Nick Clark · Published July 2, 2026 · PDF

How to Give an AI Agent Persistent Memory That Survives Restarts

If your agent forgets everything the moment its process dies, the usual fix is to bolt on an external database or a durable-execution engine that reconstructs state at each call. This guide teaches a different architecture: carrying the agent's execution state inside the agent object itself, so it resumes across restarts without re-instantiation. The approach described here is disclosed in United States Patent Application 19/538,221 (it is not a shipping library), and its home inventive step is the Memory-Resident Execution inventive step.

What You Are Building

You are building an AI agent that can stop and start without amnesia. When the process is killed, the container is recycled, the machine reboots, or the task simply sits idle for a week, the agent should pick up exactly where it left off: same objective, same accumulated history, same knowledge of what it already tried and what failed.

This is the search-intent problem behind "how do I give an agent persistent memory that survives restarts." It shows up the moment an agent has to do anything longer than a single request/response turn: a research task that spans hours, a job that waits on an external dependency, an agent that polls until a condition is met, or a fleet of agents coordinating over unreliable infrastructure.

The architecture below comes from a filed patent disclosure. Its core idea is that the execution state travels *inside the agent object* rather than being held by whatever runtime happens to be executing it. This guide teaches that design so you can implement it yourself. It is not a package you install.

Why the Obvious Approaches Fall Short

There are two mainstream ways teams solve this today, and both are legitimate. The point here is structural, not a knock on either.

External state stores. You keep the agent's code stateless and push its progress into a database, a cache, or a message queue. On restart, the agent reads its row back and reconstructs context. This works, but the state and the thing that acts on it are now two separate artifacts. Every execution node needs a live connection to the same store, the schema of "what progress looks like" lives outside the agent, and rehydrating context correctly at each invocation becomes its own source of bugs. The patent describes this general pattern in its background: conventional systems maintain execution state externally through runtimes, schedulers, or session-bound mechanisms, and must reconstruct execution context at each invocation.

Durable-execution / workflow engines. You express the agent as a workflow whose steps a central engine can replay, retry, and resume. This is powerful for deterministic pipelines. The structural cost, as the disclosure frames it, is that these systems rely on predefined task graphs and externally managed execution state coordinated by a central controller. The progress lives in the engine, not the task. That is a fine trade for fixed workflows, but it assumes a central authority governs sequencing, and it couples every long-running or adaptive task to that orchestrator.

The gap both approaches share: **the unit that carries the objective and the unit that carries the progress are different things.** When they are separate, "survives restart" becomes "successfully rejoin two separate things," which is exactly the part that

breaks across asynchronous gaps, disconnected environments, and heterogeneous nodes.

The Architecture

The disclosed approach collapses that separation. Instead of an ephemeral process whose state is held elsewhere, the task is instantiated as a **persistent executable object** that carries its own execution state. Everything below traces to United States Patent Application 19/538,221.

The object has three fields. Per the disclosure, the persistent executable object comprises:

- an **intent field**, a structured, machine-parseable descriptor of the objective or execution directive;
- a **context block**, holding identity, trust-scope, and execution-relevant metadata used for local policy evaluation; and
- a **memory field**, a structured region that records execution history: execution traces, mutation records, delegation references, policy outcomes, and reentry information.

The memory field is the load-bearing part for your problem. The spec describes it as **append-only**: prior execution records are not overwritten during mutation, delegation, or termination. Each memory entry, as disclosed, records a discrete event with a trace identifier, a timestamp, an origin-node identifier, a policy reference, an outcome descriptor, and a signature for cryptographic verification of the entry.

Execution is an evaluation cycle, run wherever the object lands. When an execution node receives the object, the disclosure describes it performing an execution evaluation cycle: parse the intent field, evaluate the context block against locally applicable policy, read the memory field for prior execution records, then select an

action based *solely* on those three inputs. The action is chosen from a defined set: execution, mutation, delegation, dormancy, reentry, or termination. The node performs the action and appends a new record describing the outcome. Continuity across lifecycles is maintained by the memory field.

That is the whole trick for surviving restarts. Because progress is a field on the object rather than in the runtime, a claim in the filing states the object persists across asynchronous execution intervals and resumes execution **without re-instantiation** based on reentry conditions encoded in its memory field. Another claim states the object can be serialized for propagation between nodes and deserialized before each cycle, so continuity is preserved independently of which node runs it.

Dormancy and reentry are how "idle for a week" works. The disclosure treats dormancy as a first-class execution action, not an error or a passive pause: an object deliberately suspends execution when conditions are unmet while remaining valid, addressable, and memory-resident. It later transitions to reentry when a reentry condition is satisfied, where those conditions are derived from stored execution history, elapsed time, or policy evaluation. The spec describes wake triggers recorded in the memory field and polling behavior in which the object repeatedly evaluates its own reentry criteria without a central scheduler or persistent connection.

Cognition, authority, and execution are kept separate. This matters for correctness and is explicit in the disclosure. Reasoning or inference (including from a probabilistic inference engine) is *advisory*: its output is recorded as an input to policy/execution evaluation and does not itself mutate the object or authorize an action. Policy evaluation decides whether an action is permitted but does not itself perform it. Only authorized state transformations, recorded in the memory field, count as execution. This separation is what keeps a confident model from unilaterally rewriting execution state.

Adaptation and composition come along for free. Because the intent field can be refined during execution (narrowed, re-scoped, reclassified) and every change is recorded as a memory entry, the object adapts without being re-instantiated as a new task. Delegation lets an object spawn subordinate objects that execute independently while linked by memory-resident lineage references, and their outcomes are aggregated back into the parent's memory field. The disclosure notes coordination here emerges from object-resident state and lineage, not from a central controller.

How to Approach the Build

You implement this yourself. The steps below are the order a developer would follow, with illustrative interface sketches that are faithful to the disclosure. Treat the sketches as shape, not as shipping code.

1. Define the object schema first, not the runtime. Your durable unit is the object. Sketch it as three fields (illustrative only):

```
PersistentObject {
  intent:  { descriptor, constraints }        // the objective
  context: { identity, trustScope, policyRefs } // for local policy checks
  memory:  [ MemoryEntry ]                      // append-only history
}
MemoryEntry { traceId, timestamp, originNode, policyRef, outcome, signature }
```

Make `memory` append-only in your data model from day one. The disclosure's continuity and auditability properties depend on never overwriting prior records.

2. Make the object serializable end to end. Restart-survival reduces to: serialize the object, persist or transmit the bytes, deserialize, run the next cycle. Verify that a round-trip through serialization reproduces an object that resumes correctly. Per the disclosure this is what makes continuity independent of node identity.

3. Write the evaluation cycle as a pure function of the object. The node should decide what to do using only the parsed intent, the evaluated context, and the memory field. Sketch:

```
function evaluate(obj, localPolicy):
    op       = parse(obj.intent)
    eligible = policyCheck(obj.context, localPolicy)   // local, no central
    history  = read(obj.memory)
    action   = select(op, eligible, history)  // execution | mutation | dele
                                              // | dormancy | reentry | ter
    outcome  = perform(action)
    obj.memory.append(record(action, outcome))
    return obj
```

Keep this function free of hidden external state. The spec's stateless-node embodiment has the node store nothing about the object outside the object's own memory field; one claim makes this explicit.

4. Model the lifecycle as an explicit state machine. Implement the states named in the disclosure: instantiated, evaluation, execution, mutation, delegation, dormant, reentry, terminated. Each transition should append a memory entry. This is what turns "a process that runs" into "a stateful progression you can stop and resume."

5. Implement dormancy and reentry deliberately. Decide dormancy as an action when conditions are unmet, and store a wake trigger (elapsed time, an accumulated outcome, a satisfied prerequisite, an observed event) in the memory field. On any node that later encounters the object, evaluate reentry locally against that trigger. This is how idle-for-a-long-time works without holding a connection open.

6. Separate the decider from the doer. Route any model or inference output into policy/execution evaluation as an *input*, and only apply state changes through the execution path. Do not let the reasoning step write to the object directly. The disclosure

is emphatic that cognition is advisory and does not authorize execution.

7. Record outcomes as signals, not just success/failure. The spec treats latency, timeout, partial results, non-response, and "negative capability" (this node/context could not satisfy the intent) as first-class recorded signals that shape later behavior, including a semantic backoff that paces retries from recorded outcomes rather than a fixed timer. Capture these in the outcome descriptor so a resumed object reasons about where and when to try next.

8. Add delegation last, if you need it. Once a single object resumes reliably, let an object spawn subordinate objects linked by lineage references in memory, and aggregate their outcomes back into the parent. Get single-object continuity solid first.

What This Does Not Give You

Be clear-eyed about the boundaries.

- **This is an architecture, not a drop-in library.** There is no package to install and no SDK behind this guide. You design the schema, the serializer, the evaluation cycle, and the state machine yourself.
- **It is disclosed, not benchmarked.** The approach is described in a patent filing. This guide makes no performance, throughput, or reliability claims, and nothing here has been presented as a production-proven or benchmarked system. Any latency, memory, or scaling characteristics are yours to measure once you build it.
- **The hard parts are still hard.** Serialization correctness, schema evolution of long-lived objects, durable and tamper-evident storage of the memory field, signature/verification key management, and safe concurrent evaluation of the same object on different nodes are all real engineering you own. The disclosure names these as properties of the design (append-only history, signed entries, local policy) but implementing them well is on you.

- **It does not replace every workflow engine.** If your problem is a fixed, deterministic pipeline that benefits from centralized replay and a single source of orchestration truth, a durable-execution engine may fit better. This architecture targets long-horizon, adaptive, or trust-divergent execution where you specifically want progress to live with the task and decisions to be made locally.
- **"No central coordinator" is a design stance with costs.** Local, independent evaluation means different nodes can legitimately reach different decisions for the same object. The disclosure treats that as a feature preserved by the append-only audit trail, but you must design for eventual consistency rather than assume a single global ordering.

Disclosure Scope

The architecture described in this guide is disclosed in United States Patent Application 19/538,221 ("Memory-Resident Execution of Persistent Executable Objects in Distributed Computing Systems"). Every mechanism attributed to the approach above (the intent/context/memory object structure, the append-only memory field, the local evaluation cycle, the execution/mutation/delegation/dormancy/reentry/termination action set, resumption without re-instantiation, and the cognition/authority/execution separation) traces to that filing. This guide is educational: it teaches how to approach building such a system yourself. It is not a warranty, a guarantee of results, an offer of software, or a grant of any license, and it does not describe a shipping product.

Memory-Resident Execution (/memory-resident-execution) All 40 steps → (/inventive-steps)

Persistent objects that execute without orchestration.

U.S. 19/538,221 (/patents/19-538221)

PRIMARY TECHNICAL DISCLOSURE

- Memory-Resident Execution: Persistent Semantic Objects Without Orchestration (/articles/memory-resident-execution-persistent-semantic-objects-without-orchestration)

SECONDARY TECHNICAL

- Six-Action Execution Evaluation Cycle: Parse, Evaluate, Select at Every Node (/articles/memory-resident-execution/execution-cycle)
- Cognition-Authority-Execution Separation: Reasoning Cannot Authorize Action (/articles/memory-resident-execution/cognition-authority-separation)
- Dormancy as First-Class Execution State: Valid Suspension Without Failure (/articles/memory-resident-execution/dormancy-state)
- Semantic Backoff: Retry Pacing From Execution Outcomes Rather Than Fixed Timers (/articles/memory-resident-execution/semantic-backoff)
- Wake Triggers for Dormancy Exit: Explicit Reentry Conditions in Memory (/articles/memory-resident-execution/wake-triggers)
- Persistent Polling Behavior: Autonomous Condition Evaluation Without Schedulers (/articles/memory-resident-execution/persistent-polling)
- Intent Refinement During Execution: Adaptive Objectives Without Re-Instantiation (/articles/memory-resident-execution/intent-refinement)
- Compositional Execution Through Recursive Delegation: Parent-Child Lineage Tracking (/articles/memory-resident-execution/recursive-delegation)
- Negative Capability Signals: Recording What Cannot Be Done as Structured Constraint (/articles/memory-resident-execution/negative-capability)
- Swarm-Based Execution Emergence: Coordinated Behavior Without Centralized Control (/articles/memory-resident-execution/swarm-execution)
- Latency and Failure as Semantic Signals: Structured Inputs From Adverse Conditions (/articles/memory-resident-execution/failure-signals)
- LLM as Advisory Execution Node: Inference Without Authority Over Agent State (/articles/memory-resident-execution/llm-advisory-node)
- Append-Only Memory Field: Preserving Execution Lineage Through Appended Records (/articles/memory-resident-execution/append-only-memory)

APPLICATIONS · GENERAL

- Execution Continuity for DDIL Coalition C2: Memory-Resident Tasking Across Disconnected, Trust-Divergent Tactical Networks (/articles/memory-resident-execution/defense-tactical-edge-ddil)

- Stateful Serverless: Eliminating Cold Starts and State Loss in FaaS (/articles/memory-resident-execution/serverless-persistence)
- Long-Running Business Workflows Without an Orchestration Engine (/articles/memory-resident-execution/long-running-workflows)
- Autonomous Drone Operations Surviving Ground Control Link Loss (/articles/memory-resident-execution/autonomous-drone-operations)
- Deep Space Agent Execution Without Ground Control (/articles/memory-resident-execution/space-exploration-agents)
- Autonomous Underwater Vehicle Mission Autonomy Without Surface Connectivity (/articles/memory-resident-execution/underwater-robotics)
- Offline Clinical Agents for Rural Healthcare With Intermittent Connectivity (/articles/memory-resident-execution/rural-healthcare-agents)
- Disaster Response Software That Works When Infrastructure Is Destroyed (/articles/memory-resident-execution/disaster-zone-operations)
- Offline Payment Agents That Stay Compliant When the Network Drops (/articles/memory-resident-execution/offline-financial-agents)

APPLICATIONS · SPECIFIC

- Cloudflare Durable Objects vs Memory-Resident Execution: Who Holds Authority Over the Object (/articles/memory-resident-execution/durable-objects)
- Azure Durable Actors Alternative: Governed, Cross-Domain Execution Beyond Service Fabric Reliable Actors (/articles/memory-resident-execution/azure-actors)
- Akka Alternative: Governed, Self-Executing Objects Beyond the Reactive Actor Model (/articles/memory-resident-execution/akka)
- Microsoft Orleans Alternative: Governed, Cross-Domain Execution Beyond Silo-Cluster Grains (/articles/memory-resident-execution/orleans)
- Dapr Alternative for Governed State: Where Authority Lives When State Moves (/articles/memory-resident-execution/dapr)
- wasmCloud vs Memory-Resident Execution: Message-Reactive Actors and Self-Executing Objects (/articles/memory-resident-execution/wasmcloud)
- Spin Alternative for Governed Agents: WebAssembly Serverless vs Memory-Resident Execution (/articles/memory-resident-execution/spin)
- Fermyon Spin vs a Persistent Executable Object: Which Hosts Governed Agents That Carry Their Own Policy and Lineage? (/articles/memory-resident-execution/fermyon)
- Fly Machines Alternative: Governed, Self-Carrying Execution Beyond Externally Orchestrated Micro-VMs (/articles/memory-resident-execution/fly-machines)

- Railway Alternative for Long-Running Autonomous Services: Memory-Resident Execution vs Trigger-Driven Deployment (/articles/memory-resident-execution/railway)
- Temporal alternative: object-resident execution state versus a durable-execution service (/articles/memory-resident-execution/temporal)
- Restate vs object-resident execution state: where durable execution keeps the journal (/articles/memory-resident-execution/restate)
- AWS Step Functions alternative: where does execution state live, in the orchestrator or in the object? (/articles/memory-resident-execution/aws-step-functions)
- Golem vs object-resident execution state: who carries the task state across nodes? (/articles/memory-resident-execution/golem)

Memory-Resident Execution overview → (/memory-resident-execution)