure is. Cryptographic binding of field contents and lineage is described as optional and does not change the validation model, so tamper-evidence is something you add deliberately, not something you get for free. And if your agents genuinely live and die inside a single process with no transfer, pausing, or federation, the portable-object framing may be more machinery than your case needs; the payoff shows up specifically in stateless, asynchronous, or federated environments.

Disclosure Scope

The architectural approach described in this guide, including the canonical six-field semantic agent object, structural validation prior to execution, partial-agent support, and deterministic structural scaffolding, is disclosed in United States Patent Application 19/452,651. This guide is educational. It explains how the disclosed approach works and how a developer might build an implementation of it. It is not a

warranty, a specification of any commercial product, or an offer of software, and it does not grant any license. Any implementation you build from these ideas is your own, and you are responsible for its behavior, security, and compliance.

Agent Schema (/agent-schema)

All 40 steps → (/inventive-steps)

Define what an autonomous agent is — structurally.

U.S. 19/452,651 (/patents/19-452651)

PRIMARY TECHNICAL DISCLOSURE

- Cognition-Compatible Semantic Agent Objects and Structural Validation (/articles/cognition-compatible-semantic-agent-objects-and-structural-validation)

SECONDARY TECHNICAL

- Partial Agent Structural Validity: Fewer Fields, Still Deterministic (/articles/agent-schema/partial-validity)
- Minimum Two-Field Validation Threshold: The Floor of Semantic Structure (/articles/agent-schema/two-field-threshold)
- Field Interaction Rules: Deterministic Constraints Between Canonical Fields (/articles/agent-schema/field-interaction-rules)
- Field-Based Role Typing: Agent Roles Derived From Structural Composition (/articles/agent-schema/role-typing)
- Semantic Templates: Predefined Field Arrangements as Agent Class Contracts (/articles/agent-schema/semantic-templates)
- Structural Scaffolding Logic: Resolving Missing Fields Through Inference or Defaulting (/articles/agent-schema/scaffolding-logic)
- Field-Aware Default Resolution: Deterministic Behavior When Fields Are Absent (/articles/agent-schema/default-resolution)
- Traceable Semantic Lineage Graph: Mutation History Embedded in Agent Objects (/articles/agent-schema/lineage-graph)
- Serialization With Statelessness Compatibility: Reconstruction Without External Session State (/articles/agent-schema/stateless-serialization)

- Schema Governance Through Versioned Policies: Cross-Version Structural Interoperability (/articles/agent-schema/versioned-policies)

APPLICATIONS · GENERAL

- Edge and IoT Agents That Survive Disconnection: Stateless Rehydration and Partial-Agent Operation Without a Persistent Runtime (/articles/agent-schema/edge-iot-partial-agents)
- Proving AI Decision Provenance to Auditors and Regulators with Schema-Embedded Accountability (/articles/agent-schema/schema-embedded-ai-governance)
- Enterprise AI Agent Interoperability: A Canonical Schema for Multi-Framework Agent Governance (/articles/agent-schema/enterprise-interoperability)
- Multi-Vendor Robot Standardization and Interoperability with a Canonical Agent Schema (/articles/agent-schema/robotic-standardization)
- Multi-Vendor AI Agent Interoperability: A Canonical Agent Schema for Cross-Framework Coordination (/articles/agent-schema/multi-vendor-ai-agents)
- Digital Twin Standardization Through Canonical Fields (/articles/agent-schema/digital-twin-standardization)
- Portable Healthcare AI Agents: Carrying Governance and Clinical Lineage Across EHR Platforms (/articles/agent-schema/healthcare-agent-portability)
- Coalition Defense AI: Cross-National Agent Interoperability Without System Unification or Sovereignty Concessions (/articles/agent-schema/defense-coalition-interop)
- Automating Insurance Claims Across Insurer, Adjuster, and Repair-Shop Systems with a Canonical Agent Schema (/articles/agent-schema/insurance-claims-agents)
- Legacy System Integration for AI Agents Without Rewriting the Mainframe (/articles/agent-schema/legacy-system-integration)

APPLICATIONS · SPECIFIC

- LangChain vs Governed Agent Execution: The Canonical Schema LangChain Does Not Define (/articles/agent-schema/langchain)
- AutoGen Alternative for Governed Agents: Structural Agent Definition Beyond Conversation (/articles/agent-schema/autogen)
- CrewAI Alternative for Governed Agents: Role Teams vs. the Agent Schema (/articles/agent-schema/crewai)
- Semantic Kernel vs Governed Agent Execution: The Agent It Builds Has No Schema (/articles/agent-schema/semantic-kernel)
- OpenAI Assistants API vs Governed Agent Execution: Tooling Without an Agent Schema (/articles/agent-schema/openai-assistants)

- Google Vertex AI Agents vs a Self-Describing Agent Object: Managed Runtime Without a Canonical Schema (/articles/agent-schema/google-vertex-agents)
- Amazon Bedrock Agents Orchestrate Foundation Models. The Agents Have No Canonical Schema. (/articles/agent-schema/amazon-bedrock-agents)
- Haystack Alternative for Governed Agents: Composable Pipelines Beyond the Agent Schema (/articles/agent-schema/haystack)
- LlamaIndex vs Governed Agent Objects: The Data Framework That Has No Agent Schema (/articles/agent-schema/llamaindex)
- Dify Alternative for Governed Agents: Visual Builder, No Agent Schema (/articles/agent-schema/dify)
- AutoGen and CrewAI Alternative: Governed Multi-Agent Execution with a Self-Describing Agent Schema (/articles/agent-schema/autogen-crewai)
- LangChain and LangGraph Alternative: Governed Agents Beyond Orchestration (/articles/agent-schema/langchain-langgraph)
- LlamaIndex Agents vs Governed Agent Objects: Structural Validation Beyond the Runtime (/articles/agent-schema/llamaindex-agents)
- ROS 2 vs a Portable, Structurally Validated Agent Object (/articles/agent-schema/ros2-robotics)
- Cursor vs Governed Agent Execution: A Structural Comparison (/articles/agent-schema/cursor-coding-agent)
- Replit Agent vs a Governed Agent Schema (/articles/agent-schema/replit-agent)
- MCP vs a Governed Agent Object: The Agent Layer Model Context Protocol Does Not Define (/articles/agent-schema/anthropic-mcp)
- Google A2A vs a Governed Agent Object: What the Agent Card Leaves Out (/articles/agent-schema/google-a2a)
- AGNTCY Internet of Agents vs the Canonical Agent Object at Its Center (/articles/agent-schema/cisco-langchain-agntcy)
- Letta (formerly MemGPT) vs a portable, self-validating agent object: the memory-portability axis (/articles/agent-schema/letta-memgpt)
- W3C Decentralized Identifiers and Verifiable Credentials vs a Governed Agent Object: Identity for Subjects Versus Portable Behavior (/articles/agent-schema/w3c-did-vc)
- IBM Agent Communication Protocol (ACP / BeeAI) vs a portable agent object: the transport-versus-state axis (/articles/agent-schema/ibm-acp)

Agent Schema overview → (/agent-schema)