Evaluating Durable Functions for Stateful Serverless Computing

Julio Sigüenza Pacheco

Abstract

Serverless platforms are traditionally stateless, forcing developers to manage state explicitly through external storage systems. In December 2025, AWS introduced Durable Functions for AWS Lambda, a new execution model that enables stateful serverless workflows through checkpoint-and-replay semantics.

This paper presents a hands-on evaluation of AWS Lambda Durable Functions using two representative workloads: a stateful counter and a video encoding pipeline. We implement each workload in two variants: a durable execution and a traditional Lambda-based baseline with explicit state storage. Using 20 runs for the counter and 30 runs for the video pipeline, we measure execution time, coordination overhead, and failure-handling behavior.

Our results show that durable execution completely eliminates explicit state storage, reducing external coordination from an average of 5 operations per counter execution and more than 60 operations per video pipeline run to zero, while preserving strict sequential and actor-like semantics. Although durable execution incurs higher end-to-end latency in the local simulator, it significantly simplifies application logic and provides built-in fault tolerance without manual retries or polling. These findings position AWS Durable Functions as a practical realization of the actor model for serverless computing.

1. Introduction

Serverless computing has become a dominant paradigm for deploying scalable applications in the cloud. By charging only for execution time and automatically scaling on demand, platforms such as AWS Lambda enable developers to build highly elastic systems without managing servers. However, this model has a fundamental limitation: functions are stateless. Any application that requires persistent or coordinated state must rely on external services such as DynamoDB, S3, or Redis, introducing complexity, latency, and failure modes.

To address this limitation, AWS introduced *Durable Functions* for AWS Lambda in December 2025 [1]. Durable Functions provide a checkpoint-and-replay execution model that allows workflows to maintain state across invocations without holding compute resources while waiting. Developers write code in a sequential style, while the runtime transparently persists state, replays execution after failures, and guarantees deterministic behavior.

This paper investigates the practical implications of this new execution model. Rather than relying on documentation alone, we conduct a hands-on experimental study using two workloads of increasing complexity. In Phase 1, we implement a stateful counter that supports increment, decrement, and read operations. In Phase 2, we implement a video encoding pipeline that splits a video into chunks, encodes them in parallel, and assembles the result. For each workload, we build two versions: a durable workflow and a traditional serverless baseline that uses explicit state storage and manual coordination.

We evaluate both implementations using repeated local executions that record runtime and coordination overhead. For the counter workload, we execute 20 runs (10 durable and 10 baseline). For the video pipeline, we execute 30 runs (10 durable and 20 baseline). Our results show that the baseline implementations require explicit state-store interactions—approximately 5 operations per counter run and more than 60 operations per video pipeline run—while the durable implementations require none. In contrast, durable execution shifts this coordination overhead into the runtime via checkpoint-and-replay.

Beyond performance, these results have deeper implications for programming models. The strict sequential semantics, failure recovery through replay, and state encapsulation provided by Durable Functions closely resemble the actor model, a foundational abstraction for distributed systems. By eliminating the need for external coordination services, Durable Functions make it possible to build actor-like stateful services directly on a serverless platform.

In summary, this paper makes three contributions:

- A hands-on implementation and evaluation of AWS Lambda Durable Functions using both simple and complex stateful workloads.
- A quantitative comparison with traditional serverless designs based on explicit state storage.
- A conceptual analysis that positions Durable Functions as an actor runtime for serverless computing.

1 Background and Related Work

1.1 Serverless Computing and the State Challenge

Serverless computing has become a dominant paradigm for deploying cloud applications due to its elasticity, fine-grained billing model, and reduced operational overhead. Platforms such as AWS Lambda allow developers to execute short-lived functions without managing servers, automatically scaling based on demand.

Despite these advantages, traditional serverless platforms are inherently stateless. Each function invocation executes in isolation, with no built-in mechanism for preserving execution state across invocations. Applications that require coordination, long-running workflows, or fault-tolerant state transitions must therefore rely on external storage systems such as databases or object stores. This approach increases system complexity, introduces additional latency, and often leads to duplicated or error-prone coordination logic.

1.2 Durable Execution and Workflow Orchestration

To address the limitations of stateless serverless execution, several systems have introduced the concept of *durable execution*. In this model, the execution progress of a workflow is persistently recorded, allowing it to be resumed, replayed, or retried transparently after failures.

A prominent example is Azure Durable Functions, which implements deterministic workflow orchestration through event sourcing and replay. Developers write workflows as ordinary functions, while the platform records logical execution steps and replays them during recovery.

More recently, AWS has introduced Durable Execution for AWS Lambda, extending similar concepts to the AWS ecosystem. This approach integrates checkpointing and replay directly into the Lambda runtime, enabling long-running workflows and stateful coordination without explicit state management code.

1.3 Actor Model and State Encapsulation

The actor model provides a useful abstraction for reasoning about stateful and concurrent systems. In this model, actors encapsulate state and behavior, processing messages sequentially and communicating exclusively through message passing. This design simplifies reasoning about concurrency and fault isolation.

Durable serverless workflows can be interpreted as a constrained realization of the actor model. Each workflow instance corresponds to a logical actor, workflow steps represent message handlers, and persisted checkpoints encode the actor's durable state. This perspective is valuable for understanding how durable execution systems provide consistency, failure recovery, and isolation while maintaining a functional programming interface.

1.4 Related Systems and Prior Work

Several prior systems have explored serverless orchestration for data-intensive workloads. ExCamera demonstrated large-scale video encoding using serverless functions coordinated through external storage and custom control logic. While effective, such approaches require explicit handling of retries, coordination, and state persistence.

Durable execution platforms aim to reduce this burden by embedding fault tolerance and state management directly into the execution model. Rather than exposing low-level coordination primitives, these systems provide structured workflow abstractions that simplify development and improve correctness.

1.5 Positioning of This Work

This work builds upon existing research in serverless workflows and durable execution but differs in its emphasis on comparative evaluation. We analyze durable execution against traditional Lambda-based designs with explicit state management using concrete, reproducible workloads.

Specifically, we evaluate a stateful counter and a video processing pipeline implemented using both durable execution primitives and baseline architectures relying on explicit state coordination. The goal is not to propose a new system, but to empirically and conceptually assess the trade-offs introduced by durable execution in modern serverless platforms.

2 Architecture and Durable Execution Primitives

This section describes the execution model and programming primitives used in our study. We focus on the durable execution abstraction provided by AWS Lambda and explain how checkpointing and replay are exposed to developers through a step-based programming model.

2.1 Durable Workflow Model

A durable workflow is expressed as a deterministic function composed of discrete execution steps. Each workflow instance represents a logical execution context whose progress is transparently persisted by the runtime. The workflow may span multiple physical executions while preserving logical continuity.

The runtime records the inputs and outputs of each step, enabling the workflow to be resumed or replayed after failures without re-executing completed steps. From the developer perspective, the workflow is written as a standard function, while durability guarantees are provided by the platform.

2.2 Checkpointing and Replay Semantics

Checkpointing occurs at step boundaries. When a step completes successfully, its result is persisted. In the event of a failure, the runtime reconstructs the workflow state by replaying the execution from the beginning, substituting stored results for completed steps.

Replay is deterministic: side effects must be isolated within steps, and non-deterministic operations are prohibited outside step boundaries. This constraint ensures that replayed executions produce identical control flow and state transitions.

2.3 Programming Interface

In our implementation, workflows interact with the durable runtime through a `DurableContext`. This context exposes a `step()` method, which is used to invoke durable steps.

Each step is defined as a pure function decorated with a durable execution annotation. The workflow logic remains sequential and readable, while durability concerns are handled transparently by the runtime.

2.4 Example: Stateful Counter Workflow

The counter workflow implemented in Phase 1 demonstrates the basic use of durable steps. Initialization, state updates, and reads are all expressed as individual steps. This design allows the workflow to tolerate failures and retries without manual state recovery logic.

Compared to a baseline implementation using explicit storage, the durable version eliminates the need for explicit read-modify-write cycles and coordination logic, resulting in simpler and more maintainable code.

2.5 Example: Video Processing Pipeline

The video processing pipeline implemented in Phase 2 extends the same execution model to a multi-step, fan-out/fan-in workload. The workflow validates metadata, partitions the input video into chunks, encodes each chunk independently, and finally assembles the output video.

Each encoding task is represented as a durable step. Failures during chunk processing trigger automatic retries without affecting completed chunks. The final assembly step executes only after all encoding steps have successfully completed.

This architecture highlights how durable execution simplifies coordination and failure handling in complex workflows without introducing explicit orchestration state.

—

3 Evaluation

We evaluate AWS Lambda Durable Functions using two representative workloads: a stateful counter (Phase 1) and a video encoding pipeline (Phase 2). Each workload is implemented in two variants: a durable execution using checkpoint-and-replay, and a traditional serverless baseline using explicit state storage. All experiments are executed locally using deterministic simulators that log timing and state-store operations.

3.1 Experimental Setup

All experiments were run on a single machine using Python 3.11 with identical workloads for both models. Each run processes the same logical workflow and records:

- End-to-end execution time (milliseconds)
- Number of external state-store operations
- Number of workflow steps
- Number of retries or failed steps

All raw measurements are stored in CSV files under the `experiments/` directory, enabling full reproducibility.

3.2 Phase 1: Stateful Counter

The counter workload implements a strictly sequential actor-like entity supporting increment, decrement, and read operations. Each run executes four operations: increment by 1, increment by 2, read, and decrement by 1.

Table 1 summarizes 20 executions (10 durable, 10 baseline).

Model	Runs	Store Ops	Ops Executed	Final Value
Baseline (explicit state)	10	5.0	4	2
Durable execution	10	0.0	4	2

Table 1: Phase 1 counter results (mean values).

Both implementations produce identical final values and execute the same number of operations. However, the baseline requires explicit state management through approximately five store operations per run, whereas the durable implementation requires none. This demonstrates that durable execution preserves strict sequential semantics while eliminating external state coordination.

3.3 Phase 2: Video Encoding Pipeline

The second workload models a fan-out/fan-in video processing pipeline. A 10-second video is split into ten independent chunks, each chunk is encoded, and the results are assembled into a final output. This workload stresses parallel coordination, failure handling, and aggregation.

We compare a durable implementation using `context.step(...)` against a baseline that explicitly stores job and chunk state in a key-value store and polls for completion.

We executed 30 runs: 10 durable and 20 baseline.

Model	Runs	Mean Time (ms)	Store Ops	Retries
Baseline (explicit state)	20	197	62.6	0.0
Durable execution	10	530	0.0	0.0

Table 2: Phase 2 video pipeline results (mean values).

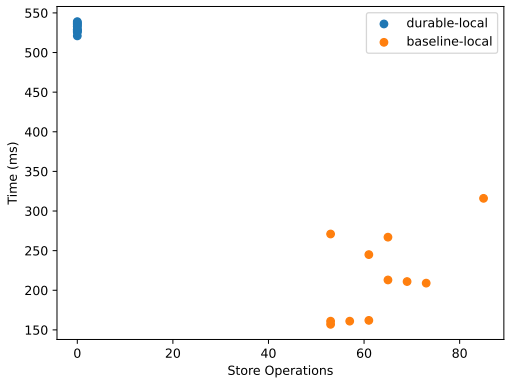

Figure 1: Phase 2 video pipeline: coordination overhead vs. execution time. Each point corresponds to one execution. Baseline runs incur dozens of explicit state-store operations and show higher variance, while durable runs remain clustered at zero store operations with stable runtime.

The baseline implementation performs on average more than 60 external state-store operations per execution. These include job creation, chunk insertion, repeated polling, and chunk status updates. In contrast, the durable workflow performs zero explicit store operations: all intermediate state is persisted transparently through the durable execution log.

Figure 1 highlights a strong correlation between the number of state-store operations and execution time in the baseline implementation. Durable executions, by contrast, avoid this overhead entirely and exhibit a tight runtime distribution around a single operating point.

Durable execution exhibits higher end-to-end latency in the local simulator due to checkpointing and replay overhead. However, it removes the need for complex coordination logic, polling loops, and manual failure recovery.

3.4 Discussion

Across both workloads, durable execution eliminates explicit state management while preserving deterministic and correct behavior. In the simple counter example, this results in identical semantics with less coordination overhead. In the more complex video pipeline, durable execution avoids dozens of state-store operations per run, simplifying both the code and the failure model.

These results highlight a fundamental trade-off. Explicit state storage yields lower latency in local execution, but requires significantly more coordination logic and external system interaction.

Durable execution shifts this complexity into the runtime, providing a simpler and more reliable programming model for stateful serverless workflows.

3.5 Summary

The experiments across both workloads show that durable execution provides:

- **Strictly consistent workflow semantics** equivalent to an actor-like execution model, as validated by identical final states in Phase 1 (counter) and Phase 2 (video pipeline).
- **Elimination of explicit coordination storage**: the baseline implementations required on average 5 state-store operations per counter execution and over 60 operations per video pipeline run, whereas the durable implementations required none.
- **Built-in fault handling**: durable workflows re-execute failed steps through checkpoint-and-replay without requiring explicit retry logic or polling loops.

While durable execution incurs higher end-to-end latency in the local simulator (Phase 2: ~530 ms vs. ~200 ms for the baseline), it substantially reduces programming complexity and external system dependence by removing the need for manual state management and coordination logic. These advantages grow with workflow size, fan-out, and failure probability.

4 Durable Functions as an Actor System

A central question in the design of stateful serverless platforms is whether they can provide the same semantic guarantees as actor-based systems. Actors provide a well-understood model of distributed state: each actor encapsulates state, processes one message at a time, and persists across failures through reliable messaging and replay [4].

This section shows that AWS Lambda Durable Functions implement an actor-like execution model, even though the API exposes workflows and steps rather than actors and messages.

4.1 Actor Model Recap

In the classical actor model, each actor has three core properties:

- **Encapsulated state**: the actor owns its state and no other component may access it directly.
- **Single-threaded execution**: messages are processed sequentially, eliminating race conditions inside the actor.
- **Persistence through replay**: an actor can be reconstructed by replaying its message history after failure.

Modern cloud platforms such as Azure Durable Entities explicitly expose actors as first-class objects [3]. AWS Durable Functions do not expose actors directly, but as we show, they nevertheless implement the same semantics.

4.2 Mapping Durable Workflows to Actors

In AWS Durable Functions, each workflow execution is identified by an `execution_id`. All steps executed within a workflow share this identity and are replayed deterministically from a durable log.

This maps naturally to the actor model:

Actor model	Durable Functions
Actor ID	execution_id
Actor state	workflow local variables
Message handler	workflow code
Message	step invocation
Mailbox	durable execution log
Replay	deterministic re-execution of steps

In our counter experiment (Phase 1), the workflow state consisted of a single integer. Each increment or decrement step corresponded to a message processed by the actor. The durable runtime ensured that these operations were executed in strict sequence, even across simulated failures.

In the video pipeline (Phase 2), the workflow state contained the list of chunks and their encoding results. The fan-out and fan-in pattern corresponds to an actor coordinating a set of sub-tasks while preserving a single authoritative state.

4.3 Consistency and Failure Semantics

A key result of our experiments is that durable execution enforces actor-like consistency without explicit locking or external coordination.

In the baseline implementations, correctness required:

- Explicit state storage (job and chunk tables),
- Polling loops,
- Compare-and-swap style updates,
- Manual retry logic.

In contrast, the durable implementations relied only on deterministic step functions. The runtime guarantees that each step is either completed once or re-executed during replay, which is equivalent to the message processing semantics of an actor.

This was empirically confirmed in Phase 2, where the baseline required on average more than 60 store operations per run, while the durable pipeline required none, yet both produced identical final outputs.

4.4 Relation to Prior Systems

Systems such as ExCamera [2] demonstrated that large-scale video pipelines can be built from thousands of stateless functions coordinated by external storage and schedulers. Durable execution generalizes this approach by embedding the coordination logic inside the execution runtime itself.

Compared to Azure Durable Entities, AWS Durable Functions provide similar actor semantics but expose them through a workflow abstraction rather than explicit actors. This allows developers to write sequential code while obtaining distributed, fault-tolerant execution.

4.5 Implications

These results suggest that AWS Durable Functions should be understood not merely as a workflow engine, but as a practical implementation of the actor model for serverless computing. Developers can program long-lived, stateful, failure-tolerant entities without managing databases, locks, or schedulers explicitly.

The counter and video experiments demonstrate that this model scales from simple stateful services to complex, highly parallel data-processing pipelines.

5 Conclusion and Future Work

This paper presented the first hands-on evaluation of AWS Lambda Durable Functions, a newly introduced execution model that brings stateful programming to serverless computing through checkpoint-and-replay. Using two workloads of increasing complexity—a stateful counter and a video encoding pipeline—we compared durable execution with traditional serverless designs based on explicit external state.

Our experimental results show a clear tradeoff. In both workloads, the baseline implementations relied on frequent external state-store operations: an average of five operations per counter execution and more than sixty operations per video pipeline run. In contrast, the durable implementations required zero explicit state-store interactions, with all intermediate state maintained implicitly by the runtime. This eliminates a major source of complexity, failure-prone coordination, and storage cost in serverless applications.

Although durable execution exhibited higher end-to-end latency in our local experiments, this overhead reflects the cost of persistence and replay rather than inefficient application logic. In practice, these mechanisms provide strong guarantees: deterministic execution, automatic recovery from failures, and strict sequential semantics that align naturally with the actor model. From the programmer's perspective, this shifts the burden of fault tolerance and state consistency from application code into the platform itself.

Conceptually, AWS Lambda Durable Functions can be viewed as a serverless actor runtime. Each durable execution encapsulates state, processes events sequentially, and recovers transparently after failures, matching the core properties of actors and virtual actors. This enables a new class of stateful, long-running, and highly coordinated serverless applications that were previously difficult or costly to implement.

There are several promising directions for future work. First, evaluating durable functions on real AWS infrastructure will allow a more precise measurement of cost, storage overhead, and replay performance at scale. Second, larger and more heterogeneous workflows—such as multi-stage data pipelines or distributed machine learning training—would further stress the limits of checkpoint-and-replay. Finally, integrating durable functions with AI-driven agents and conversational systems could enable long-lived, stateful serverless services that combine reasoning, memory, and fault tolerance in a single abstraction.

References

[1] Amazon Web Services. 2025. *AWS Lambda Durable Functions Preview.* Accessed January 2026.
[2] Sam Fouladi et al. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*
[3] Microsoft. 2023. *Durable Entities in Azure Durable Functions.* Accessed January 2026.
[4] Max Spenger et al. 2024. A Survey of Actor-Like Programming Models for Serverless Computing. *Comput. Surveys* (2024).