Nick Clark · Published April 25, 2026 · PDF

How to Make an AI Agent Self-Describing and Inspectable

If you have built AI agents, you know the frustration: pause one, move it to another node, or hand it to a teammate, and its intent, memory, and permissions live somewhere else entirely. This guide teaches an architecture where the agent describes and validates itself from its own fields, so any receiving node can inspect it without shared state. It describes an approach disclosed in United States Patent Application 19/452,651 (not a shipping library), built on the Agent Schema inventive step. You implement it yourself.

What You Are Building

You are building an AI agent that carries everything a receiving system needs to understand it inside the agent itself. Not a runtime process with state scattered across a session store, a message queue, and application logic, but a single serializable data object that can be read, validated, and governed from its own contents.

Concretely, "self-describing" means any node can look at the object and know the agent's goal, the environment it assumes, what it remembers, which policies bind it, how it is allowed to change, and where it came from, without querying an external system. "Inspectable" means that same node can decide, before running anything,

whether the object is well-formed and safe to act on. This is the search-intent problem: developers want an agent they can pause, serialize, ship across a stateless or federated boundary, and have the far side reason about correctly with no shared session.

If you are building multi-agent systems, edge or federated deployments, or anything where agents outlive a single process, this is the shape you are reaching for. The architecture below comes from United States Patent Application 19/452,651, and its home inventive step is the Agent Schema inventive step.

Why the Obvious Approaches Fall Short

The common approach is to treat an agent as a runtime construct: a process, session, or control loop running over external stores and an orchestration framework. Agent behavior is procedural execution, and the things that give the agent meaning (its intent, memory, trust context, and governance constraints) live outside the agent, in application logic, workflow engines, or session-scoped state. This works, and for a single long-lived process it works well.

The gap appears at boundaries. Because identity and behavior are coupled to a specific execution environment, semantic continuity is hard to preserve when an agent is paused, transferred, rehydrated, or run across stateless or federated systems. The far side does not hold the session that gave the agent meaning. A frequent patch is to attach memory or metadata to the agent's payload, but partial or degraded representations then tend to be invalid or need ad hoc repair logic, which produces fragility and inconsistent behavior across asynchronous environments. And integrity, auditability, and trust verification usually depend on external orchestration and centralized coordination, which does not scale cleanly across decentralized systems.

None of these are broken tools. Message queues, workflow engines, and session stores do exactly what they are designed to do. The structural gap is that none of them makes the *agent itself* the unit that a stranger node can inspect and trust. That is the specific

problem the architecture targets.

The Architecture

The core move is to represent semantic agency as a first-class data object rather than a transient runtime process. The agent is a structured, serializable object that embeds its own goal expression, trust context, memory, policy references, mutation rules, and lineage. Because that information travels inside the object, a receiving node can validate, interpret, and govern the agent based solely on its internal composition, independent of any particular execution process.

The six canonical fields. The schema specifies six canonical semantic fields:

- **Intent** encodes the agent's goal or purpose. It anchors semantic identity and is the reference point for evaluating permissible behavior and mutation eligibility. It states a desired outcome, not execution steps.
- **Context** records environmental, trust, identity, and domain metadata: origin identifiers, trust scope, role classifications, environmental parameters, deployment constraints. It lets a node interpret the agent relative to local conditions without shared session state.
- **Memory** retains trace outcomes: prior evaluations, mutation events, delegation records, scaffolding resolutions, validation results. Unlike an external log, it is embedded in the object and appended in a traceable manner, so history travels with the agent.
- **Policy reference** identifies the governing policies that constrain behavior, mutation pathways, delegation authority, scope, and trust thresholds. References may resolve to internal policy objects, external identifiers, or decentralized aliases, provided they are resolvable and verifiable at validation time.
- **Mutation descriptor** defines the authorized pathways by which the agent may change: the conditions, triggers, and constraints under which its intent or structure may evolve. It works together with the policy reference and context.

- **Lineage** references one or more semantic ancestors, forming a directed graph of inheritance and evolution that preserves provenance across generations.

Validation happens before execution. This is the load-bearing property. A node interacting with the object determines (1) whether it is structurally coherent based on the presence of canonical fields, and (2) whether the fields present are structurally compatible under rules about which fields are permitted to coexist. Both determinations are made using only information embedded in the object. Validation is performed prior to any execution, mutation, delegation, or propagation, so eligibility to participate is a consequence of structural coherence, not of runtime behavior.

The minimum threshold in the disclosure is that a valid object must contain at least two of the six canonical fields. Above that threshold, validation checks logical compatibility: alignment between intent and policy, consistency between memory traces and mutation descriptors, continuity of lineage references. Objects that fail the threshold or show irreconcilable field conflicts are deemed non-compliant and may be rejected, quarantined, or deferred. Validation outcomes are described as deterministic and reproducible across nodes, which is what lets enforcement be decentralized rather than routed through a central validator.

Partial agents are first-class. An object with fewer than all six fields is not invalid by that fact alone. The disclosure treats subsets as valid roles: an intent plus context plus policy expresses a governed objective; a memory plus lineage object works as a reflective or audit-oriented agent; a context plus mutation plus lineage object participates in controlled transformation under inherited constraints. Where fields are missing, structural scaffolding can infer or default them under schema rules, but only within tight bounds. If an intent field is absent, purpose may be resolved from context, lineage, or policy defaults. If memory is absent, the agent is treated as a first-instance actor and a blank trace is initialized and marked as scaffolded (it does not fabricate history). Critically, if the mutation descriptor is absent, the resolved agent is treated as

immutable until mutation is explicitly authorized. Every inferred or defaulted field is recorded in memory as a trace outcome, so a downstream reader can always tell scaffolded state from inherited state.

Roles emerge from structure. Because the fields present define what an agent can do, roles are read directly off the object rather than assigned by an external registry. Intent plus memory plus mutation reads as a mutator; context plus policy plus memory reads as an observer or poller; context plus policy plus lineage reads as a delegate. The taxonomy is not fixed; other coherent combinations yield other roles.

Serialization and statelessness. Each object serializes to a structured representation where the six fields are individually addressable and independently parseable, preserving field boundaries and validation metadata. A receiving node reconstructs and validates the agent from the serialized contents alone, with no prior knowledge of its execution history or transport path. Lineage references let systems reconstruct ancestry graphs after the fact without a central registry, and the disclosure notes trace outcomes may optionally be cryptographically bound to field contents for tamper-evidence, without that binding being required by the validation model.

How to Approach the Build

You are implementing this yourself. The steps below are an ordering, not a package.

1. Define the object, not a process. Model your agent as a serializable data object with slots for the six canonical fields. Use an extensible, hierarchical format so each field is independently addressable. The following is an illustrative sketch, faithful to the field roles above, not a runnable API:

```
Agent {
  intent:    { goal, direction }            // declarative, no steps
  context:   { origin, trustScope, role }   // environmental metadata
  memory:    [ traceOutcome, ... ]           // appended history
  policy:    { ref | aliases }               // resolvable at validate time
  mutation:  { pathways, triggers, bounds }  // absent => immutable
  lineage:   [ ancestorRef, ... ]            // directed graph
}
```

2. Write the structural validator first. Before any execution path exists, build the function that takes a serialized object and returns coherent / non-compliant using only the object's own fields. Enforce the minimum threshold (at least two canonical fields), then the compatibility checks (intent aligns with policy, memory is consistent with mutation descriptors, lineage references are continuous). Keep it deterministic: the same object must validate the same way on every node.

3. Make policy references resolvable and verifiable. Decide how a policy reference resolves (internal object, external identifier, or alias) and ensure resolution and verification happen at validation time, before the agent acts. Design for the missing-policy case: apply default governance scoped by context, and record that default application as a trace outcome.

4. Define your scaffolding rules explicitly. For each field that can be absent, specify the deterministic resolution: how intent is inferred, that memory initializes blank and marked scaffolded, that absent mutation means immutable. Write every resolution into memory as a trace outcome. Do not let scaffolding grant authority the existing fields do not already imply.

5. Enforce mutation through field interaction, not code paths. When an agent evolves, produce a derived object whose lineage references the origin's lineage (never overwriting it), record the mutation in memory on both origin and derivative, and reject

any proposed change outside the mutation descriptor. Semantic evolution should be a validated field-level transformation, not an imperative side effect.

6. Serialize for strangers. Ensure your serialized form lets a node with zero prior context parse each field, run the validator, apply scaffolding if permitted, and reconstruct lineage, all from the bytes it received. Test this by validating an agent on a node that never saw it created.

7. Add templates and contracts as you standardize. Once you have recurring agent shapes, define semantic templates (required and optional fields, coherence rules, thresholds) and contractual structures (what fallback is permitted, what must be recorded before lineage extension). These are validation schemas, not workflows.

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 write the object model, the validator, the scaffolding rules, the serialization format, and the policy resolution yourself, and the quality of your validator and policy design determines the guarantees you actually get.

The approach is disclosed in a patent filing. It is not a benchmarked or production-proven product, and this guide reports no performance numbers because the disclosure states none. The schema is deliberately execution-agnostic: it validates structural completeness and admissibility, and it explicitly does not initiate, schedule, or perform execution of semantic actions. It says nothing about what runs your agent, how you host models, or how you schedule work; you supply all of that.

It also is not magic for genuinely broken input. Structural scaffolding does not guarantee resolution. Objects with too few fields, irreconcilable policy conflicts, or invalid context are non-compliant, and the architecture's answer is to reject, quarantine, or defer them, not to guess. Fallback inference is rule-bound and does not

use probabilistic or learned inference unless a governing policy explicitly authorizes it. If your problem is a single long-lived process on one machine with no boundary crossings, the self-describing object model may be more structure than you need. Its value shows up precisely when agents cross stateless, asynchronous, or federated boundaries.

Disclosure Scope

The architecture described in this guide is disclosed in United States Patent Application 19/452,651. This guide is educational: it explains an approach to designing self-describing, inspectable AI agents so that a skilled developer can implement it independently. It is not a warranty, a specification of any product, or an offer of software, and it does not grant any license. Every mechanism described above traces to the filed disclosure; where the disclosure is silent, this guide makes no claim.

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 Stateless 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)