Nick Clark · Published April 25, 2026 · PDF

AWS Step Functions alternative: where does execution state live, in the orchestrator or in the object?

AWS Step Functions is a managed orchestration service that keeps the state of a long-running workflow inside an external state machine that AWS runs and tracks for you. It is a mature, reliable way to coordinate distributed steps, and it removes a large amount of undifferentiated retry and checkpointing code. This article, built on the Memory-Resident Execution inventive step disclosed in United States Patent Application 19/538,221, examines a specific architectural axis: Step Functions holds execution state in an orchestrator external to the work being coordinated, whereas memory-resident execution carries execution state, decision history, and governance inside the object itself so it persists across asynchronous intervals and resumes without re-instantiation, evaluated locally without centralized coordination.

What AWS Step Functions Does

AWS Step Functions is a managed workflow orchestration service. A developer defines a state machine, typically in Amazon States Language, describing a set of states such as tasks, choices, parallel branches, maps, waits, and error handling, and Step Functions

executes that definition, advancing from state to state, invoking the underlying work (commonly Lambda functions, but also many other AWS service integrations), and recording the progress of each execution.

At the architecture level, the well-documented properties are:

- **Externally managed execution state.** The service maintains the state of each running execution: which state is current, what input and output flowed between states, and the history of transitions. The individual steps do not have to persist that progress themselves.
- **Declarative workflow definition.** Control flow, sequencing, branching, retries, and error handling is expressed as a definition the service interprets, separating orchestration logic from step implementations.
- **Durable, long-running executions.** Standard workflows can run for extended periods and survive individual step failures, with built-in retry and catch semantics and a recorded execution history that supports auditing and debugging.
- **Broad AWS integration.** Step Functions integrates directly with many AWS services and supports patterns such as waiting for an external callback, making it a strong fit for coordinating multi-service processes inside AWS.

These are real strengths. For coordinating distributed steps that live inside AWS, an externally managed state machine with declarative retries and a durable execution history is exactly the right shape, and it is a genuine simplification over hand-rolling checkpointing, retry loops, and progress tracking in each service.

The Architectural Axis

The axis here is not that Step Functions is missing a feature. It is a structural choice about where execution state lives, and it is inherent to the managed-orchestrator model rather than particular to any one product.

In Step Functions, the execution state lives in the orchestrator, external to the work being coordinated. The state machine definition and the running execution's progress are held and advanced by the service. The steps are, by design, largely stateless units the orchestrator invokes; the memory of what has happened, what comes next, and whether the process may proceed is a property of the external state machine, not of the units doing the work. That is deliberate and reasonable: externalizing state is precisely what lets the service furnish durable, resumable, auditable executions on the developer's behalf.

The consequence is that the coordinating state is bound to the orchestrator. The steps can be moved or reused, but the thing that carries progress, decision history, and eligibility to proceed does not travel with them; it stays in the service. For processes that live entirely inside AWS and want that durability handed to them, this is a fine trade. The comparison below concerns only the case where the execution state itself needs to be portable, local, and survivable across domains that do not share an orchestrator.

How the Disclosed Approach Differs

The invention disclosed in United States Patent Application 19/538,221 relocates the execution state into the object being executed. Rather than an external state machine advancing stateless steps, the unit of execution is a persistent executable object whose execution state, decision history, and governance rules are intrinsic properties of the object itself.

Grounded in the specification, a persistent executable object comprises an intent field (a machine-parseable execution descriptor encoding the object's objective, which may itself be refined or reclassified during execution based on recorded history), a context block (identity and trust-scoped metadata used for local policy evaluation), and a memory field (an append-only execution history of traces, mutation records, delegation references, policy outcomes, and reentry information).

Each execution node that receives the object performs an execution evaluation cycle: it parses the intent field, evaluates the context block against locally applicable policy without reliance on centralized coordination, reads the memory field for prior execution records, and selects one action from a fixed set, execution, mutation, delegation, dormancy, reentry, or termination. The outcome is appended to the memory field. As the specification states, execution continuity across multiple execution lifecycles is maintained by the memory field of the object, not by an external scheduler, controller, or session-bound runtime.

Several properties fall directly out of this structure and distinguish the model on the axis above:

- **State travels with the work.** The intent, context, and memory fields are carried inside the object. A node that has never seen the object before reconstructs the full execution situation by reading those fields, because the object is serialized for propagation and deserialized prior to each evaluation cycle, so continuity is preserved independently of node identity. The specification is explicit that a node need not store execution progress, eligibility, or history for the object outside the object's own memory field.
- **Resumption without re-instantiation.** The object persists across asynchronous execution intervals and resumes based on reentry conditions encoded in the memory field. Dormancy is a deliberate, first-class execution action, distinct from failure or termination; wake triggers and reentry conditions derive from the object's own memory field and context block, so a resumption after a long interval needs no persistent connection, synchronized clock, or centralized scheduler.
- **Local authority, no centralized coordination.** Policy references travel inside the object, and each node evaluates them against its own local conditions. The specification is explicit that different nodes, under different local policy, may lawfully select different actions for the same object while continuity and auditability

are preserved through the append-only memory field. Execution progression is not constrained to a predefined task graph, and execution state is not externalized to an orchestration layer.

- **Adaptive intent rather than a fixed graph.** The intent field is not static; it may be refined, constrained, expanded, or reclassified during execution based on recorded history, letting the object adapt without terminating or being re-instantiated as a new object.
- **Cognition, authority, and execution are separated.** Reasoning or inference components can recommend an action, but their output is advisory and does not itself mutate state or grant permission. Authority is policy evaluation; execution is the application of authorized state transitions, so a high-confidence recommendation cannot bypass a policy constraint.

Where They Fit Together

These are not mutually exclusive, and the comparison is not a scorecard. For a workflow whose steps all run inside AWS and whose principal requirement is durable, auditable coordination handed to the developer by a managed service, Step Functions is the better fit precisely because the orchestrator holds the state.

The two can also compose. A persistent executable object can be evaluated at a node that happens to be a Lambda function invoked by a Step Functions state, carrying its own intent, memory, and policy while the surrounding AWS workflow coordinates the parts of the process that live inside AWS. Equally, an object could originate inside an AWS-coordinated process, then propagate out to nodes in other domains where no shared orchestrator is reachable, carrying its execution state with it. The memory-resident model matters when the requirement is inverted: the execution state must be portable, local, and survivable across administrative and trust domains that do not share an orchestrator, and continuity must be a property of the object rather than of a single operator's control plane.

Boundary Conditions

Several honest limits apply. Memory-resident execution gives up a centrally enforced, single-source-of-truth view of progress in exchange for object-resident state that any receiving node can evaluate locally; where a workload genuinely wants one authoritative external record advanced by one operator, an external orchestrator expresses that more directly. Carrying execution history inside the object means the object grows as its append-only memory field accumulates records, a design consideration for very long-lived objects. The model's guarantees concern preserving and serializing execution-state transitions within the object and recording all decisions in the memory field; it does not by itself furnish a globally enforced ordering or a global lock across nodes, and the specification frames coordination as emerging from lineage and accumulated history rather than from consensus.

This is an early-stage disclosure. The subject matter is a patent application, United States Patent Application 19/538,221, which describes embodiments and enumerated variations rather than a shipped, independently benchmarked product; claims about what the approach does trace to the specification, not to measured production results. The comparison above is scoped to a single architectural axis and is not a general ranking of either approach.

Disclosure Scope

The technical statements in this article about persistent executable objects, the intent field, context block, and memory field structure, local policy evaluation without centralized coordination, the fixed execution-action set, dormancy and reentry, resumption without re-instantiation, adaptive intent, delegation and lineage, and cross-domain continuity, are grounded in and limited to the disclosure of United States Patent Application 19/538,221 ("Memory-Resident Execution of Persistent Executable Objects in Distributed Computing Systems"). This article is a dated public technical disclosure tied to that filing. All statements about AWS Step Functions and the broader

managed-orchestration and durable-execution category are provided as external market and architectural context based on publicly available product documentation, and describe those systems as of the publication date. They are not claims of the filing, are not a representation on behalf of Amazon Web Services or any other named provider, and are intended as a fair, architecture-level characterization rather than an assertion of any provider's defect. Product names are the marks of their respective owners and are used for identification and comparison only.

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)