Nick Clark · Published April 25, 2026 · PDF

How to Run a Long-Running Agent Workflow That Survives Crashes

If you build agents that run for hours or days, a crash, a redeploy, or a node going offline can wipe out everything the agent was in the middle of doing. This guide describes an architecture for making execution survive interruption by carrying the agent's state inside the agent object itself, so it can resume without being rebuilt. The approach here is disclosed in United States Patent Application 19/538,221; it is an architecture you implement, not a library you install. Its home inventive step is the Memory-Resident Execution inventive step.

What You Are Building

You want an agent workflow that can run across a long horizon (minutes, hours, or days) and pick back up correctly after something interrupts it. The interruption might be a process crash, a container redeploy, an execution node going offline, or simply a deliberate pause while the agent waits for an external condition it cannot control yet.

The people who hit this are building autonomous agents, multi-step reasoning pipelines, long-running orchestration jobs, and workflows that span asynchronous or intermittently available systems. The failure they keep running into is the same one: the agent's progress lived somewhere other than the agent, and when the surrounding machinery went away, the progress went with it.

This guide teaches an architectural approach to that problem, grounded entirely in the disclosure of United States Patent Application 19/538,221, which describes memory-resident execution of persistent executable objects. The core idea is to stop treating your agent as an ephemeral process and start treating it as a persistent object that carries its own execution state. You will build this yourself; there is no drop-in package being offered here.

Why the Obvious Approaches Fall Short

The usual way to keep a long job alive across failures is to externalize its state. That is a legitimate and widely used pattern, and it is worth stating accurately.

External orchestration engines. Workflow engines, business process managers, and job schedulers track progress, retries, and failure handling on behalf of the task. The task itself is a set of stateless steps; a controller outside the task remembers where you are. This works, but it means execution state is coupled to that external controller. The task cannot resume on its own; it can only resume when the specific orchestrator that was tracking it brings it back. The patent's background describes this class of systems as relying on predefined task graphs, transactional state transitions, and externally managed execution state.

Stateless calls plus a database checkpoint. A common hand-rolled version: after each step, write progress to a database or queue, and on restart, read it back and reconstruct context. This also works, but the reconstruction is the fragile part. The mapping between "rows in a table" and "what the agent was actually thinking" is code you maintain by hand, and it drifts. Every new field the agent needs to remember is another column, another migration, another place the rehydration logic can be wrong.

Prompt-response chaining for LLM agents. Many agent frameworks treat each model interaction as an isolated prompt and response, with an external loop deciding what to do next. Context has to be re-assembled and re-supplied on every turn. The

patent's background notes this leads to repeated rehydration of context and limits the object's ability to persist intent, history, or governance across time.

The structural gap in all three is the same: **the execution state is not part of the thing being executed.** So "surviving a crash" always means "reconstructing state from somewhere else," and reconstruction is where correctness leaks out.

The Architecture

The disclosed approach inverts the relationship. Instead of the runtime holding the agent's state, the agent object holds its own state, and any execution node can pick it up. The spec calls this object a persistent executable object (also referred to as a semantic object). Every mechanism below traces to US Patent Application 19/538,221.

State lives in three fields inside the object. The persistent executable object carries an intent field, a context block, and a memory field. The intent field encodes a machine-readable execution descriptor: what the object is trying to do. The context block encodes execution-relevant metadata: identity, trust scope, and execution context used for local policy evaluation. The memory field records execution history as an append-only log of prior execution state. Because all three travel inside the object, the object is self-contained.

Execution is a state machine over the object, not an ephemeral process. The spec defines a lifecycle in which the object progresses through states rather than running as a single throwaway invocation: instantiated, evaluation, execution, mutation, delegation, dormancy, reentry, and termination. Each execution node that receives the object runs an execution evaluation cycle: parse the intent field, evaluate the context block against locally applicable policy, read the memory field for prior records, and then select an execution action from a defined set: execution, mutation, delegation, dormancy, reentry, or termination.

Every step appends to the memory field. After the selected action runs, the node records the outcome by appending a new execution record to the memory field. The spec describes each memory entry as carrying a trace identifier, a timestamp, an origin node identifier, a policy reference, an outcome descriptor, and a signature for cryptographic verification. The log is append-only, which is what makes execution continuity and auditability hold across cycles. Per the spec, "execution continuity across multiple execution lifecycles is maintained by the memory field."

Resumption needs no re-instantiation. This is the property that answers the crash question directly. The claims state that the persistent executable object "persists across asynchronous execution intervals and resumes execution without re-instantiation." Because the object already contains its intent, its context, and its full execution history, a node that picks it up after an interruption reads the memory field, sees what has already happened, and continues from there. There is no separate rehydration layer to keep in sync, because the state was never externalized in the first place.

Dormancy is a first-class state, not an error. The spec is emphatic that a "pause" is a deliberate execution decision, distinct from failure and from termination. An object enters dormancy when execution is currently inadvisable, inefficient, unsafe, or non-optimal, and it stays a valid, addressable, evaluable execution entity while dormant. It is not discarded, reset, or re-instantiated. Reentry happens when reentry conditions, derived from stored execution history, elapsed time, or policy, are satisfied. This is what lets the same architecture handle "the machine crashed" and "we are intentionally waiting for an external condition" with one mechanism: in both cases the object simply resumes from its own memory.

No centralized coordination is required. Each execution node decides locally, using only the object's contents and locally applicable conditions. The spec defines "without centralized coordination" carefully: no centralized scheduler, controller, orchestrator, or master node governing sequencing; no shared global execution state outside the object; no required consensus before execution. It also explicitly clarifies

that this does not preclude distributed communication, eventual consistency, or independent verification by other nodes. The point is that no single external authority owns the workflow's progress, so no single external failure can lose it.

How to Approach the Build

These are the steps a developer would follow to implement the architecture. The sketches below are illustrative only and are meant to be faithful to the spec's structure, not to be copy-paste production code.

1. Model the object, not the process. Define a serializable object with the three fields. Everything the agent needs to survive a crash must live here.

```
// Illustrative sketch, not a shipping API.
PersistentObject {
  intent:  { descriptor, constraints }        // what to do
  context: { identity, trustScope, metadata }  // for local policy
  memory:  [ MemoryEntry, ... ]                // append-only history
}
MemoryEntry { traceId, timestamp, originNode, policyRef, outcome, signature }
```

2. Make serialize/deserialize the boundary of every cycle. The object must be fully serializable to durable storage between cycles and fully reconstructable from that serialized form, with no side state held in the process. Persist after each appended memory entry so an interruption at any point leaves a valid, resumable object.

3. Implement the execution evaluation cycle as a pure function of the object. Given a deserialized object, parse intent, evaluate context against local policy, read memory for prior records, then select exactly one action from execution, mutation, delegation, dormancy, reentry, or termination. Selecting based only on the object's own fields is what keeps resumption independent of any external controller.

4. Append outcomes; never overwrite. After the action runs, append a new memory entry describing what happened. Retry counts, failures, latencies, and partial results all belong in the memory field, because per the spec they become the signals that drive later decisions.

5. Make dormancy and reentry explicit states. When conditions are unmet, write a dormancy decision into memory rather than throwing or looping. On the next poll, an execution node reads the object, evaluates reentry conditions from the memory field and context block, and either resumes or stays dormant. The spec describes reentry timing driven by conditions derived from recorded history (for example, elapsed time or recovery from a prior failure pattern) rather than a fixed schedule.

6. Define terminal conditions. Give the object explicit terminal conditions: objective satisfied, constraints expired, a policy cutoff, or accumulated failures beyond a threshold. On termination the object keeps its final history and stops. This prevents dormant objects from lingering forever.

7. Let any node pick up any object. Because state is object-resident, resumption is not tied to the node that started the work. On restart, load persisted objects and run the evaluation cycle. A crashed run and a deliberately deferred run resume through the same code path.

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 serialization boundary, the evaluation cycle, the policy evaluation, and the durable storage yourself.

It is disclosed in a patent filing, not shipped as a benchmarked or production-proven product. This guide makes no performance, throughput, or reliability claims, because the spec states none, and you should measure your own implementation.

The approach also does not remove your hard problems; it relocates them. You still need durable storage that does not lose the serialized object, and your serialize/deserialize boundary must be disciplined, because any state you keep in the process instead of in the object is state that will not survive a crash. Your evaluation cycle should be safe to re-run, since a node may crash after acting but before persisting. The spec records outcomes to an append-only, signed memory field for auditability, but it does not, by itself, hand you a transaction manager or an exactly-once guarantee across your storage layer; those remain your engineering responsibility.

Finally, this is aimed at long-horizon, resumable, possibly-distributed workloads. For a short synchronous request that either completes or is simply retried from scratch, the object-resident machinery is more structure than the job needs.

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." This guide is educational. It explains an architectural approach so that a skilled developer can understand and build it, and it is not a warranty, a guarantee of fitness for any purpose, or an offer of software. Every description of how the approach works is drawn from that filing; where the filing is silent, this guide makes no claim.

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)