White Paper:
Data-Centric Serverless Computing with LambdaStore

Kai Mast [*]

February 2, 2025

Abstract

LambdaStore is a new serverless platform that embeds computation into the storage layer using WebAssembly. It implements an object-oriented model, where functions are bundled with their associated data and execute directly where the data resides. This co-location allows to provide better semantics and higher performance than the state-of-the art.

This paper outlines the design of LambdaStore and makes the three contributions. First, it describes the architecture of a scalable object-oriented database that co-locates storage and execution. Second, the paper explains a novel mechanism to provide serializabilty for serverless workflows while maintaining high performance. Finally, we cover how to make such a system workload-agnostic by dynamically adjusting object granularity.

Evaluation shows that LambdaStore outperforms conventional serverless designs while providing stronger consistency guarantees, lower latencies, and better fault tolerance. We also demonstrate that this design uses less resources, enabling cloud computing at much lower cost.

1 Introduction

Serverless computing promises to enable the rapid development and deployment of applications and services [11, 9]. The serverless approach is attractive for many reasons, as one can entirely avoid purchasing, provisioning, and managing backend machines; rather, one simply develops the necessary application logic, and the infrastructure handles execution, automatically scaling up and down as needed. The market for serverless is already large (nearly $8B), and some project it to exceed $20B by the end of 2026 [27]. As one CEO said, "Serverless has demonstrated that it is the operational model of the future." [13]

Existing serverless platforms largely follow a *disaggregated* approach. For example, let us examine the most popular platform for serverless today, AWS Lambda [35]. Here, lambda functions, custom executables or scripts, execute in a virtualized environment provided by AWS [3]. Data is often stored in a scalable data service, such as Amazon S3 [6]; lambda functions, when executed, can read and write objects within S3, and thus implement the desired functionality. By separating data and compute, each can be scaled independently. Application logic is then implemented by composing multiple function calls into a so-called workflow.

The current approach works well with certain non-interactive applications, but one quickly encounters difficulties when building more complex, data-intensive applications and services [5, 23]. Recent surveys noted that many serverless functions have an execution length in the order of seconds, are invoked infrequently, and operate on datasets of less than 10 Mb [15, 36]. These use cases hint at the limitations of current platforms: serverless execution in its current form exhibits high latencies, and, additionally, encounters poor performance when interacting with underlying storage systems [24].

In addition, most serverless platforms are challenging to program, providing "at least once"-semantics and no transactional guarantees; functions may execute more than once, and partial results of an ongoing execution can be observed by other function calls. As a result, serverless platforms often require functions to be idempotent to accommodate these semantics [2] and for applications to provide their own concurrency control mechanisms if needed, directly contradicting one of the core tenets of serverless systems: less complexity for the application developer.

This paper introduces LambdaStore: an integrated serverless execution engine and scalable storage system for low-latency cloud applications. Modern storage systems already provide all of the correctness guarantees and scalability required by most applications; our work details how we can embed *untrusted* applications into a cloud-based storage system, which enables different

[*] Work on this project started at UW Madison. Please see the conclusion for a full list of collaborators.

applications to share the same storage system, increasing resource efficiency.

Such a co-located design closely follows the end-to-end argument [34]; instead of providing consistency and fault-tolerance separately at the storage and compute layers, a unified design reduces overhead incurred by replication and concurrency control significantly. The system relies on an object-oriented data model that logically stores functions with their associated data (§3.1).

This holistic design enables LambdaStore to provide low latency for function calls and to guarantee strict serializability across entire serverless workflows (§4.5). To maintain the scalability and elasticity that is crucial in the cloud setting, we rely on two mechanisms: First, the system adjusts the storage layout of an application depending on the workload it experiences. It spreads application objects across a variable number of shards or adjusts the lock granularity within an object if needed (§4.4). Second, microsharding allows migration of individual objects between shards to quickly accommodate workload changes.

LambdaStore outperforms the conventional disaggregated serverless design in both throughput and latency while providing stronger consistency. We compare it against Faasm [37], OpenLambda [30], Apiary [25], and OpenWhisk [17] using microbenchmarks to demonstrate its performance benefits. We find that our approach performs excellently, delivering high throughput and low latency, and is often orders of magnitude better than existing platforms. Our experiment results show that LambdaStore is able to achieve over 20× the throughput of container-based solutions for functions accessing data. LambdaStore is able to process twice as many request than other implementations based on WebAssembly, while delivering the same throughput. We also evaluate the system under two application workloads – an online message board and a microblogging application – to demonstrate its scalability. The system has strictly lower end-to-end latencies than all other systems we evaluated. We find that performance scales well, linearly with the number of shards, processing up to 500k transactions/s with 12 shards, where each transaction contains one or multiple serverless function invocations.

2 Background and Motivation

Serverless platforms enable mutually-distrusting applications to execute on the same cluster, and even one the same machine, using *virtualization* and *dynamic resource assignment*. First, virtualization isolates individual jobs from each other, guaranteeing that they do not starve other jobs of resource, and enusres that jobs that misbehave due to a software bug or a malicious developer cannot affect other jobs. As a result, virtualization guarantees high fault resiliency and permits the safe co-location of mulitple applications on the same hardware. Second, dynamic resource assignment allows for the (re-)allocation of resources to applications at runtime. This mechanism ensures high resource efficiency and enables scalability by, ideally, allocating the minimum amount of resources that is sufficient for the currently experienced workload to an application.

Serverless systems then provide an abstraction for developers to build applications without explicitly provisioning any machines, virtual or physical, and without explicitly initializing application processes or services. Applications are instead implemented as a set of functions and a cloud provider (such as AWS Lambda, Microsoft Azure Functions, or Google Cloud Functions) manages their instantiation and execution.

Serverless application logic executes in the form of *workflows* which consist of multiple function calls or *jobs*. A workflow is formed by a client or external application issuing the initial job and those jobs then recursively call other jobs to compose more complex application logic. In some system designs, each job only performs a small task and workflows consist of mulitple jobs [20], but the complexity and runtime of a job varies depending on the application.

2.1 Conventional Serverless

A conventional stateful serverless system architecture usually consists of three components: a coordinating layer, a compute layer, and a storage layer. We outline this architecture based on OpenWhisk [17]. OpenWhisk relies on Apache Kafka [16] to track outstanding jobs. These jobs are then delegated to a compute layer (such as Kubernetes [32]). The serverless application communicates with the dedicated storage layer, e.g., a DBMS or key-value store, to persist state across function invocations.

Existing serverless systems provide an elastic compute environment but cannot support data-intensive applications, especially those that require strong consistency or low latencies. The core problem with the design of contemporary serverless systems is the strict separation between compute and storage layers (i.e., their disaggregated nature), which manifests in two ways.

First, strict disaggregation causes frequent data movement which results in high latencies. For a simple read/write workload, a job first needs to fetch data from the remote storage system, perform the update, and push the new changes back to the storage system. While performing the update itself might only take a few microseconds, fetching and storing the data will take significantly longer. Often this issue is exacerbated

	Isolation Granularity	Elasticity	Serializable Workflows	Cold-Start Latency	Job Execution Guarantee	Replicated Storage
OpenLambda	+ Job	+ Yes	- No	+ <1ms (WASM) - >10ms (SOCK)	- none	N/A
OpenWhisk	+ Job	+ Yes	- No	- >100ms	- at least once	N/A
Faasm	+ Job	+ Yes	- No	+ <1ms	- at least once	+ Yes
Shredder	+ Job	- No	- No	+ <1ms	- at least once	- No
Apiary	- Application	- No	+ Yes	+ <1ms	- at least once	+ Yes
AWS Lambda	+ Job	+ Yes	- No	- >10ms	- at least once	+ Yes
Google Cloud Functions	+ Job	+ Yes	- No	- >100ms	- at least once	+ Yes
Azure Functions	+ Job	+ Yes	- No	- >100ms	- at least once	+ Yes
LambdaStore (This paper)	+ Job	+ Yes	+ Yes	+ <1ms	+ exactly once	+ Yes

Table 1: Comparison of LambdaStore to other serverless architectures

by additional time spent establishing a TCP connection between the virtualized environment and the storage system. While caching can improve performance in this setting, it is hard to predict which data a binary will access; as a result, cache misses are frequent.

Second, the lack of coordination between the storage and compute layers makes providing transactional guarantees across function calls difficult. The storage system, by design, has no notion of jobs and cannot detect that a job aborted or re-executed, which means it cannot easily provide atomicity for job executions or entire workflows. In addition, even if a transaction primitive is provided in this setting, the high latencies caused by disaggregation will cause frequent lock contention. For example, previous work ensured serializability with the help of atomic logging in the disaggregated serverless setting which roughly tripled latencies [39].

As a result, most real-world serverless platforms do not provide transactional guarantees across function calls and many do not even ensure that a job executes exactly once but at least once. For example, AWS Lambda executes functions up to three times [7] if it encounters failures. Similarly, Azure and MongDB require developers to define custom retry policies [28, 29].

2.2 The Missing Pieces

Our goal is to realize a cloud computing infrastructure that is *data-centric* with *unified transaction and job management.* Such a platform can leverage knowledge about the data, its access pattern, and associated functions, to provide an execution layer with low latencies, high throughput, and strong consistency.

Existing mechanisms for user defined functions (UDFs) (such as SQL stored procedures) provide no proper isolation mechanisms. For example, Apiary [25] is a mechanism that aims to provide a serverless infrastructure using stored procedures in VoltDB or PostgreSQL, but does not virtualize individual jobs, requiring a separate database deployment for each application.

A data-centric design must overcome three core challenges. First, it needs a data and execution model that allows the system to anticipate which data a particular function will access. Second, the system must support serializable transactions for generalized programs that have no pre-defined read or write sets. Third, like existing cloud computing systems, it must be sufficiently scalable and elastic. In this paper, we describe LambdaStore, which implements such a data-centric design.

Table 1 outlines how LambdaStore differs from existing approaches. OpenLambda [1, 30], OpenWhisk [17], Faasm [37], Shredder [40], AWS Lambda, Google Cloud Functions, and Azure Functions provide fine-grained virtualization at the job-level, like LambdaStore, but do not provide serializable transactions. Most existing systems exhibit cold start latencies of tens of milliseconds making them not suitable for real-time applications, while Apiary, Shredder, Faasm, and LambdaStore can spawn a new function in under one millisecond. Apiary [25] co-locates storage and execution, and provides serializeable transactions, but, unlike LambdaStore, only provides coarse-grained isolation at the application-level and no straightforward means of dynamically reallocating resources among applications. Shredder also co-locates storage and execution, but provides no mechanisms for replication or sharding.

3 The LambdaObject Abstraction

LambdaStore is a compute-enabled, sharded, and durable storage system that organizes data and execution around *stateful objects*. This section demonstrates the benefits of such model by detailing its abstraction and discussion an example application built with it.

3.1 Data Model

3.1.1 Objects

Each application consists of objects, which can hold data entries and executable code. Encapsulating logic and data within objects allows for an intuitive way to break down an application into small and independent components, similar to how microservices are structured [10]. In addition, LambdaStore uses this notion of objects to determine where data will be located and where associated functions can execute. The notion of objects is designed to be as flexible as possible, supporting different programming languages, objects of varying sizes, and application-specific data types.

Objects are instantiated from *object types*. Object types define the methods and associated functions of an object, and what data fields an object has. They are similar to classes in object-oriented programming. Type information allows to de-duplicate (meta-)data that is shared between objects of the same type.

3.1.2 Applications

Each object belongs to an application. The developer of an application is billed for the storage of all its data and execution of functions belonging to the application. Developers define which functions of an object are part of the application's public API and which can can only be accessed from within the application. Currently, objects can only invoke functions 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 and are stored as (part of) a *field* within a particular object. Data fields then define how entries can be accessed and stored, and how they are indexed. Currently, LambdaStore supports maps, multi-maps, and cells (unstructured data) for its fields. Other data types can be implemented in the application code. For example, applications can implement a set type 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 data field of a particular object. For example, an entry could represent a single item in an object's map or the entire content of a cell. The meaning and content of an individual entry in an object's storage are application-specific and opaque to the datastore. Functions are only exposed to a minimal API that allows reading, writing, and performing simple 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 executes in the form of function calls (or jobs). Jobs represent the invocation of a particular function. As with conventional serverless systems, more complex application logic can be created by composing multiple jobs into a workflow. The graph of jobs within a workflow is not pre-defined but generated dynamically as a function executes.

Functions come in different variants that are similar to those in object-oriented programming. Constructors create a new object and initialize it. Methods access or modify existing objects. Static functions do not have direct access to any data.

In addition, LambdaStore provides a mechanism for certain operations that require touching many objects: *map calls*. For example, one might need to get the maximum age of all clients within an application, where each client is a dedicated object. In this case, execution a function per object is highly inefficient. Map calls take a set of objects O and a function f, and execute at most one instance of f per shard. Each invocation of f then iterates over the subset of O that is located within the specific shard. This enables efficient batch processing without adding data movement. The result of a map call (if any) is aggregated and returned to the caller.

Function calls only have direct access to the data of the object(s) they are associated with. Functions can invoke constructors and methods of other objects, or rely on map calls, to indirectly access or modify their data. This pattern minimizes data movement which we discuss in more detail in Section 4.4.

Functions in LambdaStore are exposed to a minimal API on which more complex application logic can be built. This API allow to read and write object entries, call functions, manage user sessions, retrieve function arguments, set return values, get the current time, and generate randomness. Analogous to system calls in an operating system, which only provide low-level functionality, high-level abstractions are then implemented in user space (or here, within the virtualized environment).

Functions execute until their complete execution or are terminated by the runtime. Termination can

happens due to fatal errors, e.g., stack overflows, violating security policies, e.g., attempting to access data that does not belong to the application, or reaching their maximum execution time. The latter is important because malicious functions might never terminate, starving the system of resources.

3.2.2 Transactions

LambdaStore guarantees strict serializability across an entire workflow. A workflow can be represented directed acyclic graph (DAG), or, more concretely, a directed rooted tree, of function calls. In this graph, a vertex corresponds to a job and an edge corresponds to a job creating another job through a function call. Clients initiate workflows by invoking a public function. This initial function call forms the root of the workflow's DAG, which is then automatically extended as functions call other functions. DAGs are not predefined by the application developer but generated dynamically at runtime.

Developers can inspect the DAG of a transaction using an *execution trace*. They can set varying level of verbosity for traces. By default, traces contain a graph of all function invocation, but developers can request to log every API call and storage access as well.

Function calls by themselves execute sequentially, but workflows can execute multiple jobs concurrently. This is achieved by letting functions issue multiple function calls at the same time and wait for all of the to terminate. For example, in a social network application, a function that creates a user's post might invoke a function on all followers of that user to update their respective timelines. While the initial post creation executes sequentially, all follower timelines can be updated in parallel.

Because each workflow is contained within a single transaction, we use the terms workflow and transaction interchangeably when talking about our system.

3.3 Example Application

LambdaStore supports arbitrary applications which we demonstrate in this section. We implemented: CloudForum, an online discussion board similar to Reddit, as an application in our serverless system. This section outlines part of the implementation in Rust code, with some simplifications due to space constraints.

Listing 1 shows how object types are declared for an application. Here we oultine three types. `Accounts` represent users of the online forum and contain references to all threads created by the users and all comments they made. Comments are referenced by the thread identifier and their index within the thread. `Threads` store an initial post by the thread creator and a sequence of comments. They store comments in a custom structure

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>,
}
```

(`Comment`) which is not an object type itself but defines how data within the comment field is structured. To the storage system, this structre is justs opaque data. Finally, threads are indexed using the `Community` type.

Listing 2 shows the workflow of adding a new comment to an existing thread. It relies on LambdaStore's Rust bindings which hide most boilerplate code such as serialization of data. User's invoke the workflow by calling the `create_thread` method on an object with the `Account` type. They pass arguments, such as the comments content, in a JSON document which the job retrieves using the `get_json_args` call. While LambdaStore also supports binary arguments, JSON makes interacting with client code written in JavaScript easier.

The function then authenticates the user (not shown) and look up the users identifier and name. It then calls the `add_comment` method on the particular thread a comment should be added to, which executes in a dedicated job. When a call is invoked they will execute in the background by default, unless the parent job explicitly calls `join` like shown. This API enables concurrency similar to the thread API in most operating systems: multiple child jobs can be spawned and waited on at the same time.

The `add_comment` method then stores the comment as part of the `Thread` object. It adds time information to the comment using the `get_unix_time` host call. Finally, it returns the comment's identifier, which is then used by the Account object to store a reference to the comment.

4 LambdaStore

This section explains LambdaStore's design in detail. First, we outline the overall architecture of the system. Then, we describe how it allows executing functions on

Listing 2: Implementation of CloudForum's comment functionality. Low-level API calls are hidden behind a higher-level abstraction. Access control and error handling code was 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(thread_id)
        .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});
  }
}
```

a single object using virtualization and dynamically adjusting its granularity. Finally, we describe applications operate on multiple objects and at scale, using sharding and serverless transactions.

4.1 System Architecture

A LambdaStore cluster has four types of participants: clients, frontends, storage nodes, and coordinating nodes. Figure 1 sketches how these components interact with each other.

4.1.1 The Coordinating Service

The coordinating service (or simply "coordinator") is responsible for maintaining all metadata. In particular, it keeps track of all participants in the cluster and the

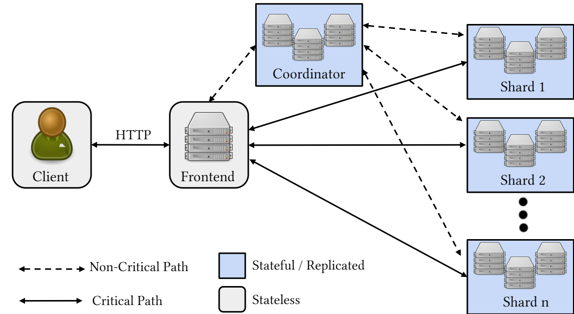

Figure 1: LambdaStore relies on a shared-nothing architecture. A centralized coordinator is only needed during reconfiguration or when encountering failures.

configuration of individual shards. It also tracks on which shards objects are placed. When nodes, frontends, or clients join the network, they will first connect to the coordinating service, which will then inform them about all (other) nodes in the system.

In contrast to conventional serverless systems where all requests pass through some stateful frontend, this service is not involved in most transactions. Transactions only involve the coordinator if they create new objects or when there is an ongoing reconfiguration of the cluster. Reconfiguration occurs when the system performs a load balancing decision (see Section 4.4) or during failures.

4.1.2 Nodes and Replica Sets

For each shard, a set of $f+1$ nodes will replicate its state to tolerate up to f simultaneous failures. Shards rely on chain replication [21] to ensure durability. As the name indicates, nodes in the set are arranged in a chain, where the head represents the primary and all other nodes are secondary replicas. State changes, such as transaction commits, will traverse the chain from the head to the tail. Once the state change reaches the tail, a confirmation message will traverse the chain the other way.

Requests are always sent to the primary first, which then determines where to execute it. When contention is low, it is most efficient to execute jobs directly at the primary. The primary may delegate jobs to other nodes in the replica set during high contention. Section 4.5.2 discusses the latter in more detail.

4.1.3 Clients and Frontends

Clients can directly interact with the datastore through a proprietary protocol or indirectly through a HTTP frontend. In the former case, they connect to the coordinating service which will give them information about other participants in the cluster.

When using the HTTP frontend, clients do not need to maintain information about cluster configuration or

implement a custom protocol. Instead, the frontend acts as an intermediary that maintains an up-to-date view of the cluster configuration and forwards client requests. Frontends are still stateless, in that they can fail without losing any valuable information or corrupting data. The use of HTTP frontends is particularly useful when building web applications or short-lived clients.

4.2 Virtualization Layer

LambdaStore enables untrusted computation to be directly embedded in storage servers through software-based virtualization, specifically through the use of WebAssembly (or *WASM*) [22]. WASM is a bytecode that a high-level language, like C++ or Go, can be compiled to. The WebAssembly runtime can then directly interpret and execute that bytecode or, more commonly, compile it to machine code. LambdaStore takes the latter approach to achieve performance that is close to that of native programs while still protecting against software bugs and malicious code.

We chose WebAssembly as a virtualization mechanism because its overheads are orders of magnitude lower than those of virtual machines (VMs) or containers. While this technology is software-based, recent interest in WASM in the context of blockchains [18, 19] has led to its implementations being thoroughly vetted for security. It is important to point out that a system like LambdaStore could also be built by co-locating containers or VMs on the same machine as the storage node. We found the overheads of these mechanisms outweigh potential benefits such as slightly better isolation or more fine-grained control over CPU access.

After the application developer has compiled their code into WebAssembly it will register it with the coordinator. The coordinator compiles its WebAssembly instructions to machine code and then forwards it to all storage nodes. During compilation, it injects additional code that protects against misbehaving programs. For example, every memory access is guarded by a bounds check, and trap handlers are installed to manage other software failures, e.g., division by zero.

Nodes then directly embed the generated code into their address space. To start a function, a node will assign it some address space using `mmap` and perform a context switch by storing register contents and changing the program counter to some location inside the function's code. The function can interact with the host environment to a predefined set of API calls that perform a context switch back to regular storage code, similar to how system calls switch from user to kernel space.

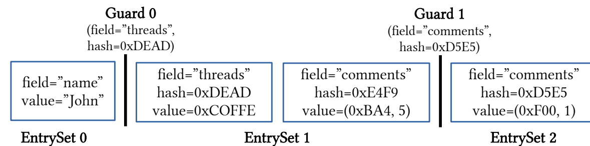

Figure 2: Visualisation of a Account object storage, where the keyspace is partitioned into three sets

4.2.1 Metering

LambdaStore protects against non-terminating functions using periodic timer interrupts. Upon encountering an interrupt, the trap handler will check if the current function has reached its maximum execution time and aborts if it needed. This is a key difference to conventional storage systems which can be stalled indefinitely by faulty stored procedures.

Keeping track of the execution time of a function is also necessary to support a proper billing model in the cloud setting.

4.3 Dynamic Object Granularity

A key challenge in the design of an object-oriented storage system is that interpretation of what data and program logic an object contains is highly application-specific and that application can have objects of varying sizes. For example, one type of object might represent an index of all users within an online forum. Another type of object might represent an individual user. In most cases, the former would contain many more entries and receive more frequent queries and updates.

LambdaStore tracks metadata for each object in order to perform concurrency control. Naively there could always be one set of metadata per object or one set of metadata per entry. The former might be much too coarse-grained in the case of large objects, while the latter would create too much bookkeeping overhead. For context, the system's design aims to support millions of objects, some of which might be extremely popular.

4.3.1 Entry Sets

Entries within an object are grouped into sets to support varying granularity of objects. Initially, an object only consists of a single entry set. As entries get created, LambdaStore probabilistically splits the corresponding entry set into smaller sets.

Each object has a number of *guards* which define where an entry set ends and a new set begins. A guard is simply a key that maps into the keyspace of an object. Figure 2 gives an example of such a partitioning. Here, an instance of the `Account`-type from Section 3.3 is split into three entry sets. The first entry set encompasses the account name and parts of the `threads`-field, the

second the remainder of the `threads`-field and part of the `comments`-field, and the last the remainder of the `comments`-field. Note, that this is simplified and, for example, mechanisms to handle hash collision are not shown.

Entry sets serve as units of locking and version control. Instead of locking entire objects, transactions only involve entry sets they interact with. Each set can be locked individually and its version number increases when written to. For example, in the case of Figure 2, one transaction could modify the account name, while another updates the thread field concurrently.

Version information of entry sets is also used to implement consistent range queries. When a transaction reads a range, it will track the version numbers of all entry sets covered by this range to detect concurrent updates. For example, in the case of Figure 2, a transaction can lock entry sets 1 and 2 to read all contents of the `comments`-field. A limitation of this design is that LambdaStore does not support range queries across multiple objects. Entry sets of the same object are always located within the same shard and there exists are clear order among them, while neither is true for entry sets belonging to different objects. However, transactions can still span multiple objects.

4.3.2 Entry Set Partitioning

With some chance, a write to a key will turn that key into a new guard and, thus, split the entry set it writes to in half. As a result, objects that are written to more often will likely contain more guards and thus be split into more separate entry sets. The intuition behind this design is that only write-heavy workloads benefit from higher lock granularity. A similar mechanism has shown to work well in the context of Log-Structured Merge Trees [31].

Updates to entry sets happen without interrupting the function execution and without violating consistency of the system. Entry sets are modified only during a transaction's commit when the datastore processes the transaction's writes. At this point, all involved entry sets are write-locked by the transaction and no other transaction can modify the sets. For example, in the case of Figure 2, a transaction would have written the entry at `("comments", 0xD5E5)` and inserted the corresponding guard as part of its commit. We discuss later how the transaction protocol also avoids inconsistent reads during this phase.

Our current system never removes guards or, in other words, never merges entry sets, but this can be added in a future version. For this, a transaction would need to lock both entry sets before merging them. Additionally, the system would need to develop a policy that decides when entry sets are no longer needed.

We outline in Section 4.5 how these locks and version numbers are used for concurrency control. A future version of LambdaStore could also leverage these version numbers for multi-version concurrency control or to provide an efficient caching mechanism.

4.4 Sharding Applications

LambdaStore treats objects as *microshards* [8], which ensures objects and all their data get mapped to exactly one physical shard. This has two benefits. First, the system can react to changes in the workload in a more fine-grained manner by migrating individual objects. This is important because objects can vary greatly in size. Second, function invocations are guaranteed to only touch one shard and, thus, avoid copying data from another shard.

Generally, the system tries to map objects of the same application to as few shards as possible to increase locality. Like other serverless systems, LambdaStore can also map objects of different applications to the same shard to maximize resource utilization. Access control and virtualization (see Section 4.2) ensure that applications executing on the same machine cannot interfere with each other.

The coordinator manages the shard assignment. It keeps a mapping from nodes to shards and objects to shards. In addition this mapping is sharded as well and management of the mapping be distributed across multiple physical machines. Each other participant, i.e., nodes and clients, fetch object locations from the coordinator as needed and cache them locally.

Node assignments for replica sets are only changed during failures or when the overall size of the cluster changes. These mappings are very small, only a few hundred bytes per shard, and kept up to date at all participants. The coordinator establishes a TCP connection to all nodes and interprets termination of the connection as a node failures. When it detects a node failure, it will reconfigure the replica set by promoting the next node in the chain to primary and adding a new secondary node to the end of the replication chain.

The object mapping is considerably larger and changes when new objects are created or the workload changes. For example, when a new thread is created in CloudForum, will require the system to create a new object. When someone adds a comment, no new object is created, merely the contents of an existing object are updated. Similarly, if a particular thread is very popular, it may be moved to a different shard.

The coordinating services only pushes mapping updates to nodes that are affected by the change to increase efficiency. Concretely, nodes of the object's

new and old replica sets are notified. Other nodes and clients detect the mapping change in one of two ways if needed. If the location of the object is not cached locally yet, they request it directly from the coordinator. Otherwise, they contact the replica set of the cached location. If the location is still up to date, the request will be processed normally. If the location is outdated, the replica set will respond with a cache miss message in which case the coordinator needs to be contacted.

4.5 Serverless Transactions

Transactions in LambdaStore encapsulate a serverless workflow: They ensure that all functions invocations within the same workflow execute with ACID guarantees.

LambdaStore enforces strict serializabilty using optimistic concurrency control. Transactions are processed in three phases: execute, prepare, and finalize. During the execute phase the function(s) involved in the transaction execute, and the transaction tracks their read and write sets. In the prepare phase write locks are acquired and the transaction checks for conflicts and validates all it reads. During the finalize phase the transaction either commits or aborts. A transaction aborts if in the execution phase the application code requested to do so, there was a fatal error (i.e., the program crashed), object placement information was outdated, or the prepare phase failed.

As outlined in Section 3.2, a workflow consists of multiple jobs all contained within a single transaction A workflow is initiated by a client issuing a request to the LambdaStore cluster in form of a function invocation. This initial function invocation will create a new transaction. When a function calls another function, the latter will inheirt the former's transaction. As a result, a transaction tracks the read and write set of an entire workflow.

4.5.1 Multi-Shard Transactions

LambdaStore supports sharding and cross-shard transactions. Objects can be located on different shards and, as a result, a function invocation might lead to execution on that shard. Alternatively, one could ship the data to the original node and execute all functions there. However, that would limit the processing power of a transactional workflow to that of a single node and defeat the purpose of co-locating storage and computation. We also experienced in early evaluation that such a disaggregated design increases the likelihood of aborts as transactions might execute on stale data.

When executing across multiple shards, a single shard coordinates the transaction. When a job terminates, it will return its output back to the calling job. In addition, it will report the identifiers of any additional shards involved in the transaction, and whether new objects were created. This process happens recursively until the initial job (the root vertex of the workflow's DAG) terminates.

During the prepare phase, the coordinating shard will tell all other shards to prepare and only move to the commit phase if all shard succeeds. If the prepare phase fails the coordinating shard will tell all shards to abort. During the commit phase, the coordinating shard will tell all shards to commit and inform the coordinating service about newly created objects (if any).

4.5.2 Delegated Transactions

A shard's replica set consists of multiple nodes holding an up-to-date copy of the shard's data. LambdaStore leverages this fact and does not only allow execution of jobs at the primary of a shard but at any replica.

A replica set's primary keeps track of which replica is executing which transactions Once a backup has finished processing a transaction it sends its read and write sets to the primary to finalize the transactions. This last step is necessary to uphold serializability. Because of this extra communication and validation step, it is generally more effective to execute at the primary, but delegation can quickly provide more compute power for short-term workload changes or until the system is done migrating objects to adapt to long-term workload changes.

4.5.3 Concurrency Control Protocol

LambdaStore uses a variant of Silo's optimistic concurrency control protocol [38]. During reads in the execution phase, instead of atomically reading the value and corresponding version number atomically, LambdaStore takes advantage of the atomic interface provided by the key-value storage backend and allows each transaction to read a version number staler than the read value. That is, during read, each transaction first records the version number of the corresponding entry set and then queries the storage backend for the value; and since writers do not block concurrent readers, we additionally require the transactions to write each value back before updating its version number during commit phase.

4.5.4 Fault Tolerance

LambdaStore can tolerate crash failures of individual nodes. We refer to a node crashing, losing power, or disconnecting as "failure." Note, that for function execution, the platform supports arbitrary (or Byzantine [26]) failures through its virtualization mechanism, which we discuss in Section 4.2.

Transaction execution supports failures of any node and of the client/frontend that issued the transaction.

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)
```

We assume that clients are unreliable and, after issuing a transaction, they have no role in coordinating a transaction. Clients can merely re-issue a transaction if it failed. Transactions may succeed even if the issuing client has failed.

We generally refer to a shard failure if one of its nodes fails. Each shard has a replica set of $f+1$ nodes, where at most f nodes can fail. Node failures result in a reconfiguration of the affected shard. A shard will reconfigure by restarting or replacing the affected node and the coordinator will notify all other shards about the reconfiguration.

Recovery from Secondary Replica Failures Failures of secondary replicas can be handled by the shard's primary as it will always process transaction prepares and commits first. Once the primary gets informed about a shard reconfiguration, it will re-issue any delegated transactions/jobs that were affected by the failure.

Recovery from Transaction Manager Failures Generally, when the primary fails a new primary will take over. That primary is chosen by the coordinator from one of the non-faulty replicas remaining. This allows the new primary to take over immediately without executing a complex recovery protocol. A new replica then joins the set and synchronizes the state from the other nodes. The primary then will reissue commit requests if needed. State about executing transactions will be lost, but this does not violate atomicity guarantees.

For multi-shard transactions, the manager also needs to consolidate the state at other nodes. Some shards might have prepared the transaction already and need to finalize it in order for locks will be released properly. Additionally, some shards might have finalized the transaction already and we need to make sure to commit/abort transactions properly at all shards to uphold atomicity.

All shards notify the failed shard about prepared transactions that originated from it. If the new transaction manager has logged a commit for the transaction, it will ask the involved shard(s) to commit. If it does not have logged a commit for the transaction, it either aborted or has not finished its prepare phase yet, which means it can be safely aborted.

Recovery from Remote Shard Failures When a shard fails, information about in-progress transactions, including remote transactions, might get lost. For transactions that are still executing, jobs will simply be re-issued. Similarly, for transactions that are still in the process of being prepared, the prepare request will be re-issued.

Transactions that are already prepared or partially finalized need to be finalized consistently across all shards. To do this, the new primary first has to check for any in-progress transactions and then query each transaction's manager whether to commit or abort it. The new primary will always have access to the transactions write set in the case of a commit, as it has been replicated during the prepare phase At this point, it is guaranteed that the prepare phase at every shard succeeded as otherwise the transaction would not have been able to move to the commit phase.

5 Preliminary Evaluation

We now 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 (a framework for asynchronous execution) [12] and Wasmtime (a WebAssembly runtime) [4].

Our evaluation answers the following questions

- What are the benefits and overheads of LambdaStore's co-located design?
- How does object granularity affect performance?
- How expensive is the creation of objects?
- Can LambdaStore handle arbitrary failures of application code?
- What benefit does LambdaStore provide to applications?

5.1 Experimental Setup

We compare LambdaStore with OpenLambda, OpenWhisk, Apiary, and Faasm. Like most open source serverless systems, OpenLambda's SOCK runtime [30] and OpenWhisk rely on containers. While containers are more lightweight than, for example, virtual machines, they still incur a significant overheads. Faasm combines containers and WebAssembly to reduce that overhead. OpenLambda also provides a more efficient WebAssembly runtime [1] against which we evaluate as well. Finally, Apiary allows to execute serverless functions as stored procedures in PostgreSQL or VoltDB. However, unlike LambdaStore, Apiary does not isolate functions executions.

All benchmarks were execute on Cloudlab [14] with a cluster of `c220g5` machines. These machines are equipped with two Intel Xeon Silver 4114 CPUs (each having ten physical cores and Hyper-Threading), 200GB of DDR-4 memory, and 10Gbit NICs.

In our evaluation setup, OpenLambda and OpenWhisk do not perform any replication of client requests or concurrency control. Most other serverless systems contain basic fault-tolerance mechanisms (as outlined in Section 2) that incur additional overheads. For example, Faasm – as of version `0.12` – has a centralized planner component that keeps track of outstanding requests. For Faasm, data is persisted in Minio, a datastore with an API similar to S3, and Apiary uses PostgreSQL to store data. We use LambdaStore as a storage backend for all other setups.

In the OpenWhisk setup, each compute machine runs a standalone OpenWhisk instance which encapsulates all components including 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 results of each lambda invocation. Each client machine runs a light-weight client frontend that evenly distributes requests among the compute instances to balance load. We note that our setup deviates from traditional OpenWhisk setups where a dedicated controller node manages a set of invoker nodes, but the experimental results presented in our setup agree with its performance presented in other recent work [25].

For Apiary we instantiate one "worker" process on one of the storage node. This process merely forwards function invocations to PostgreSQL. In addition, we instantiate a HTTP frontend on each client machine. The frontends take in client requests and talk to the worker process using Apiary's proprietary protocol. Data is replicated by configuring two backup nodes for each PostgreSQL database.

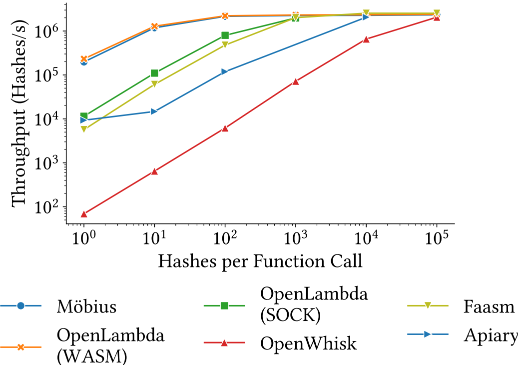

Figure 3: Comparison of the virtualization and coordination overhead of different serverless systems, which are especially visible for short-running functions

5.2 Microbenchmarks

5.2.1 Disaggregation and Coordination Overheads

Figure 3 shows the overheads introduced by comparing jobs of different lengths. Virtualization and coordination overheads are mostly independent of the job length, so they will be less significant for longer-running jobs. This benchmark computes a number of SHA512 hashes of 1kb input data. We gradually increase the number of hashes computed per function call to generate longer-running functions (shown on the x-axis). Once the overhead is sufficiently amortized overall throughput (in hashes per second, shown on the y-axis) for all workloads is about the same. Note that all observed numbers are warm starts as the same function will be invoked repeatedly. In all cases, we use a single machine to execute functions: either a dedicated compute nodes for the disaggregated designs and a single store node for the co-located design.

We observe that the overhead generated by containers, both for our standalone implementation and for OpenWhisk, initially outweighs that of WebAssembly by multiple orders of magnitude. At on hash per function call, LambdaStore is able to achieve 798× the throughput of OpenWhisk. It also outperforms Faasm and OpenLambda's SOCK runtime significantly. OpenLambda's WebAssembly runtime initially outperforms LambdaStore slightly, because the latter has needs to perform additional coordination work before delegating the function. This work is needed to ensure serializability in case the function does access data, which cannot be predicted ahead of time for generalized functions. This overhead is amortized at about 1k hashes per function call, roughly 80ms of function execution time. All workloads converge at about 100k hashes per function call or roughly 800ms of execution time.

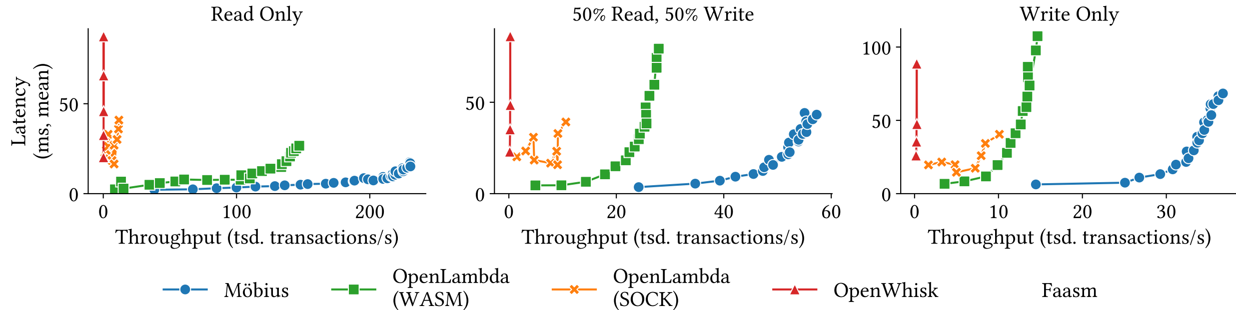

Figure 4: Comparison of latencies for read and write workloads. LambdaStore's outperforms other system in all cases.

Write Chance	0%	25%	50%	75%	100%
Möbius	230487	80401	57293	44045	36748
OpenLambda (WASM)	147096	41005	27898	20098	14652
OpenLambda (SOCK)	11719	10902	10610	10392	10109
OpenWhisk	282	276	275	283	278
Faasm	242	204	306	422	16922

Table 2: Comparison of throughput in read and write workloads

5.2.2 Throughput and Latency

We evaluate LambdaStore under a number of microbenchmarks to show the overhead generated by coordination as well as the benefit of colocation. Here, we pre-load the storage system with with 10000 objects, each with 100 entries of 1kb size, resulting in a total of one million entries stored at the beginning of the experiment.

We ran the microbenchmarks with on a set of six nodes for the disaggregated setting (Apiary and LambdaStore) and three nodes for the aggregated setting (all other systems). In both cases three nodes form a storage shard and, for the disaggregated setting, an addition three nodes are allocated to execute computation. For Faasm, we set up a Kubernetes cluster with six machines to execute workers, planners, and storage.

Table 2 shows the performance of LambdaStore and its baselines under different read, write, and mixed workloads. We varied the number of clients and only show the results that performed best for each individual configuration. LambdaStore significantly outperforms all other systems. Note that writes are generally slower because the require replication across storage nodes.

For Faasm we observed much lower write and entry creation performance, compared to updates. As a result, we executed this workload with only 100 objects to keep startup times for the experiments manageable. Each entry and object metadata is mapped to a single entry in the underlying blob storage. We observed write performance that is orders of magnitudes higher than read when evaluating this system. Note that Faasm allows to explicitly synchronize (or "push") writes from the worker to the storage. Our benchmark did not perform this operation and let Faasm decide when and whether to synchronize writes, which explains the higher performance when writing as compared to reading. Finally, for Faasm we batched up to 50 requests as recent versions include a centralized planner that performs much better with batching than individual requests.

The benefits of co-location are even more visible when we take latencies into account as shown in Figure 4. In this experiment, we vary the number of concurrent requests to change the total throughput and plot the mean latency as a function of the total throughput.Note that we do not plot latencies for Faasm as they exhibited high variance ranging anywhere from 10milliseconds to over one second. This variance can most likely be attributed to the centralized planner component that all request batches go through. LambdaStore does not only scale to higher performance but also has strictly lower latencies than the baselines we evaluated.

5.3 Application Performance

We evaluated end-to-end performance using two different application workloads. We first evaluate CloudForum, an online message board similar to Reddit that we outlined in Section 3.3. Here discussion is grouped into threads each consisting of a number of posts. Each thread is part of a community (or "subreddit"). We pre-load the system with 100 communities

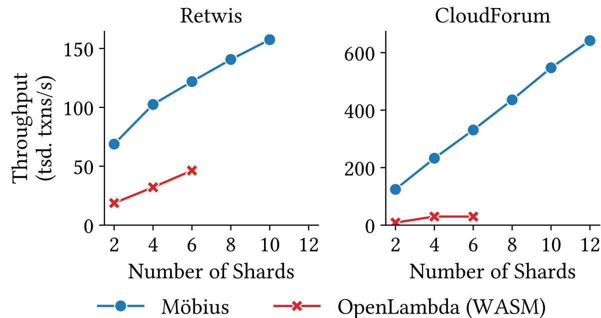

Figure 5: Scalability of applications executing on LambdaStore with up to 12 shards. Baseline performance is shown for up to six shards as it uses twice as many servers.

consisting of a total of 100,000 threads. Each user account for each client thread. The number of concurrent client threads scales with the number of shards.

This workload consists of the following transaction types with their share of all transactions in parentheses. **get-thread (90%)** fetches the contents of a thread including all its posts. This is a read-only query consisting of a single function call. **add-comment (10%)** extends an existing thread with a new post. This will update the affected thread, and the poster's account, each performed by a separate function call.

The second application we evaluate the storage system with is Retwis [33]. This benchmark simulates a Twitter-like social network where a user can follow other users, create posts, and get its "timelines" which consist of the most recent posts from its followers and themselves. We pre-load the system with 500,000 users, each has 10 followers and 10 posts on its timeline to avoid network bottlenecks. Each post is 1024 bytes long.

This workload consists of the following transaction types. **get-timeline (90%)** fetches the most recent 50 posts on the timeline of the calling user. This is a read-only query consisting of a single function call. **create-post (10%)** adds a new post to the post owner's and its followers' timelines. Since we pre-loaded each user with 20 followers, this transaction updates 21 objects. Using LambdaStore's `map` functionality, this can be

The results are presented in Figure 5 LambdaStore scales linearly with the number of shards. While the disaggregated design using OpenLambda also scale with the number of shards, it yields much lower performance while using twice as many servers. In addition we provide latency results in Figure 6.

6 Conclusion

We described LambdaStore, a new system that combines storage and execution for data-intensive applications.

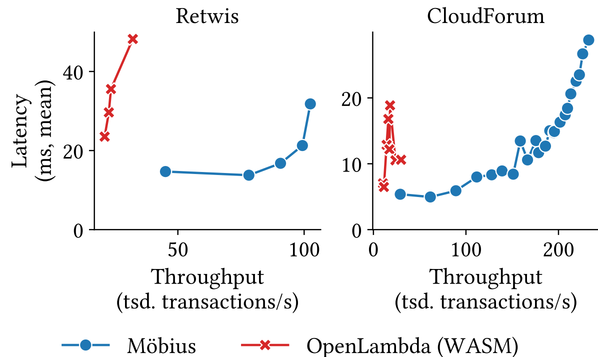

Figure 6: Per-Application latency for a configuration with four shards.

LambdaStore is a the missing middleground between rigid microservices and elastic but slow serverless systems. It provides better performance, improved fault tolerance, and stronger consistency than existing systems.

We view the system as presented as an important step towards interactive and resource efficient serverless applications. Future work will investigate full-stack frameworks to efficiently build applications with little code on top of LambdaStore.

Acknowledgements This is a long running project that started at the University of Wisconsin-Madison. LambdaStore would not have been possible to build without the help of Suyan Qu, Aditya Jain, Pu Guo, Mayur Choudhary, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau.

References

[1] OpenLambda WebAssembly Worker. https://github.com/open-lambda/open-lambda/tree/main/wasm-worker (Last Accessed October 2023).

[2] Aamazon. AWS Knowledge Center: How do I make my Lambda function idempotent? https://repost.aws/knowledge-center/lambda-function-idempotent (Last Accessed May 2023).

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. *Symposium on Networked System Design and Implementation,* pages 419-434, Santa Clara, California, February 2020.

[4] Bytecode Alliance. Wasmtime. https://wasmtime.dev/ (Last Accessed April 2023).

[5] Amazon. Learning Serverless (and why it is hard). https://pauldjohnston.medium.com/learning-serverless-and-why-it-is-hard-4a53b390c63d (Last Accessed April 2023), 2022.

[6] Amazon. S3 Cloud Object Storage. https://aws.amazon.com/s3/ (Last Accessed March 2022).

[7] Amazon Web Services, Inc. Error handling and automatic retries in AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html (Last Accessed March 2024).

[8] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. *Symposium on Operating System Design and Implementation,* pages 445-460, Carlsbad, California, October 2018.

[9] Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. Serverless Computing: Current Trends and Open Problems. *Research Advances in Cloud Computing,* pages 1–20, 2017.

[10] Netflix Technology Blog. Netflix Platform Engineering — we're just getting started. http://netflixtechblog.com/neflix-platform-engineering-were-just-getting-started-267f65c4d1a7 (Last Accessed March 2022).

[11] Cloudflare, Inc. "Why use serverless computing?". https://www.cloudflare.com/learning/serverless/why-use-serverless/ (Last Accessed April 2023).

[12] The Tokio Contributors. tokio-uring. https://github.com/tokio-rs/tokio-uring (Last Accessed January 2023).

[13] Data Dog. The State of Serverles. https://www.datadoghq.com/state-of-serverless/ (Last Accessed May 2023).

[14] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. *USENIX Annual Technical Conference,* pages 1-14, Renton, Washington, July 2019.

[15] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. Serverless Applications: Why, When, and How? *IEEE Softw.,* 38(1):32–39, 2021.

[16] Apache Software Foundation. Apache Kafka. https://kafka.apache.org/ (Last Accessed January 2023).

[17] Apache Software Foundation. OpenWhisk Architecture. https://cwiki.apache.org/confluence/display/OPENWHISK/System+Architecture (Last Accessed January 2023).

[18] Ethereum Foundation. Ethereum flavored WebAssembly (eWASM). https://github.com/ewasm (Last Accessed January 2023).

[19] Web3 Foundation. Polkadot Wiki: WebAssembly. https://wiki.polkadot.network/docs/learn-wasm (Last Accessed January 2023).

[20] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy,

Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. *International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 3-18, Providence, Rhode Island, April 2019.

[21] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis E. Shasha. The Dangers of Replication and a Solution. *SIGMOD International Conference on Management of Data,* pages 173-182, Montréal, Canada, June 1996.

[22] W3 WebAssembly Working Group. WebAssembly Specification. https://webassembly.org/specs/ (Last Accessed March 2022).

[23] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless Computing: One Step Forward, Two Steps Back. *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings,* 2019.

[24] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding Ephemeral Storage for Serverless Analytics. *USENIX Annual Technical Conference,* pages 789-794, Boston, Massachusetts, July 2018.

[25] Peter Kraft, Qian Li, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Danny Cho, Jason Li, Robert Redmond, Nathan W. Weckwerth, Brian S. Xia, Peter Bailis, Michael J. Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. Apiary: A DBMS-Backed Transactional Function-as-a-Service Framework. *CoRR,* abs/2208.13068, 2022.

[26] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems,* 4(3):382-401, 1982.

[27] Marketsandmarkets Private Ltd. Serverless Architecture Market. https://www.marketsandmarkets.com/Market-Reports/serverless-architecture-market-64917099.htm (Last Accessed May 2023).

[28] Microsoft, Inc. Azure Functions error handling and retries. https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-error-pages (Last Accessed March 2024).

[29] MongoDB, Inc. Handle Errors in Functions. https://www.mongodb.com/docs/atlas/app-services/functions/handle-errors (Last Accessed March 2024).

[30] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. *USENIX Annual Technical Conference,* pages 57-70, Boston, Massachusetts, July 2018.

[31] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. *Symposium on Operating Systems Principles,* pages 497-514, Shanghai, China, October 2017.

[32] Redis. Kubernetes Documentation. https://kubernetes.io/docs/home/ (Last Accessed January 2023).

[33] Redis. Retwis Documentation. https://redis.io/docs/reference/patterns/twitter-clone/ (Last Accessed March 2022).

[34] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems,* 2(4):277-288, 1984.

[35] Amazon Web Services. AWS Lambda. https://aws.amazon.com/lambda/ (Last Accessed March 2022).

[36] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Irfan Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. *USENIX Annual Technical Conference,* pages 205-218, Virtual, Anywhere, July 2020.

[37] Simon Shillaker and Peter R. Pietzuch. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. *USENIX Annual Technical Conference,* pages 419-433, Virtual, Anywhere, July 2020.

[38] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. *Symposium on Operating Systems Principles,* pages 18-32, Farmington, Pennsylvania, November 2013.

[39] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. *Symposium on Operating System Design and Implementation,* pages 1187-1204, Banff, Canada, November 2020.

[40] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. *Proceedings of the ACM Symposium on Cloud Computing,* pages 1–12, 2019.