Nick Clark · Published July 2, 2026 · PDF

How to Make an AI Agent Portable Across Frameworks

If you build an agent on one framework, its intent, memory, trust context, and governance rules end up scattered across that framework's session state and orchestration layer, so the agent cannot be paused on one runtime and resumed on another. This guide teaches an architecture that makes the agent itself the portable unit: a self-describing, structurally validatable object that survives serialization and rehydration across runtimes. The approach described here is disclosed in United States Patent Application 19/452,651 (it is not a shipping library), and its home inventive step is the Agent Schema inventive step.

What You Are Building

You want an AI agent you can move. Start it under one framework, pause it, hand it to a different runtime (an edge device, a serverless function, a peer node, a different orchestration stack), and have it resume with its goal, its memory, its governing rules, and its history fully intact. No shared database the second runtime has to reach back into. No session the second runtime has to reconstruct from logs.

The people who hit this are teams running agents across heterogeneous infrastructure: multi-framework stacks, federated systems, asynchronous message-passing pipelines, or anything where the runtime that finishes a task is not the runtime that started it.

What you are building is a way to make the agent a portable object in its own right, so that portability is a property of the data, not a feature you re-implement per framework.

This guide describes the architecture. You implement it. It is not a package you install.

Why the Obvious Approaches Fall Short

The usual way to build an agent is as a runtime process: a control loop or session that reads and writes external stores, message queues, and orchestration state. The agent's semantic intent, its memory, its trust context, and its governance constraints live outside the agent, in application logic or session-scoped state. This works fine as long as the agent stays put.

The moment you try to move it, the coupling shows. Because identity and behavior are tied to a specific execution environment, you cannot cleanly preserve continuity when the agent is paused, transferred, or rehydrated somewhere stateless or federated. The second runtime does not have the first runtime's session.

The common patch is to attach memory or metadata to the agent's payload so it "carries" some state. But a payload that is partially populated, or degraded in transit, tends to be treated as invalid, so you write ad hoc repair logic to patch it up. That repair logic is per-framework and inconsistent, which is exactly the fragility you were trying to escape. And auditability, integrity, and trust verification still lean on external orchestration and centralized coordination, which does not travel with the agent and does not scale across decentralized systems.

The structural gap: none of these approaches make the agent self-describing. Every receiving runtime needs outside context to know what the agent is, what it may do, and whether it is even valid. Portability fails because the agent cannot answer those questions from its own contents.

The Architecture

The disclosed approach inverts the model. Instead of an agent being a runtime process with state bolted on, the agent is a **canonical, serializable data object** that carries everything a receiving node needs to validate and interpret it, using only information embedded within the object itself. The spec calls this a cognition-compatible semantic agent object.

Six canonical fields. An agent object is composed of up to six canonical semantic fields, each individually addressable and machine-readable:

- **Intent**: the agent's goal or semantic direction. It is declarative: it states a desired outcome, not execution steps.
- **Context**: environmental, trust, identity, and domain metadata (origin identifiers, trust scope, role, deployment constraints) used to interpret policy and mutation eligibility relative to local conditions.
- **Memory**: trace outcomes embedded in the object: prior evaluations, mutation events, delegation records, scaffolding resolutions, and validation results, appended so that history travels with the agent.
- **Policy reference**: a link to one or more governing policies that constrain permissible behavior, mutation, delegation, and trust thresholds. The reference may resolve to an internal object, an external identifier, or a decentralized aliasing mechanism, as long as it is resolvable and verifiable at validation time.
- **Mutation descriptor**: the authorized transformation pathways, meaning the conditions and bounds under which the agent's intent or structure may evolve.
- **Lineage**: references to one or more semantic ancestors, forming a directed graph of provenance.

Validation is structural, and it happens first. A receiving node determines whether the object is structurally coherent (are the canonical fields present) and whether the present fields are structurally compatible (are they permitted to coexist

under the schema's rules). Per the spec, both determinations are made based only on information embedded within the object. This validation is performed prior to any semantic execution, mutation, or propagation. Admissibility is a consequence of structure, not of runtime behavior or execution history. The spec states these outcomes are deterministic and reproducible: identical structures under identical policy references and context yield identical validation results, independent of runtime, transport, or execution order.

Partial agents are first-class. An object does not need all six fields to be valid. The spec sets a minimum threshold of at least two canonical fields, after which validation checks the logical compatibility of what is present (for example, consistency between the policy reference and the mutation descriptor, or between memory traces and lineage anchors). A partial object missing a field is resolved through **structural scaffolding**: a deterministic, schema-defined, policy-bound resolution that infers or defaults the missing field from available context, policy, and lineage. The spec is specific and conservative about this: if intent is absent, it may be resolved from context roles, lineage objectives, or policy defaults; if memory is absent, a blank trace structure is initialized and marked as scaffolded (it does not fabricate history); if lineage is absent, an origin reference is assigned; and critically, **if the mutation descriptor is absent, the agent is treated as immutable** until mutation is explicitly authorized. Every scaffolding decision is recorded in the memory field as a trace outcome. Objects that cannot be resolved are rejected, quarantined, or deferred, not silently guessed at.

Serialization is what makes it portable. The object is serialized as a structured representation in which the six fields stay individually addressable and independently parseable, preserving field boundaries, reference relationships, and validation metadata. A receiving node reconstructs and validates the agent without any prior knowledge of its execution history, its instantiation environment, or its transport path. That is the mechanism behind cross-framework portability: continuity is carried by embedded trace outcomes and lineage references, not by a session binding on any one runtime. The spec notes this supports cloud, edge, federated, and intermittently

connected environments, and that integrity can optionally be reinforced by cryptographically binding field contents or lineage references, without changing the validation model.

Evolution and lineage stay auditable across the move. When the agent changes, it does so through a policy-governed mutation: the mutation descriptor is evaluated against the policy reference and context, the derived object records the event in memory, and its lineage field references the prior object's lineage rather than overwriting it. The result is a directed ancestry graph any downstream node can verify structurally, so provenance survives the handoff.

How to Approach the Build

The following steps are how a developer would implement this architecture. The interface sketches are illustrative and faithful to the spec; they are not a library.

1. **Define the canonical object.** Pick an extensible, hierarchical serialization format and define the six fields as individually addressable members. Illustratively:

```
AgentObject {
  intent:    <declarative goal, or absent>
  context:   <origin, trust scope, role, env>
  memory:    [<trace outcomes>]        // appended, never rewritten
  policyRef: <resolvable policy reference>
  mutation:  <authorized transformation bounds, or absent>
  lineage:   [<ancestor references>]
}
```

2. **Write the structural validator.** It answers two questions from the object alone: are the present fields coherent (at least two present, meeting the schema's presence rules), and are they compatible (permitted to coexist). It must not consult external

session state or execution history. Make it deterministic so any node reaches the same verdict.

3. **Encode the field interaction rules.** These are the compatibility checks the validator enforces: intent must align with the policy reference; mutation must be consistent with policy and context; memory updates are mandatory on authorized transformations. A conflict here is a structural failure, not a runtime error.
4. **Implement scaffolding for partial agents.** For each canonical field, define the deterministic, policy-bound rule that resolves its absence. Honor the spec's guardrails: mark scaffolded memory as scaffolded, treat a missing mutation descriptor as immutable, and record every resolution in memory. If no permissible inference path exists, reject or defer rather than invent.
5. **Build serialize and rehydrate.** Serialization must preserve field boundaries and validation metadata; rehydration on the receiving runtime must run the structural validator (and scaffolding if needed) before the agent does anything. This validate-on-receipt step is what lets a different framework accept the agent safely.
6. **Implement the mutation pathway.** Evaluate the mutation descriptor against policy and context, produce a derived object, append the event to memory, and extend (never overwrite) lineage. This keeps every move auditable.
7. **Optional: semantic templates and integrity binding.** Templates declare required and optional fields for a class of agent so instantiation is consistent across runtimes; optional cryptographic binding of field contents or lineage makes tampering detectable. The spec frames both as optional to the core model.

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 implement the object model, the validator, the scaffolding rules, the serializer, and the mutation pathway yourself, and the quality of your portability is the quality of that implementation.

It is disclosed in a patent filing, not benchmarked or productized. This guide makes no performance, latency, or reliability claims, because the spec makes none, and none should be inferred.

The schema governs semantic identity, validation, governance, and provenance. It deliberately does not prescribe execution order, scheduling, or runtime control. It does not run your agent, call your models, or orchestrate your workflow; it tells a receiving runtime whether an agent object is structurally admissible and how it may evolve. You still bring your own execution layer. The spec's fallback inference is rule-bound and does not, by default, include probabilistic or learned-model inference unless a governing policy explicitly authorizes it, so do not expect scaffolding to "smartly" reconstruct missing intent. And if an object falls below the minimum field threshold or carries irreconcilable field conflicts, it is designed to be rejected or quarantined, not force-fit into validity.

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 that a developer can build, grounded in that filing. It is not a warranty, a performance representation, or an offer of software, and nothing here should be read as claiming a shipping product, a benchmark result, or a downloadable implementation. Where third-party technologies are mentioned for context, they are described only as general background.

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)