Nick Clark · Published April 25, 2026 · PDF

How to Hand Off an AI Agent Between Two Systems That Share No Runtime

You have an agent running in one system and you need another system, one that shares no session, no memory store, and no orchestration layer with the first, to pick it up and keep going. This guide lays out the architecture for doing that: make the agent a canonical, serializable data object that carries everything a receiver needs to validate and interpret it from its own fields alone. It describes an architecture disclosed in United States Patent Application 19/452,651, not a shipping library. The home inventive step is the Agent Schema inventive step.

What You Are Building

You are building a way to move an AI agent from one system to another when the two systems have nothing in common at runtime. No shared process. No shared session table. No shared message bus, orchestration engine, or memory database. System A holds the agent now; system B, which may be a different cloud, a different vendor's stack, an edge device, or a queue consumer that wakes up hours later, needs to receive that agent and continue its work without ever having talked to system A before.

The honest answer is that you cannot do it reliably as long as the agent is a running thing. A process, a session, a control loop, and its scattered state cannot be handed anywhere; only the machine that hosts them understands them. So the real task is to

stop treating the agent as a process and start treating it as a data object that fully describes itself. That is the architecture disclosed in United States Patent Application 19/452,651, and it is what this guide teaches you to build. You implement it yourself; there is no package to install.

Why the Obvious Approaches Fall Short

The usual first attempt is to serialize whatever state you happen to have and ship it alongside the agent. In most agent frameworks, the things that actually define the agent, its goal, its memory, its trust context, and the rules it operates under, live outside the agent representation: in application logic, in a workflow engine, or in session-scoped state on the originating machine. When you serialize "the agent," you capture a payload that only means something in conjunction with all of that external context. The receiving system, which does not have the context, gets a payload it cannot interpret.

The second attempt is to make the two systems share the missing pieces: a common database the receiver reads back from, a shared session store, a central orchestrator both sides call into. This works, and plenty of production systems do exactly this. But it is precisely the thing the problem forbids. The moment handoff depends on a shared store or a central coordinator, you have reintroduced the shared runtime you were trying to avoid, and its availability becomes a hard dependency for every transfer.

The third attempt is to attach memory or metadata to the agent payload so it looks more complete. This is the right instinct, but done ad hoc it is fragile. When a payload arrives with some fields missing or degraded, the receiver either rejects it or reaches for custom repair logic that differs from system to system. Behavior becomes inconsistent across environments, and interoperability erodes. None of these approaches is foolish; each is good at something. What they share is a structural gap: nothing about the agent lets a receiver decide, from the agent alone, whether it is a valid thing to act on. That gap is what the architecture below closes.

The Architecture

The disclosed approach treats the agent as a first-class data object rather than a transient runtime process, and requires that object to carry enough internal information to be validated, interpreted, and governed by a receiving node based solely on its own composition. Several mechanisms make that work.

The agent is a structured object of canonical semantic fields. The schema defines six such fields: an intent field (the agent's goal or semantic objective, stated declaratively, without procedural steps), a context block (environmental, trust, identity, and domain metadata such as origin identifiers, trust scope, and role), a memory field (embedded trace outcomes of prior validations, mutations, delegations, and scaffolding resolutions), a policy reference field (identifiers for the governing policies that constrain what the agent may do), a mutation descriptor field (the authorized pathways by which the agent may change), and a lineage field (references to the agent's semantic ancestors). Each field is embedded directly in the object and is individually addressable and machine-readable. Because memory, policy, and provenance ride inside the object instead of in an external store, they travel with the agent across system boundaries.

A receiver validates structurally, using only what is embedded. When a node receives an agent object, it does not consult session history or a central registry. It checks two things: structural coherence (the presence of one or more canonical fields) and structural compatibility (whether the fields that are present are permitted to coexist under a set of schema rules governing cross-field dependencies). Per the disclosure, both determinations are made based only on information embedded within the object. This is what makes handoff possible without a shared runtime: the receiver's decision to accept the agent is a function of the object's own structure, not of anything the sender's environment holds.

Validation happens before any execution. Under the schema, structural validation is performed prior to any semantic execution, mutation, delegation, or propagation. Eligibility to participate is a consequence of structural coherence, not a

result of running the agent. A receiver can therefore decide whether to admit a handed-off agent without first executing it, which is exactly the property you want when accepting an object from a system you do not control.

Partial agents remain valid, so degraded handoffs do not break. A full agent carries all six fields, but the schema also defines partial agents that carry a subset and remain structurally valid. Validation begins by confirming at least two canonical fields are present; above that threshold, the receiver evaluates whether the available fields are logically compatible (for example, that mutation descriptors reference an applicable policy, that lineage references resolve to a prior state, and that memory entries are compatible with mutation scope). Where a field is missing, the schema permits resolution through structural scaffolding: a deterministic, schema-defined process that infers or defaults the missing field from context metadata, referenced policies, or lineage anchors. The disclosure is specific about the safe defaults. A missing intent is resolved only from lineage, contextual role, or policy-defined default objectives, and only where a permissible inference path exists. A missing memory field is initialized as a blank trace structure and explicitly marked as scaffolded, never fabricated as if it were prior history. A missing mutation descriptor makes the agent immutable until mutation is explicitly authorized. Every scaffolding decision is itself recorded in the memory field, so a downstream reader can tell original state from resolved state.

Serialization is designed for stateless receivers. The object is serialized so that each canonical field is individually parseable and field boundaries and reference relationships are preserved. A receiving node reconstructs the agent without prior knowledge of its execution history, its instantiation environment, or the transport it arrived over. Semantic continuity is carried by the embedded trace outcomes and lineage references rather than by any persistent session binding. The disclosure notes the encoding can be any extensible object format capable of hierarchical field representation; it does not mandate a specific one.

Lineage preserves continuity across the boundary. Each handoff or derivation extends the lineage field rather than overwriting it, forming a directed ancestry graph. The receiver can verify provenance and mutation authorization from lineage plus memory traces alone, and a chain of systems can reconstruct the full semantic ancestry after the fact without any central registry. Integrity of field contents and lineage may optionally be bound with cryptographic signatures or hashes; the disclosure states this is optional and does not change the structural validation model.

How to Approach the Build

The following steps describe how a developer would implement this architecture. The interface sketches are illustrative and faithful to the disclosure; they are not a library you can drop in.

1. Define the canonical object. Fix a serialization schema with the six named fields as individually addressable members. Keep intent declarative (a goal, not a script), and keep policy as a reference to a resolvable governing artifact, not an inlined blob. An illustrative shape:

```
Agent {
  intent:   { objective, ... }        // declarative goal, no procedure
  context:  { origin, trustScope, role, env, ... }
  memory:   [ traceOutcome, ... ]      // appended, auditable
  policy:   { references: [ ... ] }    // resolvable at validation time
  mutation: { authorizedPathways: [ ... ] }
  lineage:  { ancestors: [ ... ] }
}
```

2. Write the structural validator that both sides run. Implement a function that takes a deserialized object and returns admissible or not, using only the object's contents. It should confirm the minimum-field threshold, then check the compatibility

rules among present fields. The disclosure's rule of thumb: mutation descriptors reference an applicable policy, lineage references resolve to a prior state, and memory entries are compatible with mutation scope, all determined without interpreting semantic correctness or execution results. This validator is the contract; ship the same one to every participating system.

3. Make validation a precondition of acting. Wire the receiver so that no execution, mutation, delegation, or propagation happens until structural validation passes. An agent that fails the threshold or shows irreconcilable field conflicts is rejected, quarantined, or deferred, per your environment's policy, not silently run.

4. Implement structural scaffolding for partial agents. Add a deterministic resolver for missing fields that follows the disclosed defaults exactly: infer intent only from lineage, context, or policy defaults; initialize an absent memory field as a marked-scaffolded blank trace; treat an absent mutation descriptor as immutable-until-authorized; derive an absent lineage anchor from context or environmental trust. Record every resolution as a trace outcome in memory. Resolution must be deterministic so two receivers reach the same result.

5. Serialize for a receiver that knows nothing about you. Ensure each field round-trips independently and that no meaning depends on sender-side session state. Embed enough context, policy references, memory, and lineage that the object stands alone. Test the real property by deserializing and validating in a fresh process with no access to the origin.

6. Extend lineage on every handoff and, if you need tamper-evidence, bind it. Append to lineage rather than replacing it, and record the handoff as a trace outcome in memory. If integrity matters across an untrusted transport, add the optional cryptographic binding over field contents and lineage; keep it optional so the structural model still holds where you do not use it.

7. Standardize with templates and contracts if many agent classes exist. The disclosure describes semantic templates (which fields a class of agents requires versus allows) and contractual structures (what fallback, scaffolding, or rejection is permitted). Referencing a template from the policy field lets independent systems validate the same way without per-agent custom logic.

What This Does Not Give You

This is an architecture, not a drop-in library. There is nothing to install, and the interface sketches above are illustrative, not a working implementation; you build the schema, the validator, the scaffolding resolver, and the serializer yourself, and you own their correctness. The approach is disclosed in a patent filing. It has not been presented here as a shipping product, and this guide reports no benchmarks, throughput figures, or production results, because the disclosure states none.

The schema governs structure, not behavior. It decides whether an agent object is a valid, coherent thing to act on; it does not schedule work, prescribe execution order, run a control loop, or manage a runtime lifecycle, and the disclosure is explicit that those are out of scope. You still need an execution layer of your own on each side of the handoff. Structural validation also does not judge semantic correctness: it can confirm that fields are present and compatible, not that the agent's goal is wise or its memory true.

Scaffolding does not guarantee resolution. An object with too few fields, or with irreconcilable conflicts among context, policy, and lineage, is deemed structurally non-compliant and gets rejected, quarantined, or deferred; scaffolding never invents authority or fabricates history to force a pass. Finally, this architecture is aimed at handoff across systems with no shared runtime. If your two systems already share a reliable session store or a central orchestrator, a conventional shared-state approach may be simpler, and the object-centric model buys you the most where that sharing is exactly what you cannot assume.

Disclosure Scope

The approach described in this guide, structuring an AI agent as a canonical, serializable, self-validating object of embedded semantic fields so it can be handed between systems that share no runtime, is disclosed in United States Patent Application 19/452,651. This guide is educational. It explains an architecture a developer can implement and the tradeoffs involved; it is not a warranty, a specification of a shipping product, or an offer of software, and nothing here should be read as a promise that an implementation will behave in any particular way. Any real technologies referenced for context are described only to situate the approach and are not characterized beyond their ordinary function.

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)