

Article

Data-Centric Serverless Computing with LAMBDASTORE

Kai Mast ¹, Suyan Qu ², Aditya Jain ², Andrea Arpaci-Dusseau ² and Remzi Arpaci-Dusseau ^{2,*}¹ Provable Inc., 5470 Kietzke Ln Ste 300, Reno, NV 89511, USA² Department of Computer Sciences, University of Wisconsin-Madison, 1205 University Avenue, Madison, WI 53706, USA

* Correspondence: remzi@cs.wisc.edu

[^] Joint senior authors

Abstract

LAMBDASTORE is a data-centric serverless platform that breaks the split between stateless functions and external storage in classic cloud computing platforms. By scheduling serverless invocations near data instead of pulling data to compute, LAMBDASTORE substantially reduces the state access cost that dominates today's serverless workloads. Leveraging its transactional storage engine, LAMBDASTORE delivers serializable guarantees and exactly-once semantics across chains of lambda invocations—a capability missing in current Function-as-a-Service offerings. We make three key contributions: (1) an object-oriented programming model that ties function invocations with its data; (2) a transaction layer with adaptive lock granularity and an optimistic concurrency control protocol designed for serverless workloads to keep contention low while preserving serializability; and (3) an elastic storage system that preserves the elasticity of the serverless paradigm while lambda functions run close to their data. Under read-heavy workloads, LAMBDASTORE lifts throughput by orders of magnitude over existing serverless platforms while holding end-to-end latency below 20 ms.

Keywords: serverless; cloud computing; Function-as-a-Service; storage systems; transactions

1. Introduction

Serverless computing promises rapid development and deployment of applications and services [1,2]. Instead of purchasing and managing backend infrastructure, service providers can focus on developing application logic, while cloud providers handle resource provisioning, service deployment, hardware fault tolerance, and etc. Additionally, serverless platforms offer high availability and automatic scaling, enhancing service robustness. The serverless market is already substantial—nearly \$8 billion—and is projected to exceed \$20 billion by the end of 2026 [3]. As one CEO said, “Serverless has demonstrated that it is the operational model of the future” [4].

Existing serverless platforms largely follow a *disaggregated* approach [5–13]. For example, consider AWS Lambda, the most popular serverless platform today. To deploy a service, the application developers upload custom executables or scripts to AWS Lambda, which then execute them in virtualized environments upon user invocations [14]. However, these virtualized environments lack mechanisms to store data persistently. To reliably store application data for the desired functionality, developers must rely on external storage services such as Amazon S3 [15].

The separation of compute and storage enables independent scaling of each component and works well for certain non-interactive applications. However, service providers quickly

Academic Editors: Claus Pahl,
Manuel Mazzara and Mirko Viroli

Received: 1 November 2025

Revised: 20 December 2025

Accepted: 12 January 2026

Published: 21 January 2026

Copyright: © 2026 by the authors.
Licensee MDPI, Basel, Switzerland.
This article is an open access article
distributed under the terms and
conditions of the [Creative Commons
Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

encounter challenges when building more complex, data-intensive applications [16,17]. Recent surveys have found that many serverless functions run for only a few seconds, are invoked infrequently, and operate on datasets smaller than 10 Mb [18,19]. These use cases highlight the limitations of current platforms: serverless execution in its current form suffers from degraded performance under these workloads due to high latency interacting with external storage systems [20].

In addition, most serverless platforms provide *at-least-once* semantics and lack transactional guarantees. The serverless functions may execute more than once on a single request, and partial results of an ongoing execution may be observed by other function calls. Consequently, serverless platforms often require functions to be idempotent to accommodate these semantics [21], and they place the burden of concurrency control on the application itself if needed—directly contradicting one of the core tenets of serverless systems: reducing complexity for the application developer.

To address these challenges, we identify six desirable properties for modern serverless platforms, which we abbreviate as *RAISED*: Responsiveness, Atomicity, Isolation granularity, Serializable workflows, Elasticity, and Durable storage. Based on these properties, we introduce LAMBDASTORE: a serverless execution system with an integrated storage engine designed for low-latency cloud applications. In LAMBDASTORE, serverless functions execute close to their persisted data, reducing storage access cost, while the storage engine provides transactional guarantees for the function executions. Such a co-design aligns with the end-to-end argument [22]: rather than enforcing consistency and fault tolerance separately at the storage and compute layers, a unified architecture significantly reduces the overhead associated with replication and concurrency control.

The design of LAMBDASTORE addresses three key challenges: identifying the affinity between serverless functions and application data to enable compute–storage colocation, efficiently supporting transactional workflows, enabling colocation without sacrificing the elasticity of the stateless model. To discover opportunities for colocation, LAMBDASTORE introduces LAMBDAOBJECTS, an object-oriented data model that associates functions with their application data (Section 3). To support transactional workflows, LAMBDASTORE embeds a transactional key-value store and encapsulates each workflow within its own transaction (Section 4.3). This storage layer is tailored for serverless workloads, minimizing transaction conflict rates. To preserve the scalability and elasticity essential in cloud environments, LAMBDASTORE leverages two mechanisms: light replication (Section 4.4.2) and object migration (Section 4.4.3). It employs microsharding (Section 4.4.1) to efficiently migrate or replicate individual objects, enabling rapid adaptation to changing workloads.

LAMBDASTORE outperforms the conventional disaggregated serverless design in both throughput and latency while providing stronger guarantees. We compare it against OpenWhisk [5], OpenLambda [23], Faasm [11], and Apiary [24] using microbenchmarks to demonstrate its performance benefits. Our experimental results show that, compared with other serverless systems based on WebAssembly, LAMBDASTORE is able to deliver orders of magnitude higher throughput while maintaining a low end-to-end latency for stateful workloads. We also evaluate the systems with two applications—an online message board and a Twitter-like microblog—to demonstrate its scalability. We observe that LAMBDASTORE scales almost linearly with the number of nodes. With a cluster of 36 machines, it can process up to 600 k transactions per second.

In this work, we make the following contributions:

- We identify six desirable properties for modern serverless platforms, summarized as *RAISED*: Responsiveness, Atomicity, Isolation granularity, Serializable workflows, Elasticity, and Durable storage.

- We present the LAMBDAOBJECTS abstraction that associates serverless functions with their application data to enable compute–storage colocation.
- We present the design and implementation of LAMBDASTORE, a serverless execution system with an integrated storage layer designed for serverless workloads. With the LAMBDAOBJECTS abstraction, LAMBDASTORE achieves RAISED properties.
- We evaluate LAMBDASTORE with microbenchmarks and application workloads, demonstrating its performance and scalability benefits over existing serverless platforms.

The rest of the paper is organized as follows. We first provide background on serverless computing and introduce RAISED properties (Section 2). We then explain the LAMBDAOBJECTS abstraction with a sample application (Section 3). Following, we present the design and implementation of LAMBDASTORE (Section 4). We evaluate LAMBDASTORE with microbenchmarks and application workloads to demonstrate the effectiveness of our approach (Section 5). Next, we discuss limitations in the current prototype and their mitigations as future directions (Section 6). Finally, we discuss related works (Section 7) and conclude (Section 8).

2. Background and Motivation

Serverless platforms allow mutually distrusting applications to share cloud infrastructure through *virtualization* and *dynamic resource assignment*. Virtualization isolates individual running serverless function instances (or serverless *jobs*) from one another, ensuring that jobs fail independently and do not compete for resources. This isolation provides fault resilience, allowing multiple applications to safely colocate on the same hardware. Dynamic resource allocation, in turn, enables the (re-)allocation of resources to serverless jobs at runtime. It improves resource efficiency and supports scalability by adjusting resource allocations based on each application’s current workload.

Serverless systems provide an abstraction that allows developers to build and deploy services without provisioning or managing machines. In this model, applications are implemented as a set of functions, and a cloud provider (e.g., AWS Lambda, Microsoft Azure Functions, Google Cloud Functions) manages their instantiation and execution. Each job can invoke other serverless functions, forming a serverless *workflow*. In some system designs, each job performs a small task, and workflows consist of many such jobs [25]. However, the complexity and runtime of a job can vary depending on the application.

RAISED

Our goal is to design a cloud computing infrastructure suitable for stateful applications. After evaluating existing platforms, we identified six desirable properties for modern cloud computing systems, which we abbreviate as **RAISED**:

- **Responsiveness**: Serverless applications should be able to respond to user requests quickly, even during cold starts.
- **Atomicity**: The cloud platform should guarantee *exactly-once* execution semantics to simplify application logic.
- **Isolation **G**ranularity**: The platform should offer fine-grained isolation to ensure that jobs fail independently.
- **Serializable **W**orkflows**: For applications requiring strong consistency, the platform should ensure that workflows are serializable and that applications always observe a consistent state.
- **Elasticity**: The platform should dynamically adjust resource allocations based on current workloads.
- **Durable **S**torage**: The platform must be able to reliably persist application data.

A conventional stateful serverless system architecture typically consists of three components: a coordination layer, a compute layer, and a storage layer. We describe this architecture using OpenWhisk [5] as a representative example. OpenWhisk relies on Apache Kafka [26] to track outstanding jobs, which are then delegated to a compute layer (e.g., Kubernetes [27]). Serverless applications interact with a dedicated storage layer, such as a DBMS or key-value store, to persist state across function invocations.

This disaggregated compute–storage architecture enables high elasticity, as scaling a stateless compute layer is straightforward. However, it also incurs frequent data transmissions, leading to high latencies and reduced responsiveness. For example, in a simple read-modify-write workload, a job must first fetch data from the remote storage system, perform the update, and then write the result back. While the update itself may take only a few microseconds, the data fetch and store operations dominate latency. This problem is worsened during cold starts, which require establishing new TCP connections between the compute and storage layers, and by the high software overhead imposed by the storage system on every data access.

Furthermore, because most serverless platforms do not guarantee exactly-once execution semantics, they cannot provide transactional guarantees at the workflow level—even if the storage layer supports transactions. This is due to the lack of coordination: serverless functions must explicitly commit transactions. If a transaction is committed but the workflow fails and is retried, the ACID properties can be violated. As a result, conventional cloud computing platforms, including OpenWhisk, OpenLambda [23], AWS Lambda [10], Google Cloud Functions [6], and Azure Functions [9], suffer from these limitations.

Some research efforts [7,8,11–13,28–32] seek to reduce state access latency by caching persistent state or storing intermediate state locally. Jobs within a workflow that share state can benefit significantly from this approach, as the platform can improve data locality by colocating them on the same machine, avoiding repeated data transmission over the network. Faasm, in particular, further improves responsiveness by adopting *software-fault isolation* (SFI) via *WebAssembly* to reduce cold-start latency. However, this compute-centric design still relies on remote storage for data persistence. Caching does not improve cold-start performance, since the state is not yet in cache, and it introduces complexity in maintaining consistency across the storage layer.

Other research efforts [24,33–35] propose shipping serverless jobs (or subunits of them) to the storage layer to enable colocation. Apiary, for example, leverages the user defined functions (UDFs) in database systems to achieve colocation through query optimization. This approach not only reduces the end-to-end latency, improving responsiveness, but also ensures serializability and atomicity. Nonetheless, a naïve colocation design will restrict the elasticity of serverless systems: forcing jobs to run near the data constrains the amount of compute resources they can utilize, as the resources available near the data are limited by the capacity of the physical machines storing that data. Apiary’s approach additionally introduces problems with fault isolation: UDF failures such as out-of-memory and segmentation faults can crash the local database. Stored procedures expose a huge attack surface as they are not designed to run untrusted code from end users.

Table 1 summarizes whether different systems meet each of the RAISED properties. While existing serverless platforms and research prototypes have made significant strides in improving elasticity, responsiveness, and transactional support, they often face trade-offs between colocation and scalability, or between performance and durability. These limitations highlight the need for a new system architecture that unifies compute and storage without sacrificing the core benefits of serverless computing. In this work, we present LAMBDASTORE, a serverless execution platform designed to meet the RAISED

properties by tightly integrating compute and storage, enabling transactional, low-latency, and scalable execution for stateful cloud applications.

Table 1. Comparison of LAMBDASTORE to other serverless architectures. We report only cold start time as a measure of responsiveness, since end-to-end latency is highly dependent on application logic.

	<u>Responsiveness</u> (Cold-Start)	<u>Atomicity</u>	<u>Isolation</u> <u>Granularity</u>	<u>Serializable</u> <u>Workflows</u>	<u>Elasticity</u>	<u>Durable</u> <u>Storage</u>
OpenWhisk [5]	>100 ms	No	Job	No	Yes	No
OpenLambda [23]	<1 ms (WASM) >10 ms (SOCK)	No	Job	No	Yes	No
Faasm [11]	<1 ms	No	Job	No	Yes	Yes
Apiary [24]	<1 ms	No	Node	Yes	No	Yes
AWS Lambda [10]	>10 ms	No	Job	No	Yes	Yes
Google Cloud Functions [6]	>100 ms	No	Job	No	Yes	Yes
Azure Functions [9]	>100 ms	No	Job	No	Yes	Yes
LAMBDASTORE (This paper)	<1 ms	Yes	Job	Yes	Yes	Yes

3. The LAMBDAOBJECTS Abstraction

LAMBDASTORE is a cloud computing system built on a compute–storage co-design. It organizes both data and execution around stateful objects, which we refer to as LAMBDAOBJECTS. This section introduces the LAMBDAOBJECTS abstraction and illustrates its effectiveness through a concrete example.

3.1. Data Model

3.1.1. Objects

Applications in LAMBDASTORE are defined through *Object Types*, which specify both the executable code (*functions*) and the data entries (*fields*) that each object contains. *Objects* can then be instantiated from these types, similar to the class-object relationship in object-oriented programming—a paradigm familiar to many developers. By encapsulating both logic and data within objects, LAMBDASTORE provides an intuitive way to decompose applications into small, independent components, akin to the structure of microservices [36]. Moreover, LAMBDASTORE leverages this object abstraction to determine data placement and to colocate function execution with the corresponding data. The object model is designed to be highly flexible, supporting multiple programming languages, variable object sizes, and application-specific data types.

3.1.2. Applications

Each object belongs to an application. The application developer is billed for both the storage of the application’s data and the execution of its functions. Developers define which object functions are part of the application’s public API and which are restricted to internal use. Currently, objects can invoke functions only within the same application. We reserve sharing data and code across application boundaries and more fine-grained access control for future work.

3.1.3. Object Entries

Entries form the smallest unit of data in LAMBDASTORE and are stored as (part of) a field within a specific object. Data fields then define how entries are accessed, stored, and indexed for lookup. LAMBDASTORE currently supports three field types: maps, multi-maps,

and cells (unstructured data). Other data types can be implemented in application code. For example, a set can be built on top of the map primitive.

Each entry is a key-value pair that is stored and replicated by LAMBDASTORE and is associated with a specific data field of an object. For instance, an entry might represent a single item in an object's map or the full content of a cell. The content and semantics of each entry are application-specific and opaque to LAMBDASTORE. Functions interact with entries through a minimal API that supports reading, writing, and basic range queries. Application code is responsible for (de-)serialization of data and to provide more complex data operations (e.g., increment or append).

3.2. Execution Model

3.2.1. Function Calls

Application logic in LAMBDASTORE executes in the form of function calls (or jobs). A job represents the invocation of a specific function. As in conventional serverless systems, more complex application logic can be constructed by composing multiple jobs into a workflow. The structure of the job graph within a workflow is not predefined; instead, it is generated dynamically as functions execute.

Functions in LAMBDASTORE come in several variants, similar to those in object-oriented programming. *Constructors* are used to create and initialize new objects. *Methods* operate on existing objects and can access or modify their fields. *Static functions* are not associated with any specific object and hence do not have direct access to object data.

In addition, LAMBDASTORE provides a mechanism for operations that involve many objects: *map calls*. For example, consider the task of finding the maximum age among all clients in an application, where each client is represented by a dedicated object. Executing a separate function on each individual object would be highly inefficient. Instead, a map call takes a set of objects O and a function f , and executes at most one instance of f per shard. Each invocation of f then iterates over the subset of O that is located on that shard. This mechanism enables efficient batch processing without incurring additional data movement. The results of the map call are aggregated and returned to the caller.

Function calls have direct access only to the data of the object(s) they are associated with. To access or modify data in other objects, functions must either invoke constructors or methods on those objects or use map calls. This pattern minimizes data movement.

Functions in LAMBDASTORE are exposed to a minimal API upon which more complex application logic can be built. This API allows functions to read and write object entries, invoke other functions, manage user sessions, retrieve function arguments, set return values, obtain the current time, and generate randomness. Similar to system calls in an operating system, which only provide low-level functionality, higher-level abstractions are then implemented in user space (or in our context, within the serverless functions).

Functions execute until they either complete successfully or are terminated by the runtime. Termination can occur due to fatal errors (e.g., stack overflows), violations of security policies (e.g., attempting to access data outside the application's scope), or exceeding the maximum allowed execution time.

3.2.2. Transactions

LAMBDASTORE guarantees strict serializability across the entire workflow. A workflow can be represented as a directed acyclic graph (DAG)—more specifically, as a directed rooted tree—of function calls. In this graph, each vertex corresponds to a job, and each edge represents a function call that spawns another job. Workflows are initiated by clients invoking a public function, which forms the root of the DAG. As functions call other functions, the DAG is dynamically extended at runtime. These execution graphs are not

predefined by the application developer but are constructed automatically during execution. Developers can inspect a workflow's structure by examining its *execution trace*.

Currently, individual function calls in LAMBDASTORE execute sequentially, but workflows can execute multiple jobs concurrently. LAMBDASTORE allows a function to issue multiple function calls simultaneously and wait for all of them to complete. For example, in a social network application, a function that creates a user's post might invoke separate functions for each follower to update their respective timelines. While the post creation itself executes sequentially, the timeline updates can proceed in parallel.

Each workflow in LAMBDAOBJECTS is contained within a single LAMBDASTORE transaction. Throughout this paper, we use the terms *workflow* and *transaction* interchangeably when referring to LAMBDASTORE.

3.3. Example Application

LAMBDAOBJECTS support arbitrary applications. To demonstrate its flexibility, we present an example application: CLOUDFORUM, an online discussion board similar to Reddit. This section outlines part of its implementation in Rust, with some simplifications made for clarity and space constraints.

Listing 1 shows how object types are declared within an application. In this example, we outline three types. Accounts represent users of the online forum and contain references to all threads and comments they have created. Threads store the initial post by the thread creator, along with a sequence of comments. Comments are stored using a custom structure called *Comment*, which is not an object type itself but defines how data within the *comments* field is structured. Each comment can be identified through the thread ID and its index within the thread. Finally, threads are indexed using the *Community* type.

Listing 1. Object types for an online forum application.

```
#[lambda_object] struct Account {
    name: Cell<String>,
    threads: Set<ObjectId>,
    comments: Set<(ObjectId, u32)>,
}

#[lambda_object] struct Community {
    by_name: MultiMap<String, ObjectId>,
    by_time: MultiMap<u64, IndexEntry>,
}

#[lambda_object] struct Thread {
    author_name: Cell<String>,
    community_id: Cell<ObjectId>,
    author_id: Cell<ObjectId>,
    title: Cell<String>,
    text: Cell<String>,
    comment_count: Cell<u32>,
    comments: Map<u32, Comment>,
}
```

Listing 2 illustrates the workflow for adding a new comment to an existing thread. The implementation uses LAMBDASTORE's Rust bindings, which abstract away much of the boilerplate, such as data serialization. Users initiate the workflow by invoking the *create_thread* method on an object of type *Account*. Arguments, such as the content of the comments, are passed in a JSON document.

The function first authenticates the user (omitted for brevity) and looks up the user's identifier and name. It then calls the *add_comment* method on the specific thread where the comment should be added. This method executes in a separate, dedicated job. By default, invoked function calls run in the background unless the parent job explicitly calls *join*, as shown in the example. This API enables concurrency similar to *async/await* pattern found in many programming languages: multiple child jobs can be spawned and waited on simultaneously.

The `add_comment` method then stores the comment as part of the `Thread` object. It attaches a timestamp to the comment using the `get_unix_time` host call. Finally, it returns the comment's identifier, which the `Account` object stores in its `comments` field.

Listing 2. Implementation of CLOUDFORUM's comment functionality. Low-level API calls are abstracted behind a higher-level interface. Access control and error handling logic are omitted for brevity.

```
#[lambda_functions] impl Account {
  fn create_comment(&self,
    app: Application, args: json::Value) {
    args.set("author_id", self.get_identifier());
    args.set("author_name", self.name.get());

    let thread_id = args.get("thread_id");
    let result = app.get_object()
      .call_json("add_comment", &args)
      .join();

    // Store a reference to the comment
    let comment_id = result.get("comment_id");
    self.comments()
      .insert(&(thread_id, comment_id));
  }
}

#[lambda_functions] impl Thread {
  #[protected]
  fn add_comment(&self, args: json::Value) {
    let comment = Comment {
      author_id: args.get("author_id"),
      author_name: args.get("author_name"),
      text: args.get("text"),
      time: get_unix_time(),
    };

    // Increase the total comment count
    let comment_cnt = self.comment_count().get();
    let comment_id: u32 = comment_cnt + 1;
    self.comment_count().put(&comment_id);

    // Store the comment in the thread's object
    self.comments().put(&comment_id, &comment);
    set_json_result({"comment_id": comment_id});
  }
}
```

4. LAMBDASTORE

This section presents the design of LAMBDASTORE in detail. We begin by outlining the overall system architecture. Next, we discuss the virtualization technique employed by LAMBDASTORE. We then describe how LAMBDASTORE efficiently provides transactional guarantees for workflows and how it enables compute-storage colocation while preserving elasticity. Finally, we describe fault tolerance and recovery mechanisms.

4.1. System Architecture

LAMBDASTORE consists of four types of participants: clients, frontends, worker nodes, and coordinating nodes. Figure 1 sketches how these components interact within the system.

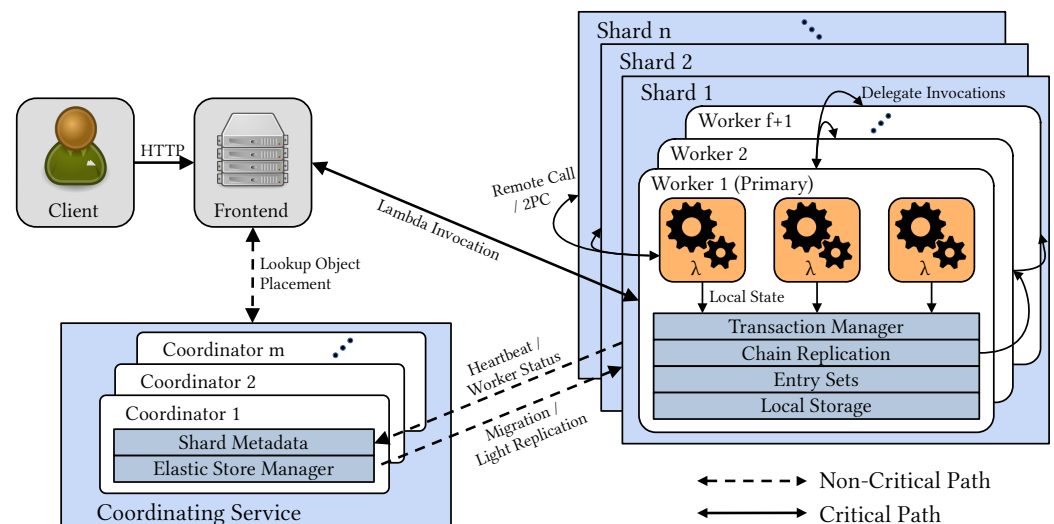


Figure 1. LAMBDASTORE employs a shared-nothing architecture. The Lambda requests are sent directly to the primary worker, which will launch serverless runtimes to execute the request locally or delegate to a secondary replica. Each worker has a Transaction Manager (Section 4.3.2), a Chain Replication module, Entry Sets (Section 4.3.1), and a local storage engine (LevelDB [37]). LAMBDASTORE has a centralized coordinating service that manages shard metadata such as object placement and shard membership. It monitors worker status and handles reconfiguration. It also makes object migration (Section 4.4.3) and light replication (Section 4.4.2) decisions to provide elasticity. Notably, the coordinating service does not participate during most lambda invocations.

4.1.1. Worker Nodes and Replica Sets

Worker nodes execute serverless jobs and store LAMBDAOBJECTS. They are organized into *replica sets* of $f + 1$ nodes. Application data are sharded, and each shard is assigned to a replica set. Nodes within a replica set run replication protocols to ensure data consistency and durability.

Requests for LAMBDAOBJECTS methods are first sent to the *primary* node of the replica set that stores the associated object. The primary determines where the method should be executed. Under light workloads, it is most efficient to execute the job directly on the primary. Under heavy workloads, however, the primary can delegate the job to other nodes in the replica set to avoid becoming a bottleneck.

Within each replica set, LAMBDASTORE uses chain replication [38] to replicate data across nodes. As the name suggests, nodes in the set are arranged in a chain, with the head serving as the primary and the remaining nodes acting as *secondary* replicas. During state changes, the primary communicates the update to only one secondary replica, which then propagates it down the chain. With a replica set of $f + 1$ nodes, LAMBDASTORE can tolerate up to f simultaneous node failures without losing data. Compared with other replication algorithms, chain replication significantly reduces the load on the primary, enabling high scalability and elasticity even under skewed workloads.

4.1.2. The Coordinating Service

The coordinating service (or *coordinator*) is responsible for maintaining all metadata. It tracks all participants in the cluster, the configuration of individual shards, and the placement of objects within those shards. When worker nodes, frontends, or clients join the network, they first connect to the coordinating service, which informs them about other worker nodes in the system and the current object placements. In addition, the coordinating service monitors the health and workload status of each worker. It can initiate object migration or create light replicas accordingly to provide elasticity. In our prototype,

we only demonstrate the effectiveness of object migration and light replication with manual triggers. We leave automatic elastic store management policies for future exploration.

For reliability, the coordinating service should ideally use a distributed metadata management system such as ZooKeeper [39]. In our prototype, however, the coordinator is not replicated for simplicity. We argue that this does not significantly impact system performance, as the coordinating service is not involved in most requests. Transactions only reach the coordinator if they create new objects or when there is an ongoing reconfiguration of the cluster. The former can be optimized: workers can asynchronously notify the coordinator of new objects after the transaction commits, and the coordinators can fetch new object placements from the workers when needed. The latter only occurs when the system performs an elastic storage decision (Section 4.4) or during failures (Section 4.5).

4.1.3. Clients and Frontends

To avoid accessing the coordinator on every request, clients maintain local caches of system configuration and object placements. However, a large number of short-lived clients can easily overwhelm the coordinating service. To mitigate this issue, LAMBDASTORE introduces *Frontends*, which maintain up-to-date caches of the cluster configuration. Clients send their requests to a Frontend, which then redirects the request to the appropriate primary worker node based on its cached view.

4.2. Virtualization Layer

LAMBDASTORE enables the execution of untrusted computation through the use of WebAssembly (or WASM) [40]. We chose WebAssembly as the virtualization mechanism due to its significantly lower overhead compared to alternatives such as virtual machines (VMs) or containers. Once an application developer compiles their code into WebAssembly, it is registered with the coordinator. The coordinator then compiles the WebAssembly instructions to machine code, injects additional safeguards to protect against misbehaving programs, and distributes the resulting code to all storage nodes.

Upon invocation, nodes directly embed the generated code into their address space. To start a job, a node allocates memory using `mmap` and performs a *context switch* by saving register contents and setting the program counter to a location within the function's code. The function interacts with the host environment via a predefined set of API calls, each of which triggers a context switch back to the storage layer, analogous to how system calls transition from user space to kernel space.

Threat Model

As a serverless platform, LAMBDASTORE must execute untrusted code provided by application developers. WebAssembly offers software-based fault isolation by enforcing memory safety through mechanisms such as bounds checking on memory accesses and control-flow integrity. It also prevents memory leakage between different functions with zero-initialized memory and protects the kernel by disabling system calls.

WebAssembly, however, does not protect against non-terminating functions. LAMBDASTORE uses periodic timer interrupts to abort indefinitely running lambda jobs. When an interrupt occurs, a trap handler checks whether the current function has exceeded its maximum execution time and aborts it if necessary. Tracking function execution time is also essential for implementing an accurate billing model in a cloud environment.

In addition, WebAssembly does not provide resource isolation, and LAMBDASTORE is prone to many side-channel attacks. We elaborate on providing resource isolation in Section 6.

4.3. Serverless Transactions

The storage layer in LAMBDASTORE provides a transactional interface. LAMBDASTORE enforces atomicity and strict serializability by encapsulating all data accesses within a workflow into a single transaction. The transaction is committed when the workflow completes. If the commit fails, LAMBDASTORE can re-execute the workflow to ensure that its effects are externalized exactly once. This guarantee holds under the assumption that workflows do not produce side effects on external services outside of LAMBDASTORE's control. Supporting exactly-once semantics for functions that interact with external services remains an open challenge and is left to future work.

Providing transactional guarantees efficiently under serverless workloads is challenging. Since developers can define arbitrary application logic, workflows may vary significantly in duration and read-write patterns, ranging from short-running to long-running executions. Running such workflows concurrently can result in high conflict rates and, in some cases, starvation. Additionally, the elastic nature of serverless platforms allows a large number of workflows to execute simultaneously under heavy workloads, further exacerbating contention. LAMBDASTORE addresses these challenges by dynamically adjusting lock granularity through entry sets and by employing a variant of Silo's optimistic concurrency control protocol [41] with two phase commit (2PC).

4.3.1. Entry Sets

LAMBDASTORE tracks metadata (e.g., locks and version numbers) for each object to support concurrency control. A naïve approach might maintain a single set of metadata per object, which can be too coarse-grained for large objects, or one set per entry, which introduces excessive bookkeeping overhead. Instead, LAMBDASTORE uses *entry sets* as the unit of locking and version control. Each object defines a series of *guards* that determine the boundaries between entry sets. Figure 2 illustrates an example of such a partitioning. Entry sets allow LAMBDASTORE to reduce transaction conflicts by dynamically adjusting the lock granularity. Rather than locking entire objects, transactions only acquire locks on the entry sets they access. In the example shown in Figure 2, one transaction can update the account name while another concurrently creates new threads without conflict.

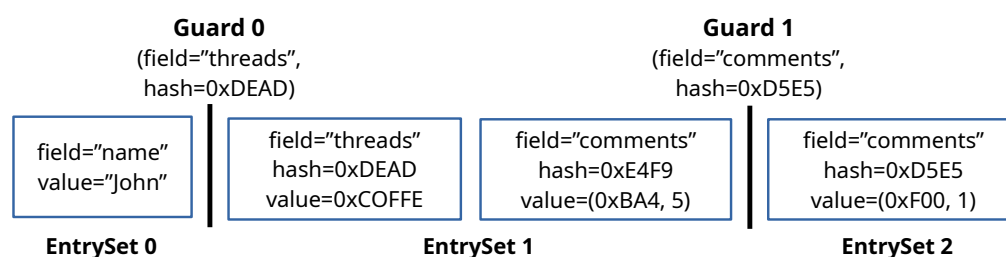


Figure 2. Visualization of an Account object's storage layout. The keyspace is partitioned into three entry sets. Entry sets can span multiple fields (e.g., EntrySet 1), and entries in the same field may belong to different entry sets (e.g., "comments").

In LAMBDASTORE, objects initially consist of a single entry set. With some probability, a write to a key will insert a new guard at that key's position, effectively splitting the entry set in half. The probability of inserting a guard increases for entry sets with higher lock contention. As a result, objects that are written to more frequently or experience more conflicts are more likely to accumulate more guards and be partitioned into more entry sets. The intuition behind this design is that only write-heavy or highly contended workloads benefit from finer lock granularity. A similar approach has proven effective in the context of Log-Structured Merge Trees [42]. Because entry sets can only be split during writes,

splitting occurs exclusively during transaction commits, when the entry set is already write-locked and inaccessible to other transactions. After the split, the resulting entry sets are assigned version numbers higher than that of the original set. This mechanism ensures that the concurrency control protocol detects and handles the change appropriately. Consequently, entry set splitting does not interfere with function execution or violate system consistency.

4.3.2. Concurrency Control

Because serverless workflows can run for arbitrary durations, using pessimistic locking may lead to starvation, where a single long-running workflow blocks all other concurrent ones. To avoid this problem, LAMBDASTORE uses a variant of Silo's optimistic concurrency control protocol [41] to enforce ACID properties within each shard. Algorithm 1 outlines the protocol. During the execution phase, instead of atomically reading both the value and its version number, LAMBDASTORE leverages the atomic interface of its key-value storage backend to allow transactions to read a slightly stale version number. Specifically, during reads, each transaction first atomically reads the version number of the corresponding entry set, then queries the storage backend for the value. During the commit phase, the transaction must write each updated value back and then atomically update its version number. This design enables reads to proceed without acquiring locks during the execution and prepare phases. We provide a correctness proof in Appendix A.1. The core intuition is that if a transaction reads a mismatched version and value during execution, then either (1) the version number it reads during the prepare phase will differ from the one read during execution, or (2) another concurrent transaction will be updating the entry set, which will thereby be write-locked. In both cases, the transaction will abort, ensuring serializability. Within each replica set, secondary nodes can execute jobs and read from their local storage during the execution phase. However, for each workflow, the secondary must send its read set and write set to the primary node, which is responsible for handling the prepare and commit phases.

Algorithm 1: The Transaction Protocol of LAMBDASTORE

```

Fn read(txn, key):
    version ← get_version(key)
    value ← get(key)
    txn.read_set.insert(key, version)
    return value
Fn write(txn, key, value):
    txn.write_set.insert(key, value)
Fn prepare(txn):
    foreach key, value in txn.write_set do
        lock(key)
    foreach key, version in txn.read_set do
        if version ≠ get_version(key) or is_locked(key) then
            return ABORT
    return COMMIT
Fn commit(txn):
    foreach key, value in txn.write_set do
        put(key, value)
        update_version(key)
        unlock(key)

```

LAMBDASTORE supports cross-shard transactions by coordinating its concurrency control protocol with two-phase commit (2PC). The worker node that initiates the workflow

acts as the *transaction manager* and is responsible for managing the 2PC process. When a workflow begins, it is associated with a transaction. Jobs spawned by other jobs inherit the transaction context of their caller. Upon completion, each job returns its output to the calling job, along with metadata about the transaction including the identifiers of any additional worker nodes involved and whether any new objects were created. This information is propagated recursively until the initial job (the root of the workflow's DAG) completes. At that point, the transaction manager begins the 2PC protocol by signaling all involved nodes to enter the prepare phase and report their results. If all nodes agree to commit, the transaction manager then sends a final signal instructing all nodes to commit the transaction.

4.4. Elasticity Storage Service

To colocate compute with storage while preserving the elasticity of serverless platforms, the storage layer must be able to adapt to changing workloads. LAMBDASTORE achieves elasticity through two mechanisms: *object migration* and *light replication*. To minimize the overhead associated with both, LAMBDASTORE employs *microsharding*, allowing the system to respond quickly to workload fluctuations.

4.4.1. Microsharding

LAMBDASTORE treats objects as *microshards* [43], ensuring that each object and all of its associated data are mapped to a single shard. This fine-grained approach enables LAMBDASTORE to quickly respond to workload changes by migrating or creating lightweight replicas of individual objects. While the exact placement policy is beyond the scope of this paper, LAMBDASTORE generally aims to map objects belonging to the same application to as few replica sets as possible in order to maximize data locality.

The coordinator manages microshard assignments by maintaining the mapping from nodes to shards and from objects to shards. Other participants fetch object locations from the coordinator as needed and cache them locally. Node assignments within shards are modified only during failures or when the overall cluster size changes. The coordinator maintains a persistent TCP connection with each node and interprets connection termination as a node failure. Upon detecting a failure, it reconfigures the affected replica set by promoting the next node in the chain to primary and appending a new secondary node to the end of the chain. In contrast to node assignments, the object-to-shard mapping is much larger and changes more frequently, such as when new objects are created or workloads shift. This mapping is sharded and distributed across multiple physical machines to ensure scalability and availability.

4.4.2. Light Replication

When nodes in a shard experience high load due to a hot object, LAMBDASTORE can temporarily create *light replicas* of the object on other worker nodes. These light replicas do not participate in the replication protocol. Instead, during updates, the primary node pushes only the updated version number to the light replicas. The light replica lazily fetches the most recent data as needed during job execution. Like secondary nodes, When a workflow completes, the light replica sends the read set and write set associated with the workflow to the primary for finalization to ensure serializability across all participating nodes.

Light replication is especially effective for compute- or read-heavy workloads. Since reads can be processed in parallel, adding light replicas increases computational capacity without raising conflict rates. However, for write-heavy workloads, light replicas offer little benefit, as they can lead to a high number of conflicting transactions running concurrently,

ultimately wasting compute resources. For this reason, LAMBDASTORE currently only creates light replicas for objects that are not write-intensive.

4.4.3. Object Migration

Compare to light replication, object migration is more heavyweight as it involves transferring ownership of an object to a different shard. During migration, certain operations such as writes cannot be processed, leading to temporary unavailability. Despite this cost, object migration is essential for effective load balancing, as it allows hot objects to be distributed across multiple shards.

Object migration is managed by the coordinating service. It begins by identifying which microshard needs to be migrated. The coordinator then instructs the primary node currently storing the microshard to initiate the migration. During this process, the object is read-locked, and its data is transferred to the target node. While migration is in progress, read requests can still be served for the object. Once the new node has received all the data, the coordinator updates the microshard's location. At that point, the original node deletes its local copy and begins rejecting any further requests involving the migrated object.

4.5. Fault Tolerance

LAMBDASTORE can tolerate crash failures of individual nodes. We define a *failure* as a node crashing, losing power, or becoming disconnected from the network. It is important to note that while the platform tolerates crash failures in the system architecture, it also supports arbitrary (or Byzantine [44]) failures during function execution through its virtualization mechanism, as discussed in Section 4.2.

Transaction execution in LAMBDASTORE tolerates failures of any node, including the client or frontend that initiated the transaction. Since clients and frontends do not participate in state storage or workflow execution after submitting a request, their failure is non-critical. They can simply re-issue the request upon restart without impacting the correctness or progress of the transaction.

We generally refer to a replica set failure when any of its nodes fails. Each replica set in LAMBDASTORE has $f + 1$ nodes, allowing the system to tolerate up to f node failures without data loss. When a node fails, the affected replica set undergoes reconfiguration by restarting or replacing the failed node. The coordinator then notifies all other replica sets of the updated configuration.

4.5.1. Replica Failures

Failures of secondary replicas can be handled by the primary, as it is always responsible for processing transaction prepare and commit phases first. Once the primary is informed of a reconfiguration, it will re-issue any delegated transactions or jobs that were affected by the failure.

When the primary fails, a new primary is selected from the remaining secondary replicas by the coordinator. This choice allows the new primary to take over immediately, without requiring a complex recovery protocol. A new replica is then added to the set and synchronizes its state from the existing nodes. If necessary, the new primary reissues pending commit requests. While the state of in-progress transactions may be lost, atomicity is not violated, as incomplete transactions will not be committed and can be safely retried.

Since light replicas do not participate in the replication protocol, they act merely as a cache for the microshards and do not require explicit recovery. The coordinator can create additional light replicas as needed, and any jobs that were running on a failed light replica can be retried on other nodes within the shard.

4.5.2. Transaction Manager Failures

For multi-shard transactions, the transaction manager must consolidate state across all participating nodes. Some nodes may have already prepared the transaction and are waiting to finalize it to release their locks properly, while others may have already finalized the transaction. To uphold atomicity, it is essential to ensure that the transaction is either committed or aborted consistently across all nodes.

After recovery, all nodes notify the new transaction manager about any prepared transactions that originated from it. If the new manager has previously logged a commit for a transaction, it instructs the involved node(s) to proceed with the commit. If no commit was logged, the transaction either was aborted or had not yet completed its prepare phase, in which case it can be safely aborted without violating correctness.

4.5.3. Transaction Participant Failures

When a replica set fails, information about in-progress transactions, including those involving remote shards, may be lost. For transactions that are still in the execution phase, their jobs will simply be re-issued. Similarly, for transactions that were in the prepare phase at the time of failure, the prepare requests will be re-issued as well.

Transactions that have already been prepared or partially finalized must be finalized consistently across all participants. To achieve this, the new primary first checks for any in-progress transactions and then queries each transaction's manager to determine whether it should be committed or aborted. In the case of a commit, the new primary will always have access to the transaction's write set, as it would have been replicated during the prepare phase. At this point, it is guaranteed that the prepare phase succeeded at all participants as otherwise, the transaction would not have advanced to the commit phase.

4.5.4. Coordinator Failures

In our current prototype, coordinators are not replicated and therefore cannot recover from failures. A production-ready version of LAMBDASTORE would use a distributed metadata management service, such as ZooKeeper, to store object and node mappings. In that case, we could rely on the fault tolerance of the underlying metadata service to support recovery. We leave this integration to future work.

5. Evaluation

We evaluate LAMBDASTORE using both microbenchmarks and more realistic application workloads. The system is implemented in 33k lines of Rust code and builds on top of Tokio [45], a framework for asynchronous execution, and Wasmtime [46], a WebAssembly runtime. Our evaluation answers the following questions:

- What overheads are introduced by LAMBDASTORE's compute-storage colocated design?
- How much performance benefit does compute-storage colocation provide?
- Is LAMBDASTORE's transaction interface efficient? How does entry set granularity impact performance?
- How scalable and elastic are workflows deployed in LAMBDASTORE?
- Can LAMBDASTORE tolerate failures in application code?
- What benefit does LAMBDASTORE provide for realistic applications?

5.1. Experimental Setup

We compare LAMBDASTORE with OpenLambda, OpenWhisk, Apiary, and Faasm. Like most existing serverless systems, OpenWhisk relies on containers for isolation. Although containers are more lightweight than traditional virtual machines, they still incur significant overhead. Faasm combines containers and WebAssembly to reduce this over-

head. OpenLambda uses SOCK runtime [23], a container-based isolation technique, but it also supports a more efficient WebAssembly runtime [47]. We include both configurations in our evaluation. Finally, Apiary executes serverless functions as stored procedures in PostgreSQL or VoltDB. Unlike the other platforms, Apiary does not isolate job executions.

All benchmarks were executed on Cloudlab [48] with a cluster of c220g5 machines. Each machine is equipped with two Intel Xeon Silver 4114 CPUs (each having ten physical cores and Hyper-Threading), 200GB of DDR-4 memory, and 10Gbit NICs.

Among the baselines, OpenWhisk provides at-most-once semantics. In presence of worker failures, it does not re-execute failed requests. We similarly configure OpenLambda to provide at-most-once guarantees. Faasm, on the other hand, provides at-least-once semantics by tracking outstanding requests and re-executing them if necessary. Apiary leverages the transactional properties of its underlying database to ensure that each function is executed exactly once. These mechanisms inevitably introduce additional overheads compared with OpenWhisk and OpenLambda. For persisted storage, Faasm uses MinIO [49], a datastore with an API similar to S3. Apiary uses PostgreSQL. We use LAMBDASTORE as a storage backend for all other systems.

In the OpenWhisk setup, each compute machine runs a standalone OpenWhisk instance which encapsulates all core components: a *controller* that accepts user requests, an *invoker* that launches lambda instances, Kafka for streaming requests from the controller to the invoker, and CouchDB for storing the results of each lambda invocation. Each client machine runs a lightweight frontend that distributes requests evenly across the OpenWhisk instances for load balancing. We note that this setup deviates from the traditional OpenWhisk deployment model, in which a centralized controller node manages a set of invoker nodes. However, the performance observed in our configuration aligns with results reported in other studies [24].

For Apiary, we instantiate a single *worker process* on one of the storage nodes. This process simply forwards function invocations to PostgreSQL. Additionally, we deploy an HTTP frontend on each client machine. These frontends receive client requests and communicate with the worker process using Apiary's proprietary protocol. Data replication is configured by assigning two backup nodes for each PostgreSQL database.

5.2. Microbenchmarks

5.2.1. Virtualization and Scheduling Overheads

We first evaluate the overheads introduced by LAMBDASTORE using a benchmark application that computes a series of SHA-512 hashes over 1KB input data. To simulate compute-only, longer-running functions, we gradually increase the number of hashes computed per function call. In this experiment, a single machine is used to execute all functions. Figure 3 presents our results. Since virtualization and coordination overheads are largely independent of job duration, their relative impact diminishes for longer-running jobs. Once this overhead is sufficiently amortized, all serverless systems achieve similar overall throughput (100k hashes per second). Note that all reported numbers reflect warm starts, as the same function is invoked repeatedly.

We observe that serverless systems using WebAssembly for isolation (OpenLambda (WASM) and LAMBDASTORE) incur significantly lower overhead than systems with containerized runtimes (OpenWhisk, OpenLambda (SOCK), and Faasm). Notably, at one hash per function call, LAMBDASTORE achieves 798× the throughput of OpenWhisk. Apiary's approach, which relies on stored procedures, also introduces higher overhead compared to more optimized container-based approaches like SOCK and Faasm's containerized WebAssembly runtime. OpenLambda (WASM) initially outperforms LAMBDASTORE slightly, because LAMBDASTORE performs additional coordination checks to determine whether

the function will access the storage layer. This check enables more informed scheduling decisions, ultimately reducing data transmission and lowering transaction conflict rates for stateful workflows.

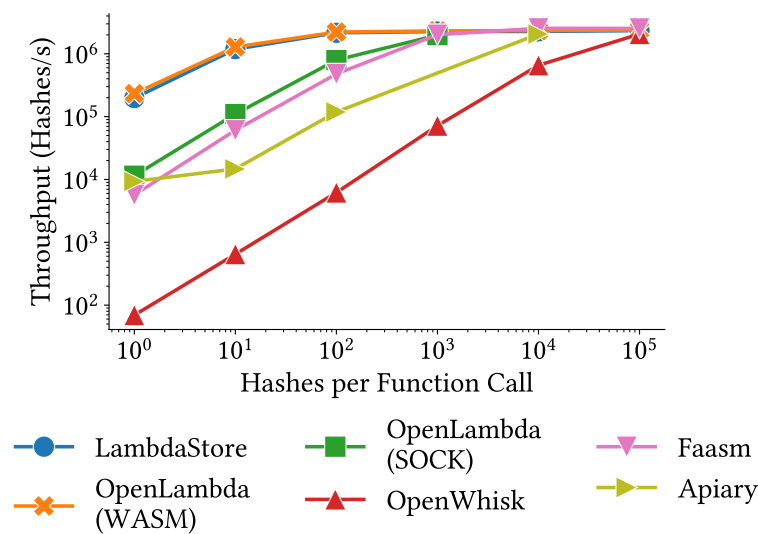


Figure 3. Comparison of virtualization and coordination overhead across different serverless systems. Overheads are most pronounced for short-running functions.

5.2.2. Throughput and Latency of Stateful Workloads

We evaluate LAMBDASTORE using a series of microbenchmarks with varying read-write workloads to quantify the performance benefits of compute-storage colocation. In this setup, we preload the storage system with 10 K objects, each containing 100 entries of 1 KB in size, resulting in a total of 1 M entries at the beginning of the experiment.

We ran the experiment using a set of three nodes to execute serverless functions. For systems without a built-in storage layer (OpenLambda and OpenWhisk), we deployed three additional nodes to form a storage shard. For Faasm, we set up a Kubernetes cluster with six machines to host workers, planners, and the storage backend. During initial testing, we observed significantly lower performance for object and entry creation compared to updates. To keep experiment startup times manageable, we limited the workload to 100 objects. We varied the number of clients and recorded both throughput and latency distribution for each configuration.

Figure 4 shows the highest throughput achieved under each workload with different read-write patterns. For LAMBDASTORE, OpenLambda, and OpenWhisk, the throughput decreases as the proportion of writes increases, primarily the cost of replicating data across storage nodes. Faasm, in contrast, shows increasing throughput with more writes and outperforms all other systems under write-only workloads. Upon investigation, we found that Faasm's write performance is orders of magnitude higher than its read performance because Faasm lazily pushes writes from worker nodes to MinIO. While Faasm provides interfaces for explicitly synchronization, this operation was not invoked in our benchmarks. Apiary also demonstrates a unique performance trend: as write chance increases, throughput initially decreases then rises again. This behavior is due to transaction conflicts, which only occur under mixed workloads involving both reads and writes.

Except for Faasm under the write-only workload, LAMBDASTORE outperforms all other systems due to compute-storage colocation. This advantage is especially pronounced under read-only workloads, where LAMBDASTORE achieves throughput at least an order of magnitude higher than all other systems. However, as the proportion of writes increases, the cost of replication begins to dominate overall performance. Under write-only workloads,

the benefit colocation offers is overshadowed by replication overhead, and OpenLambda (SOCK) achieves throughput comparable to LAMBDASTORE. We also note that, under mixed workloads, LAMBDASTORE exhibit the performance degradation seen in Apiary, demonstrating the efficiency of LAMBDASTORE's transaction interface, which we elaborate in Section 5.2.3.

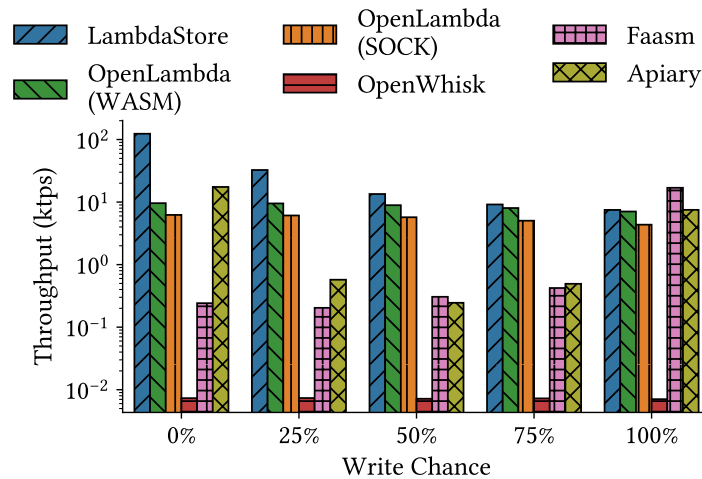


Figure 4. Comparison of throughput for read-write workloads.

The benefits of colocation become even more apparent when considering latency, as shown in Figure 5. In this experiment, we vary the number of concurrent requests to adjust total throughput and plot the mean latency against the achieved throughput. Across all workloads, LAMBDASTORE consistently exhibits lower latencies than all other systems we evaluated (except Faasm under write-only workload). The advantage is most significant under read-only workloads, where LAMBDASTORE maintains a latency below 20ms throughout the experiment since it avoids fetching data from a remote machine. Under write-only workload, LAMBDASTORE's latency is roughly equal to that of OpenLambda (WASM), as the cost of chain replication becomes the dominant factor. Nonetheless, LAMBDASTORE continues to outperform other systems, including OpenWhisk and Apiary.

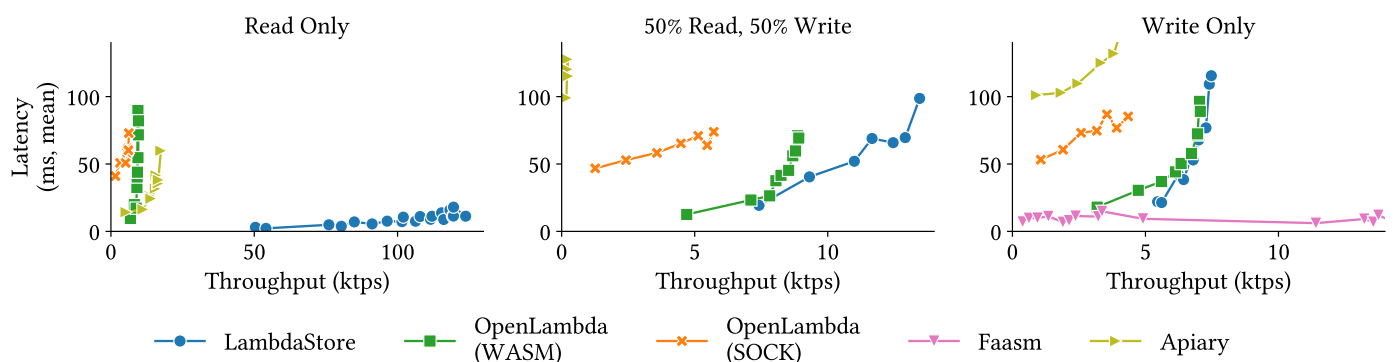


Figure 5. Comparison of latencies for read-write workloads. LAMBDASTORE outperforms other systems in almost all cases.

5.2.3. Scalability of Transactional Workflows

LAMBDASTORE provides a more efficient transaction interface than Apiary primarily due to dynamic lock granularity through entry sets. While PostgreSQL supports row-level locking as its finest lock granularity, LAMBDASTORE enables even finer-grained locking for write-heavy objects, reducing contention. Figure 6 illustrates the impact of lock granularity

on performance. In this benchmark, all operations target a single object, creating high lock contention. We manually insert guards into the object prior to execution. Since guards define the boundaries of entry sets, adding more guards results in finer-grained locking, allowing independent workflows to proceed concurrently without conflicts.

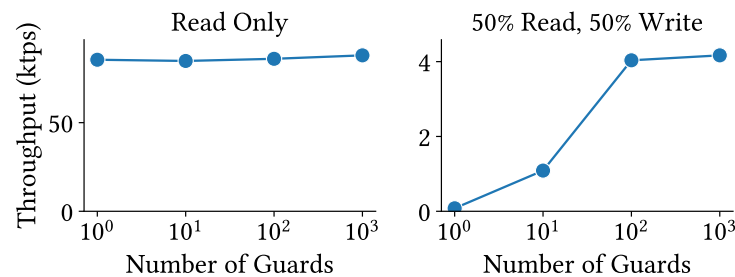


Figure 6. Effect of object partitioning on performance. For mixed workloads, performance improves significantly with more guards (i.e., finer-grained partitioning), while for read-only workloads show consistent performance.

We compare two types of workloads: a read-only workload and a mixed workload consisting of 50% reads and 50% writes. In the read-only case, transactions do not conflict, resulting in no lock contention. Consequently, throughput remains constant regardless of the number of guards. In the mixed workload, however, the number of entry sets—and therefore the lock granularity—has a significant impact on performance. As the number of guards increases, throughput improves substantially, up to the point where the number of entry sets exceeds the number of CPU cores on the nodes.

5.2.4. Elasticity

LAMBDASTORE provides high elasticity and scalability through light replication. To evaluate its impact, we use a microbenchmark that reads 1 MB of encrypted data. Currently, LAMBDASTORE does not implement automated policies to add or remove light replicas based on observed workload. Instead, we explicitly instruct the system to create light replicas while the benchmark runs. The test setup begins with a replica set containing three full replicas, and we gradually add up to three light replicas to the shard to observe their effect on total throughput.

Figure 7 presents the results of our experiment. Under this read-only workload, we observe that throughput increases almost linearly with the number of replicas in the shard. However, when the third light replica is added, the performance gain is slightly less than that observed with the first two. This sublinear increase is due to the fact that, regardless of the number of light replicas, the primary replica remains responsible for coordinating transactions. All workloads executed on secondary or light replicas must still be finalized by the primary. As more light replicas are added, the primary eventually becomes a bottleneck, limiting further performance gains. This behavior also suggests that short-running workflows benefit less from light replication than long-running workflows, since the relative cost of transaction coordination becomes more significant for shorter executions.

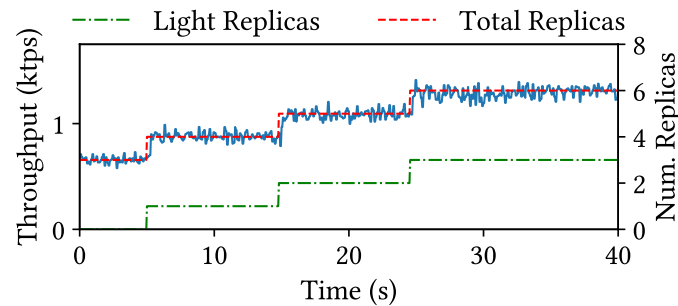


Figure 7. LAMBDASTORE's performance as light replicas are added to a shard. The blue line shows the throughput of the experiment cluster. It scales almost linearly with the total number of replicas.

5.2.5. Faulty Functions

Figure 8 illustrates the performance of LAMBDASTORE when an application's function refuses to terminate. Specifically, we execute the hashing workload from Section 5.2.1 on a single node and inject as many non-terminating function calls as the number of worker threads. In this experiment, the maximum execution time for a function is set to 1 s. Before reaching this execution time limit, system throughput is significantly degraded, as the non-terminating functions consume the majority of resources, starving other requests. However, once these faulty functions time out and are terminated, the system's performance quickly recovers. This behavior contrasts with Apiary, which executes serverless functions as stored procedures in PostgreSQL. In our test with a similarly faulty function, Apiary failed to terminate the function, due to the lack of fine-grained isolation in its execution model.

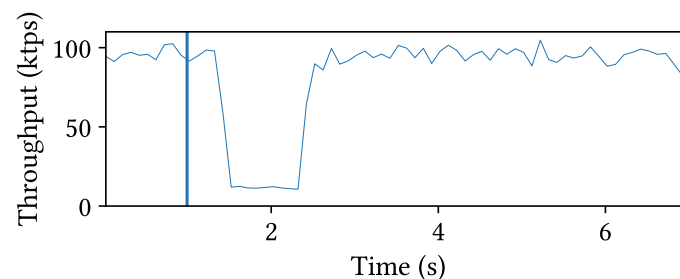


Figure 8. LAMBDASTORE's performance after injection of faulty functions (indicated by the vertical line). The light blue line shows the throughput of the experiment cluster. Non-terminating functions are aborted after a 1-s timeout, and the system recovers quickly.

5.3. Application Performance

We evaluated end-to-end performance using two different application workloads. We first examine CLOUDFORUM, an online message board similar to Reddit, introduced in Section 3.3. In this application, discussions are organized into *threads*, each containing multiple *posts*. Threads are grouped under *communities* (similar to subreddits). This workload consists of the following transaction types, with their relative frequency indicated in parentheses:

- **get-thread (90%):** Fetches the contents of a thread, including all its posts. This operation is a read-only operation implemented as a single function call.
- **add-comment (10%):** Adds a new comment to an existing thread. This operation updates both the thread and the poster's account, with each update performed by a separate function call.

For this benchmark, we preload the system with 100 communities, containing a total of 100 K threads. Each client thread is associated with a unique user account, and the number of concurrent client threads scales with the number of shards in the system.

The second application we use to evaluate the storage system is RETWIS [50] a benchmark that simulates a Twitter-like social network. In this workload, users can follow other users, create *posts*, and retrieve their *timelines* that contain the most recent posts from themselves and their followers. This workload consists of the following transaction types:

- **get-timeline (90%)**: Fetches the most recent 50 posts on the calling user's timeline. This is a read-only operation, implemented as a single function call.
- **create-post (10%)**: Adds a new post to the timeline of the posting user as well as each of their followers. This operation generates a map call on all followers of the poster.

We preloaded each user with 20 followers, so a single create-post transaction updates 21 objects. The system was initialized with 500 K users, each having 10 followers and 10 posts on their timeline. Each post is 1 KB in size. This setup avoids network bottlenecks.

In this experiment, we focus on LAMBDASTORE and OpenLambda (WASM), as they significantly outperform other systems. To evaluate scalability, we increase the number of replica sets (with three nodes per set). For OpenLambda (WASM), we maintain a one-to-one ratio between compute nodes and storage nodes. For example, with 6 replica sets, we launch 18 nodes for OpenLambda and 18 nodes for LAMBDASTORE as its storage backend, totaling 36 nodes. Figure 9 shows our results. We observe that LAMBDASTORE scales nearly linearly with the number of replica sets in both applications. While OpenLambda also benefits from increased parallelism, its disaggregated design results in much lower performance despite using twice as many servers. We also present latency vs. throughput curves in Figure 10. Consistent with the microbenchmark results from Section 5.2.2, LAMBDASTORE achieves significantly higher throughput while maintaining low latency. These results suggest that common stateful applications, which typically involves little computation, can benefit greatly from LAMBDASTORE's compute-storage co-design.

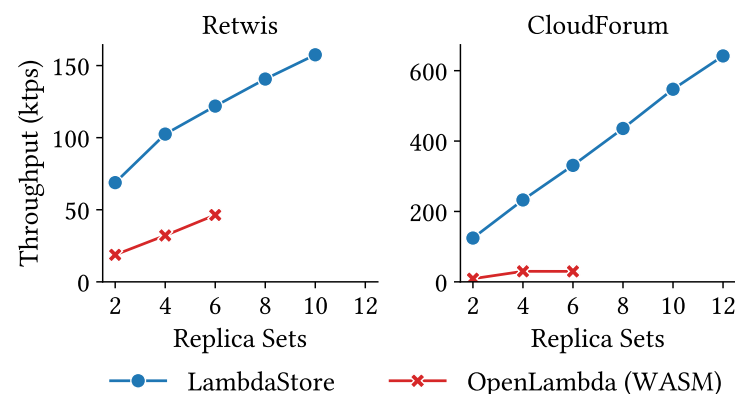


Figure 9. Scalability of applications running on LAMBDASTORE. Performance of OpenLambda (WASM) is shown for up to 6 replica sets as it uses twice as many nodes.

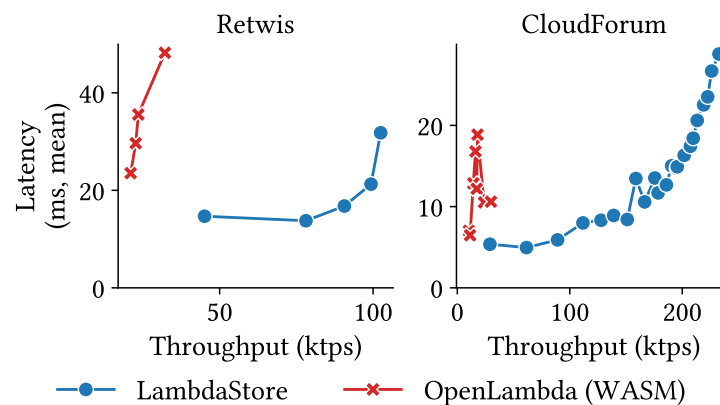


Figure 10. Application latency with four replica sets.

6. Discussion and Future Directions

In this section, we discuss several limitations in the current LambdaStore implementation and how we plan to address them in future work.

6.1. Scheduling Policies

LAMBDASTORE's design and implementation open up many scheduling opportunities—including where new objects should be placed, when to migrate an object or create new light replicas, on which replica serverless jobs should run, etc. Its compute-storage co-design enables the scheduler to make more informed decisions by leveraging additional information such as each function's read-write pattern, the size and location of states it can access, etc. A carefully designed scheduling policy has the potential to further enhance LAMBDASTORE's performance. We leave the exploration of scheduling policies to future work.

6.2. Access Control and Data Sharing

Currently, clients can invoke any public function of any application managed by the platform. A future version of LAMBDASTORE should explore session management and access control to isolate applications from one another and protect user data from unauthorized clients. Existing storage systems often include built-in access control mechanisms, which could be integrated with session information from the compute layer to enable fine-grained, application-specific access control.

6.3. Long-Running Workflows

So far, we described a system built for interactive cloud applications. Some design decisions, such as serializable workflow and optimistic concurrency control, can starve long-running workflows due to repeated conflicts. LAMBDASTORE can mitigate this issue in two ways. First, many database systems use priority scheduling and lock escalation to prevent long-running transactions from being starved. LAMBDASTORE can adopt similar techniques to ensure progress for long-running workflows. Second, long-running workflows often do not require strict serializability. LAMBDASTORE can be extended to support configurable consistency guarantees for applications or even individual workflows. We leave the design of such extensions to future work.

6.4. Resource Isolation

While WebAssembly provides fault isolation by injecting safeguards at compile time, it does not offer resource isolation at runtime. Existing wisdom suggests using container techniques such as Linux cgroups to offer strong resource isolation [11,27]. LAMBDASTORE

can follow this approach. Alternatively, LAMBDASTORE can implement resource isolation at the application level by using a custom scheduler that enforces resource limits on each job. More specifically, LAMBDASTORE currently maps serverless jobs to the worker's process space and uses Tokio for user-level scheduling. We can augment the Tokio scheduler to ensure that no serverless job consumes more resources than it has been allocated. We leave the design and implementation of such resource-aware scheduling to future work.

6.5. Support for Legacy Applications

A significant limitation of our current design is the lack of support for legacy applications that rely on a POSIX-like API. However, the WebAssembly community is actively developing a standardized system call interface similar to POSIX [51]. While the object-based model used in LAMBDASTORE does not directly map to such an interface, LAMBDASTORE could expose a local file system abstraction for each object by mapping individual files to entries in the object's storage. This abstraction would allow a legacy application to run within the system's LAMBDAOBJECTS model. It is important to note that this approach would still require minor modifications to legacy applications and may not support all system calls.

7. Related Work

7.1. Stateful Serverless

In the traditional serverless paradigm, functions are stateless, and external storage services such as Amazon S3 [15] are used to persist application state. In workloads where multiple serverless jobs operate within the same session, this architecture incurs significant performance overhead due to repeated accesses to the external storage layer by different jobs. To address these limitations, two stateful serverless abstractions have emerged. The first is *workflows*, which compose multiple function invocations into larger computations. Amazon Step Functions [52], Azure Durable Functions [53] and CloudBurst [32] follow this abstraction. The second is *actors*, where the object-oriented modeling is used. This model is seen in CloudFlare Durable Objects [54] and Azure Durable Entities [55]. The LAMBDAOBJECTS abstraction in LAMBDASTORE is inspired by the actor model, as it naturally associates functions with their application data to enable compute-storage colocation.

Existing actor-based platforms suffer from key limitations. In CloudFlare Durable Objects, requests to the same object are served sequentially, limiting elasticity. Azure Durable Entities, on the other hand, retain the traditional disaggregated architecture and do not support object partitioning, leading to long function startup times due to the need to load the entire object. Moreover, to prevent deadlocks, durable entities are not allowed to invoke functions of other durable entities. LAMBDASTORE addresses all of these problems to meet RAISED properties.

7.2. WebAssembly and Serverless

Traditional serverless platforms typically rely on mechanisms provided by operating systems, such as virtual machines or containers [5,9,10,14], to isolate functions. As discussed earlier, these mechanisms suffer from millisecond-scale cold start times due to the overhead of initializing the runtime environment. To reduce this latency, many recent serverless platforms have adopted lightweight software fault isolation techniques [11,56–58]. These platforms often compile functions (along with their dependencies) into WebAssembly [40], which is then executed in a WebAssembly runtime such as Wasmtime [46] and Wasmer [59]. This approach significantly reduces cold start times but may be more susceptible to software bugs due to the reduced isolation provided by userspace runtimes. LAMBDASTORE adopts this approach and uses Wasmtime as its run-

time environment. To our knowledge, none of the existing WebAssembly-based serverless platforms provide serializable transactions, and some—such as Shredder [11]—do not even support data replication.

7.3. Microservices

Serverless systems share similarities with microservices, which decompose an application into many semi-independent components [25,36,60–62]. This architecture promotes fault isolation, as a crash in one service does not directly impact others, and supports development efficiency, enabling different teams to work independently on separate microservice components.

Managing microservices, however, involves provisioning the required resources, starting the services, and handling failures as they arise [63], all of which typically requires prior knowledge of anticipated workloads and often leads to overprovisioning to handle potential demand spikes.

While microservices improve scalability and resilience, they still require developers to reason explicitly about failures. In particular, cascading failures—where faults propagate across services—can be difficult to debug and mitigate [64].

8. Conclusions

We described LAMBDASTORE, a new cloud computing platform that unifies storage and execution to better support stateful applications in the serverless setting. LAMBDASTORE leverages the LAMBDAOBJECTS abstraction to associate serverless functions with their application data, enabling compute–storage colocation. By embedding a transactional key-value store within the platform, LAMBDASTORE provides strong transactional guarantees for serverless workflows. To preserve elasticity, LAMBDASTORE employs light replication and object migration to adapt to changing workloads rapidly. Our evaluation shows that LAMBDASTORE outperforms existing serverless platforms in both throughput and latency while providing stronger guarantees, making it a promising step toward interactive stateful serverless applications. We hope that this work inspires future research on stateful serverless systems.

Author Contributions: Conceptualization, K.M. and S.Q.; methodology, K.M., S.Q. and A.J.; software, K.M., S.Q. and A.J.; validation, K.M., S.Q. and A.J.; formal analysis, K.M. and S.Q.; investigation, K.M., S.Q. and A.J.; resources, K.M., S.Q., A.A.-D. and R.A.-D.; data curation, K.M., S.Q. and A.J.; writing—original draft preparation, K.M. and S.Q.; writing—review and editing, A.A.-D. and R.A.-D.; visualization, K.M. and S.Q.; supervision, A.A.-D. and R.A.-D.; project administration, K.M.; funding acquisition, A.A.-D. and R.A.-D. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the National Science Foundation (CNS-2402859).

Data Availability Statement: The original data presented in the study are openly available in *lambdastore-data* at <https://github.com/qusuyan/lambdastore-data> (accessed on 11 January 2026).

Acknowledgments: During the preparation of this manuscript/study, the author(s) used GPT-5 for the purposes of grammar checks. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no competing interests. The author Kai Mast is currently employed by the company Provable Inc. The work was done when he was studying at UW-Madison. There is no conflict of interest between any of the authors and the company.

Appendix A. Transaction Protocol

Appendix A.1. Correctness Proof

For correctness, it suffices to prove that all transactions that read a version number staler than the value will abort during the prepare phase; the rest follows the correctness of Silo.

Let v_{read} denote the version number read by the transaction, v_{local} the version number associated with the value read by the transaction, v_{eset}^t the version number managed by the entry set at time t , and v_{global}^t the version number associated with the last written value for the given key at time t . For each write committed, v_{global} is incremented by 1, and we assume without loss of generality that v_{eset} is also incremented by 1 during the update so that $v_{\text{global}} = v_{\text{eset}}$ after the write is committed.

We first prove that, for each read request, $v_{\text{read}} \leq v_{\text{local}}$. Since during the commit phase, we always update the version number after writing the new value back to storage, $v_{\text{eset}}^t \leq v_{\text{global}}^t$ holds for any t . We additionally note that, since transactions acquire the write locks before committing, at any point, only one writer can update the record, so $v_{\text{eset}}^t \geq v_{\text{global}}^t - 1$ for any t . During each read, the transactions always access the entry set's version number before the value ($t_{\text{ver}} < t_{\text{val}}$). Therefore, $v_{\text{read}} = v_{\text{eset}}^{t_{\text{ver}}} \leq v_{\text{global}}^{t_{\text{val}}} = v_{\text{local}}$.

Next, we prove that all transactions that read a version number older than the corresponding read value ($v_{\text{read}} < v_{\text{local}}$) will abort. During the prepare phase, after acquiring all write locks, each transaction walks through its read set. For each record the transaction reads, it checks that the record is not locked by another transaction and the read version number matches the latest version number for that record (i.e., $v_{\text{read}} = v_{\text{eset}}^c$ where c is the current check time). If either condition is not met, the transaction aborts. Suppose for a given transaction, one record in the read set has $v_{\text{read}} < v_{\text{local}}$. We first consider if $v_{\text{read}} < v_{\text{local}} - 1$. Then, since $v_{\text{local}} \leq v_{\text{global}}^c$, we have $v_{\text{read}} < v_{\text{global}}^c - 1 \leq v_{\text{eset}}^c$. The transaction will abort. On the other hand, if $v_{\text{read}} = v_{\text{local}} - 1$ and $v_{\text{read}} = v_{\text{eset}}^c$, then we have $v_{\text{eset}}^c = v_{\text{local}} - 1 \leq v_{\text{global}}^c - 1$ and hence $v_{\text{eset}}^c = v_{\text{global}}^c - 1$. This is a special case where a concurrent transaction in the commit phase tries to update the record, so the record must be locked and the transaction will therefore abort as well.

Appendix A.2. Complexity Analysis

Let n be the number of objects in the storage. LAMBDASTORE supports various implementations for its storage, and we use $O(r(n))$ and $O(w(n))$ to denote the time complexity of reading from (*get*) and writing to (*put*) the storage, respectively. Let R and W be the size of the read set and write set of a transaction. In our current implementation, we use a hash map to store the entry sets, so *get_version*, *update_version*, *lock*, *unlock*, and *is_locked* have a time complexity of $O(1)$. Furthermore, we use hash maps to implement transaction read and write sets, so adding and looking up entries in these sets also take $O(1)$ time. Therefore, the time complexity of the transaction protocol is:

- $\text{read}(\text{txn}, \text{key}): O(1) + O(r(n)) + O(1) = O(r(n));$
- $\text{write}(\text{txn}, \text{key}, \text{value}): O(1);$
- $\text{prepare}(\text{txn}): W \times O(1) + R \times (O(1) + O(1)) = O(W + R);$
- $\text{commit}(\text{txn}): W \times (O(w(n)) + O(1) + O(1)) = O(W \times w(n)).$

Notably, *commit(txn)* does not add additional overhead since LAMBDASTORE simply postpones the actual writes to the storage until after transaction execution and validation.

References

- Baldini, I.; Castro, P.C.; Chang, K.S.; Cheng, P.; Fink, S.; Ishakian, V.; Mitchell, N.; Muthusamy, V.; Rabbah, R.; Slominski, A.; et al. Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*; Springer: Singapore, 2017; pp. 1–20. [CrossRef]
- Cloudflare Inc. Why Use Serverless Computing? Available online: <https://www.cloudflare.com/learning/serverless/why-use-serverless/> (accessed on 11 January 2026).
- Marketsandmarkets Private Ltd. Serverless Architecture Market. Available online: <https://www.marketsandmarkets.com/Market-Reports/serverless-architecture-market-64917099.html> (accessed on 11 January 2026).
- Data Dog. The State of Serverless. Available online: <https://www.datadoghq.com/state-of-serverless/> (accessed on 11 January 2026).
- Apache Software Foundation. OpenWhisk Architecture. Available online: <https://cwiki.apache.org/confluence/display/OPENWHISK/System+Architecture> (accessed on 11 January 2026).
- Google Inc. Google Cloud Functions. Available online: <https://cloud.google.com/functions> (accessed on 11 January 2026).
- Jia, Z.; Witchel, E. Boki: Stateful Serverless Computing with Shared Logs. In Proceedings of the SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Koblenz, Germany, 26–29 October 2021; pp. 691–707.
- Romero, F.; Chaudhry, G.I.; Goiri, I.; Gopa, P.; Batum, P.; Yadwadkar, N.J.; Fonseca, R.; Kozyrakis, C.; Bianchini, R. FaaS\$: A Transparent Auto-Scaling Cache for Serverless Applications. In Proceedings of the SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, 1–4 November 2021; pp. 122–137.
- Microsoft. Azure Functions. Available online: <https://azure.microsoft.com/en-us/services/functions/> (accessed on 11 January 2026).
- Amazon Web Services. AWS Lambda. Available online: <https://aws.amazon.com/lambda/> (accessed on 11 January 2026).
- Shillaker, S.; Pietzuch, P.R. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In Proceedings of the USENIX Annual Technical Conference, Virtual, 15–17 July 2020; pp. 419–433.
- Xie, M.; Qian, C.; Litz, H. En4S: Enabling SLOs in Serverless Storage Systems. In Proceedings of the 2024 ACM Symposium on Cloud Computing, SoCC 2024, Redmond, WA, USA, 20–22 November 2024; pp. 160–177.
- Zhang, H.; Mu, S.; Angel, S.; Liu, V. CausalMesh: A Causal Cache for Stateful Serverless Computing. *ACM SIGMOD Rec.* **2025**, *54*, 48–57. [CrossRef]
- Agache, A.; Brooker, M.; Iordache, A.; Liguori, A.; Neugebauer, R.; Piwonka, P.; Popa, D.M. Firecracker: Lightweight Virtualization for Serverless Applications. In Proceedings of the Symposium on Networked System Design and Implementation, Santa Clara, CA, USA, 25–27 February 2020; pp. 419–434.
- Amazon. S3 Cloud Object Storage. Available online: <https://aws.amazon.com/s3/> (accessed on 11 January 2026).
- Amazon. Learning Serverless (and Why It Is Hard). 2022. Available online: <https://pauldjohnston.medium.com/learning-serverless-and-why-it-is-hard-4a53b390c63d> (accessed on 11 January 2026).
- Hellerstein, J.M.; Faleiro, J.M.; Gonzalez, J.; Schleier-Smith, J.; Sreekanti, V.; Tumanov, A.; Wu, C. Serverless Computing: One Step Forward, Two Steps Back. In Proceedings of the 9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, 13–16 January 2019.
- Eismann, S.; Scheuner, J.; Eyk, E.V.; Schwinger, M.; Grohmann, J.; Herbst, N.; Abad, C.L.; Iosup, A. Serverless Applications: Why, When, and How? *IEEE Softw.* **2021**, *38*, 32–39. [CrossRef]
- Shahrad, M.; Fonseca, R.; Goiri, I.; Chaudhry, G.I.; Batum, P.; Cooke, J.; Laureano, E.; Tresness, C.; Russinovich, M.; Bianchini, R. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In Proceedings of the USENIX Annual Technical Conference, Virtual, 15–17 July 2020; pp. 205–218.
- Klimovic, A.; Wang, Y.; Kozyrakis, C.; Stuedi, P.; Pfefferle, J.; Trivedi, A. Understanding Ephemeral Storage for Serverless Analytics. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, 11–13 July 2018; pp. 789–794.
- Amazon. AWS Knowledge Center: How Do I Make My Lambda Function Idempotent? Available online: <https://repost.aws/knowledge-center/lambda-function-idempotent> (accessed on 11 January 2026).
- Saltzer, J.H.; Reed, D.P.; Clark, D.D. End-To-End Arguments in System Design. *ACM Trans. Comput. Syst.* **1984**, *2*, 277–288. [CrossRef]
- Oakes, E.; Yang, L.; Zhou, D.; Houck, K.; Harter, T.; Arpaci-Dusseau, A.C.; Arpaci-Dusseau, R.H. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, 11–13 July 2018; pp. 57–70.
- Kraft, P.; Li, Q.; Kaffes, K.; Skiadopoulos, A.; Kumar, D.; Cho, D.; Li, J.; Redmond, R.; Weckwerth, N.W.; Xia, B.S.; et al. Apiary: A DBMS-Backed Transactional Function-as-a-Service Framework. *arXiv* **2022**, arXiv:2208.13068.

25. Gan, Y.; Zhang, Y.; Cheng, D.; Shetty, A.; Rathi, P.; Katarki, N.; Bruno, A.; Hu, J.; Ritchken, B.; Jackson, B.; et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, Providence, RI, USA, 13–17 April 2019; pp. 3–18.
26. Apache Software Foundation. Apache Kafka. Available online: <https://kafka.apache.org/> (accessed on 11 January 2026).
27. Redis. Kubernetes Documentation. Available online: <https://kubernetes.io/docs/home/> (accessed on 11 January 2026).
28. Akkus, I.E.; Chen, R.; Rimac, I.; Stein, M.; Satzke, K.; Beck, A.; Aditya, P.; Hilt, V. SAND: Towards High-Performance Serverless Computing. In Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, 11–13 July 2018; pp. 923–935.
29. Copik, M.; Calotoiu, A.; Réthy, G.; Böhringer, R.; Bruno, R.; Hoefler, T. Process-as-a-Service: Unifying Elastic and Stateful Clouds with Serverless Processes. In Proceedings of the 2024 ACM Symposium on Cloud Computing, SoCC 2024, Redmond, WA, USA, 20–22 November 2024; pp. 223–242.
30. Klimovic, A.; Wang, Y.; Stuedi, P.; Trivedi, A.; Pfefferle, J.; Kozyrakis, C. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, 8–10 October 2018; pp. 427–444.
31. Yu, M.; Cao, T.; Wang, W.; Chen, R. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, USA, 17–19 April 2023; pp. 1489–1504.
32. Sreekanti, V.; Wu, C.; Lin, X.C.; Schleier-Smith, J.; Gonzalez, J.; Hellerstein, J.M.; Tumanov, A. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* **2020**, *13*, 2438–2452. [[CrossRef](#)]
33. Nair, A.; Szewczyk, R.; Jennings, D.; Barbalace, A. Near-Storage Processing in FaaS Environments with Funclets. In Proceedings of the 25th International Middleware Conference, MIDDLEWARE 2024, Hong Kong, China, 2–6 December 2024; pp. 145–157.
34. Tang, Y.; Yang, J. Lambdata: Optimizing Serverless Computing by Making Data Intents Explicit. In Proceedings of the 13th IEEE International Conference on Cloud Computing, CLOUD 2020, Virtual, 18–24 October 2020; pp. 294–303.
35. Zhang, T.; Xie, D.; Li, F.; Stutsman, R. Narrowing the Gap Between Serverless and its State with Storage Functions. In Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, 20–23 November 2019; pp. 1–12.
36. Netflix Technology Blog. Netflix Platform Engineering—We’re Just Getting Started. Available online: <http://netflixtechblog.com/netflix-platform-engineering-were-just-getting-started-267f65c4d1a7> (accessed on 11 January 2026).
37. Google. LevelDB: A Fast Key-Value Storage Library. 2011. Available online: <https://github.com/google/leveldb> (accessed on 11 January 2026).
38. Gray, J.; Helland, P.; O’Neil, P.E.; Shasha, D.E. The Dangers of Replication and a Solution. In Proceedings of the SIGMOD International Conference on Management of Data, Montréal, QC, Canada, 4–6 June 1996; pp. 173–182.
39. Hunt, P.; Konar, M.; Junqueira, F.P.; Reed, B.C. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In Proceedings of the 2010 USENIX Annual Technical Conference, USENIX ATC 2010, Boston, MA, USA, 23–25 June 2010.
40. W3 WebAssembly Working Group. WebAssembly Specification. Available online: <https://webassembly.org/specs/> (accessed on 11 January 2026).
41. Tu, S.; Zheng, W.; Kohler, E.; Liskov, B.; Madden, S. Speedy transactions in multicore in-memory databases. In Proceedings of the Symposium on Operating Systems Principles, Farmington, PA, USA, 3–6 November 2013; pp. 18–32.
42. Raju, P.; Kadekodi, R.; Chidambaram, V.; Abraham, I. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In Proceedings of the Symposium on Operating Systems Principles, Shanghai, China, 29–31 October 2017; pp. 497–514.
43. Annamalai, M.; Ravichandran, K.; Srinivas, H.; Zinkovsky, I.; Pan, L.; Savor, T.; Nagle, D.; Stumm, M. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In Proceedings of the Symposium on Operating System Design and Implementation, Carlsbad, CA, USA, 8–10 October 2018; pp. 445–460.
44. Lamport, L.; Shostak, R.E.; Pease, M.C. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* **1982**, *4*, 382–401. [[CrossRef](#)]
45. The Tokio Contributors. Tokio-Uring. Available online: <https://github.com/tokio-rs/tokio-uring> (accessed on 11 January 2026).
46. Bytecode Alliance. Wasmtime. Available online: <https://wasmtime.dev/> (accessed on 11 January 2026).
47. OpenLambda WebAssembly Worker. Available online: <https://github.com/open-lambda/open-lambda/tree/main/wasm-worker> (accessed on 11 January 2026).
48. Duplyakin, D.; Ricci, R.; Maricq, A.; Wong, G.; Duerig, J.; Eide, E.; Stoller, L.; Hibler, M.; Johnson, D.; Webb, K.; et al. The Design and Operation of CloudLab. In Proceedings of the USENIX Annual Technical Conference, Renton, WA, USA, 10–12 July 2019; pp. 1–14.
49. MinIO Inc. MinIO. Available online: <https://min.io/> (accessed on 11 January 2026).

50. Redis. Redis Patterns Example. Available online: <https://redis.io/docs/latest/develop/clients/patterns/twitter-clone/> (accessed on 11 January 2026).
51. The Bytecode Alliance. WASI: The Web Assembly System Interface. Available online: <https://wasi.dev/> (accessed on 11 January 2026).
52. Amazon. Amazon Step Functions. Available online: <https://aws.amazon.com/step-functions/> (accessed on 11 January 2026).
53. Microsoft. Azure Durable Functions. Available online: <https://learn.microsoft.com/en-us/azure/azure-functions/durable/> (accessed on 11 January 2026).
54. Cloudflare Inc. Using Durable Objects, Cloudflare Docs. Available online: <https://developers.cloudflare.com/workers/learning/using-durable-objects> (accessed on 11 January 2026).
55. Burckhardt, S.; Gillum, C.; Justo, D.; Kallas, K.; McMahon, C.; Meiklejohn, C.S. Durable functions: Semantics for stateful serverless. In Proceedings of the ACM on Programming Languages 5 (OOPSLA), New York, NY, USA, 18 October 2021; pp. 1–27.
56. Cloudflare Inc. Cloudflare Workers. Available online: <https://workers.cloudflare.com/> (accessed on 11 January 2026).
57. McMullen, T. How Terrarium Reframes the Compiler and Sandbox Relationship. Available online: <https://www.fastly.com/blog/terrarium-reframes-compiler-sandbox-relationship> (accessed on 11 January 2026).
58. Mast, K.; Arpaci-Dusseau, A.C.; Arpaci-Dusseau, R.H. LambdaObjects: Re-aggregating storage and execution for cloud computing. In Proceedings of the Workshop on Hot Topics in Storage and File Systems, Virtual, 27–28 June 2022; pp. 15–22.
59. Wasmer Inc. Wasmer. Available online: <https://wasmer.io> (accessed on 11 January 2026).
60. Jia, Z.; Witchel, E. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual, 19–23 April 2021; pp. 152–166.
61. Dragoni, N.; Giallorenzo, S.; Lluch-Lafuente, A.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: Yesterday, Today, and Tomorrow. *Present Ulterior Softw. Eng.* **2017**, 195–216. [CrossRef]
62. IBM. What Are Microservices? Available online: <https://www.ibm.com/topics/microservices> (accessed on 11 January 2026).
63. Ozkaya, M. Deploying Microservices on Kubernetes. Available online: <https://medium.com/aspnetrun/deploying-microservices-on-kubernetes-35296d369fdb> (accessed on 11 January 2026).
64. Meiklejohn, C.S.; Estrada, A.; Song, Y.; Miller, H.; Padhye, R. Service-Level Fault Injection Testing. In Proceedings of the SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, 1–4 November 2021; pp. 388–402.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.