Nick Clark · Published April 25, 2026 · PDF

How to Run AI Agents on Intermittent or Offline Edge Devices

You need an agent to keep making progress on a device that spends most of its life asleep, disconnected, or resource-starved, and the usual answer is a scheduler and a control plane the device cannot reach. This guide describes a different architectural approach: carry the execution state inside the agent object itself, so it can go dormant and wake on its own terms without anything coordinating it. The approach is disclosed in United States Patent Application 19/538,221, and its home concept is the Memory-Resident Execution inventive step. This is an architecture you build, not a library you install.

What You Are Building

You are building an AI agent that survives long idle or disconnected intervals on an edge device and resumes exactly where it left off, with no central scheduler, orchestrator, or always-on connection deciding when it runs.

Think of a sensor gateway that comes online for a few minutes a day, a field device on a flaky cellular link, a battery-powered node that deep-sleeps between wakeups, or a laptop agent that must keep pursuing a goal across shutdowns. In every case the agent has a multi-step objective, it cannot count on the network being there when it wants to act, and there is no reliable external system to hold its progress for it.

The goal is an agent that can decide, entirely from state it carries with it, whether to act now, wait, adapt its objective, or stop, and that can pick up a paused objective days later without being re-created from scratch. The approach below traces to the disclosure in US Patent Application 19/538,221, which describes carrying execution state inside the executable object rather than in an external runtime.

Why the Obvious Approaches Fall Short

The conventional way to run long-lived or resumable work is to keep the execution state outside the task. A scheduler, orchestration layer, or workflow engine tracks progress, retries, and completion, and the task itself is an ephemeral process that gets reconstructed at each invocation. The application notes plainly that conventional systems maintain execution state externally through runtimes, schedulers, orchestration layers, or session-bound control, so context and decision logic must be rebuilt at each call.

These are legitimate, widely used patterns, and on a connected server they work well. The structural problem appears at the edge. If the coordinator lives off-device, an intermittently connected node cannot consult it when it needs to make a decision, and the disclosure observes that these approaches introduce points of failure precisely when execution spans asynchronous or disconnected environments. A cron-style scheduled retry does not know why the last attempt failed and fires on a fixed clock whether or not the situation improved. A workflow engine assumes a predefined task graph and deterministic progression, which does not describe a device that may be offline for an unknown span and must reason about when acting is even worth trying.

The gap is not that these tools are bad. It is that they assume the coordinator is reachable and the state lives somewhere central. On an intermittent edge device, neither assumption holds. The disclosure frames this as a need for execution state that is an intrinsic property of the object rather than something an external system holds.

The Architecture

The core move is to make the agent a persistent, memory-resident object that carries its own execution state, and to make going dormant and waking up first-class things the object does, not failure conditions imposed on it.

The disclosed object has three parts. An **intent field** encodes a machine-readable execution descriptor, the objective the agent is pursuing. A **context block** carries identity, trust-scope, and execution-relevant metadata used for local policy evaluation. A **memory field** stores an append-only execution history: traces, outcomes, mutation records, delegation references, and policy outcomes accumulated over the object's life. Per the disclosure, execution continuity across multiple execution lifecycles is maintained by that memory field.

Whenever an execution node receives the object, it runs an **execution evaluation cycle**: it parses the intent field, evaluates the context block against locally applicable policy without any centralized coordination, reads the memory field for prior records, and then selects one action from a fixed set, execution, mutation, delegation, dormancy, reentry, or termination, based solely on those three inputs. It performs the action and appends a new record. Nothing outside the object is consulted for the decision.

Dormancy is a deliberate choice, not a stall. The disclosure treats the transition to dormancy as an execution action selected when acting now is inadvisable, inefficient, unsafe, or non-optimal given the intent, context, memory, and local policy. It is explicitly distinguished from failure (an inability to complete) and termination (a terminal condition being satisfied). A dormant object stays valid, addressable, and evaluable; it is not discarded, reset, or re-instantiated. This is what lets an agent sit through a long offline window without losing anything or spinning.

Reentry is driven by the object, not a clock elsewhere. The disclosure defines reentry as resumption following dormancy when one or more reentry conditions are satisfied based on execution history, elapsed time, or policy evaluation. Reentry criteria

named in the disclosure include elapsed time, accumulated execution outcomes, satisfaction of prerequisite conditions, or changes in execution context observed by a node. Dormancy may also carry explicit **wake triggers** recorded in the memory field, corresponding to elapsed time, accumulated outcomes, context changes, prerequisite satisfaction, or externally observed events; the object stays dormant until a trigger is satisfied, at which point reentry evaluation occurs without centralized scheduling or an external notification mechanism.

Retry pacing is derived from history, not a fixed timer. The disclosure describes semantic backoff, adjusting execution pacing based on outcomes recorded in the memory field such as partial success, negative capability signals, or policy constraints, rather than uniform intervals applied independent of context. Repeated latency beyond a threshold may extend the retry interval or push the object back into dormancy; recovery may satisfy reentry criteria and trigger resumption.

Two properties make this fit the edge specifically. First, the disclosure's stateless deployment mode has nodes derive every decision, policy evaluation, mutation, and trace, entirely from information embedded in the object, appending outcomes before propagating or storing it. Second, the disclosed edge-oriented mode contemplates resource-constrained nodes operating intermittently or asynchronously, where execution continuity is preserved despite limited connectivity because the state is carried in the object, and nodes defer, enter dormancy, or reenter based on locally evaluated conditions. The disclosure states this continuity is maintained without open connections, synchronized clocks, or centralized schedulers.

How to Approach the Build

You are implementing this yourself. The steps below follow the disclosed architecture; the sketches are illustrative and simplified, not a package to install.

1. **Define the object schema.** Give your agent object three regions: intent, context, and memory. Keep memory append-only so history is never overwritten during mutation, delegation, or termination. A memory entry in the disclosure carries a trace identifier, timestamp, origin-node identifier, policy reference, outcome descriptor, and a signature for verification. An illustrative shape:

```
AgentObject {
  intent:  { descriptor, constraints }
  context: { identity, trustScope, eligibility }
  memory:  [ { traceId, ts, node, policyRef, outcome, sig }, ... ]   // ap
}
```

2. **Make the object self-contained and serializable.** The disclosure serializes the object for propagation and deserializes it before each evaluation cycle, so continuity is independent of which node runs it. On an edge device this is your durable, on-device representation: write it to non-volatile storage before sleep, load it on wake.
3. **Write the evaluation cycle as a local function.** On each wake or receipt: parse intent, evaluate context against local policy, read memory, then select one action from {execute, mutate, delegate, dormancy, reentry, terminate}. Consult nothing external for the decision. Append an outcome record afterward.
4. **Implement local eligibility policy.** The disclosure evaluates node-local eligibility conditions: resource sufficiency, isolation/sandbox constraints, rate-limit or cooldown conditions, and retry or attempt thresholds derived from memory. If compute, memory, time window, or connectivity do not permit the authorized action, select dormancy or defer instead of forcing execution.
5. **Encode dormancy with explicit wake triggers.** When you choose dormancy, record the wake triggers in memory: an elapsed-time trigger, a prerequisite-satisfied trigger, an accumulated-outcome trigger, or an observed-event trigger. On the next wake, the object evaluates whether any trigger is met before doing work.

6. **Derive reentry and backoff from memory, not a fixed clock.** Compute the next retry interval from recorded outcomes. Extend it after repeated latency or failure; shorten or trigger reentry when conditions recover. This is the difference between the object reasoning about when to act and a dumb periodic retry.
7. **Record terminal conditions.** Let the object self-terminate when a terminal condition is recorded, objective satisfied, a policy cutoff reached, or failure accumulated beyond threshold, and keep dormancy distinct from termination so a paused agent is never mistaken for a finished one.
8. **Keep cognition advisory.** If you use a model to recommend an action, the disclosure treats that output as advisory input to policy or execution evaluation, recorded as an outcome, not as something that authorizes or performs the action itself. This separation lets a model be absent, offline, or wrong without the agent losing its integrity.

A practical loop on the device: wake, load and deserialize the object, run one evaluation cycle, append the outcome, persist, then either act or go back to sleep with fresh wake triggers.

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 schema, the evaluation cycle, the local policy, and the persistence yourself, on your own device and runtime.

The approach is disclosed in a patent filing. It has not been presented here as a shipping product, and this guide contains no benchmarks, performance figures, or production-proven guarantees, because the disclosure states none and you should not assume any. Any timing, energy, or reliability characteristics are yours to measure on your hardware.

The architecture also does not remove real edge constraints. It carries state through offline intervals, but it does not manufacture connectivity, compute, or power. If an action fundamentally requires a network resource that never becomes available, the object can defer, adapt its intent, or terminate, but it cannot complete the impossible. Wake triggers still depend on the device actually waking, so integration with your platform's real sleep and wake mechanism is on you. And where you genuinely have a reliable central coordinator and always-on connectivity, a conventional scheduler or workflow engine may be the simpler fit; the value here is specifically the intermittent, disconnected, coordinator-absent case.

Disclosure Scope

The architectural approach described in this guide, memory-resident execution in which a persistent executable object carries its intent, context, and append-only execution history and selects execution actions including dormancy and reentry through local evaluation without centralized coordination, is disclosed in United States Patent Application 19/538,221. This guide is educational. It explains how to approach building such a system yourself and is not a warranty, a specification, a benchmark, or an offer of software, and nothing in it should be read as a promise of performance or fitness for any particular purpose.

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)