FaasKeeper: a Blueprint for Serverless Services

Marcin Copik
marcin.copik@inf.ethz.ch
ETH Zürich
Switzerland

Alexandru Calotoiu
ETH Zürich
Switzerland

Konstantin Taranov
ETH Zürich
Switzerland

Torsten Hoefler
ETH Zürich
Switzerland

Abstract

FaaS (Function-as-a-Service) brought a fundamental shift into cloud computing: (persistent) virtual machines have been replaced with dynamically allocated resources, trading locality and statefulness for a pay-as-you-go model more suitable for varying and infrequent workloads. However, adapting services to function within the serverless paradigm while still fulfilling requirements is challenging. In this work, we introduce a design blueprint for creating complex serverless services and contribute a set of requirements for efficient and scalable FaaS computing. To showcase our approach, we focus on ZooKeeper, a centralized coordination service that offers a safe and wait-free consensus mechanism but requires a persistent allocation of computing resources that does not offer the flexibility needed to handle variable workloads. We design FaaSKeeper, the first coordination service built on serverless functions and cloud-native services. *FaaSKeeper provides the same consistency guarantees and interface as ZooKeeper with a price model proportional to the activity in the system.* In addition, we define synchronization primitives to extend the capabilities of scalable cloud storage services with consensus semantics needed for strong data consistency.

1 Introduction

In recent years, the cloud gained a new paradigm: Function-as-a-Service (FaaS) computing. FaaS combines elastic and on-demand resource allocation with an abstract programming model where stateless functions are invoked by the cloud provider, freeing the user from managing the software and hardware resources. The flexible resource management and a pay-as-you-go billing system of serverless functions are the next step in solving the problem of low server utilization caused by resource overprovisioning for the peak workload [1–3]. These improvements come at the cost of performance and reliability: functions are not designed for long computations, and they require storage support to backup state and communicate between invocations. Nevertheless, stateful and complex applications can benefit from serverless services [4], and storage services have started to adapt on-demand offerings to handle infrequent workloads more efficiently [5–7]. Just like the elasticity of databases was considered challenging due to increased configuration difficulty and performance overheads [8], consistency protocols and coordination services have not yet found a way to benefit from the serverless revolution.

Apache ZooKeeper [9] is a prime example of a system that has found its way into many cloud applications but is not available today as a serverless service. Zookeeper is used by distributed applications that depend on coordination services to control the shared state and guarantee data consistency and availability. In comparison to cloud-native key-value storage, ZooKeeper provides additional semantics of total order with linearizable writes, atomic updates, and ordered push notifications (Table 2, page 2). ZooKeeper consists of an ensemble of provisioned data replica servers with a single leader, and it has been used to implement distributed configuration, synchronization, and publish-subscribe services [9–14].

R1 Low-latency Invocations (§ 2.2)	**R2** Safe Asynchronicity (§ 2.2)
R3 Synchronization Primitives (§ 2.2)	**R4** Fast Streaming Queues (§ 4.3)
R5 Stateful Functions (§ 4.4)	**R6** Partial Storage Updates (§ 4.4)
R7 Outbound Channels (§ 4.5)	

Table 1. Serverless requirements for complex applications.

Modern services are expected to match the temporal and geographical variability of production workloads [15–17]. Workloads are often bursty and experience rapid changes as seen in systems utilization: the maximum can be multiple times higher than even the 99th percentile [15, 18]. However, the static ZooKeeper architecture and allocation of compute resources make the readjustment to the burst workload difficult, forcing users to overprovision resources for the peak. A *serverless* service with the same consistency guarantees would offer an opportunity to consolidate variable workloads, helping both users and cloud operators to increase their efficiency. Unfortunately, the path to serverless for stateful and distributed applications such as ZooKeeper is not clear due to the restricted and vendor-specific nature of FaaS computing.

In this paper, we present the blueprint for serverless applications to answer the challenges of designing complex serverless services that scale with workload variability. First, we establish **design principles** (Sec. 3). We follow the previously established conventions of decoupling system from application state and compute from storage tasks [19–21], Then, we use ephemeral and fine-grained serverless functions to implement computing tasks in parallel and employ auto-scalable cloud storage to achieve the reliability of serverless processing. We formalize the types of serverless functions and introduce **serverless synchronization primitives**. Our serverless design should be **cloud-native and deployable** to clouds today. Thus, we consider only auto-scalable cloud services with the pay-as-you-go billing, freeing users from the responsibility of provisioning resources and managing self-hosted services. Our guidelines focus on the semantics of services and abstract away programming differences, helping design **cloud-agnostic** systems that are easily portable between clouds (Sec. 3.2). We complement the discussion with a list of **serverless requirements** (Table 1) for complex applications such as ZooKeeper. These future goals present features that platforms must offer fulfill to support many classes of stateful and distributed serverless applications (Sec. 8).

To showcase our design principles, we port the stateful and distributed ZooKeeper consensus service to a fully serverless model. We introduce **FaaSKeeper**, the first coordination service with

serverless scaling and billing model, and demonstrate how our design principles help to map a complex application into commercial FaaS platforms. In FaaSKeeper, we combine the best of two worlds: the **ordered transactions** and **active notifications** of ZooKeeper with cloud storage's **elasticity** and **high reliability** (Table 2). Standing on the shoulders of the ZooKeeper giant, we show how the consensus can be implemented as a FaaS application (Sec. 4) that offers fast data access and the pay-as-you-go cost model while upholding consistency and ordering properties (Sec. 6). To the best of our knowledge, we present the first complex serverless service that offers the same level of service as its IaaS counterpart and retains state without provisioned resources.

We implement a prototype of the provider-agnostic system on AWS, one of the major cloud providers (Section 4). We make the following contributions:

- A blueprint for serverless services that composes functions with cloud queue and storage services to support synchronization, message ordering, and event-based communication.
- The first fully serverless coordination service that benefits from elastic billing, with a provider-agnostic design and an open-source prototype implementation for AWS.
- A serverless implementation of ZooKeeper consistency model with a compatible interface, that achieves up to 37,5 times lower costs on infrequent workloads.

2 Function-as-a-Service (FaaS)

While serverless systems differ between cloud providers, they can be reasoned about with a simple and abstract platform model. Cloud storage is necessary for FaaS to retain state and implement reliable computing, but the various services have different performance, consistency, and cost. In the following, we will first provide an overview of FaaS systems, and then introduce a set of building blocks we envision as fundamental in the design of serverless services.

2.1 Background

Serverless functions complement the computing and storage solutions in the cloud by providing an elastic and dynamic execution engine for fine-grained tasks. Instead of persistent virtual machines and containers, users deploy stateless functions executed on dynamically allocated resources. The software and hardware stack management becomes the sole responsibility of the cloud provider, and the users are charged only for the time and resources consumed during the function execution (*pay-as-you-go*). In place of the cloud resource management and orchestration systems, functions offer various *triggers* to process internal cloud events and external POST requests (1). A function scheduler (2) selects the server responsible for the execution [22], and the function is processed within an isolated sandbox on a multi-tenant server (3). A *cold* invocation occurs when a new sandbox is allocated to handle the execution. The cloud scheduler aims to increase the performance by co-locating concurrent and consecutive invocations, as *warm* invocations in an existing sandbox are faster and can reuse cached resources.

The users adopt the FaaS computing model to benefit from improved efficiency on irregular workloads. The major cloud operators provide serverless platforms [23–26], as they benefit from increased resource utilization when running fine-grained functions on multi-tenant and oversubscribed servers. Nevertheless, serverless *ain't all sunshine and rainbows*. The main issues reported are high-latency

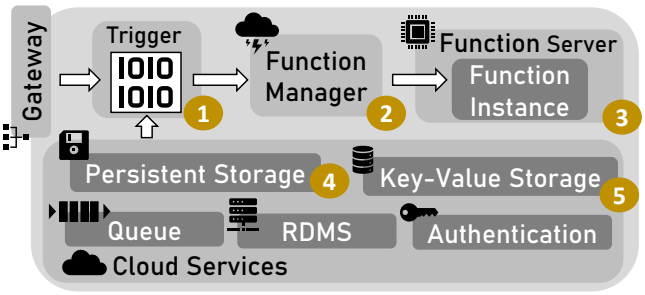


Figure 1. A high-level overview of a FaaS platform.

invocations, huge cold starts overheads, low data locality, variable and inconsistent performance and I/O bandwidth, and high costs on compute-intensive tasks [27–29]. Furthermore, due to the stateless nature of functions, message passing between invocations [30] and concurrent functions in distributed applications [31] requires a persistent data storage.

Cloud Storage. The cloud operators offer various storage solutions that differ in elasticity, costs, reliability, and performance.

Object. The persistent object storage is designed to store large amounts of data for a long time while providing high throughput and durability (4). Prime examples include AWS S3 [32], Azure Blob Storage [33], and Google Cloud Storage [34]. The cloud provider manages data replication across multiple instances in physically and geographically separated data centers. Modern object stores offer strong consistency on the read operations [35], guaranteeing that results of a successful write are immediately visible to other clients. The service offers high availability and data reliability, with usual annual SLAs of 2 and 11 9's, respectively. The users are charged a linear fee for the amount of data stored and a fixed fee for operations performed, creating a genuinely elastic billing model.

Key-Value. The second type of cloud storage common in serverless applications is a nonrelational database (5). Key-value storage, such as AWS DynamoDB [36] and Azure CosmosDB [37], easily matches resource provisioning to the application's expectation for throughput and availability. The service availability of two or three 9's is usually guaranteed, and automatic backups can be used to ensure data durability. In addition to the strong consistency, the read operations can be executed with eventual consistency [38], where updates are guaranteed to be visible *eventually*. The weaker consistency of those operations is compensated by lower costs, latency, and higher availability. Furthermore, the storage can offer optimistic concurrency through *conditional* updates and modification requests that apply simple operations to existing attributes. In addition to provisioned database instances, the users can opt-in for a serverless billing model where the costs depend only on the stored data and operations performed.

Other. Other cloud storage examples are in-memory caches, distributed file systems, and self-hosted data stores. A particular example in the serverless world is *ephemeral* storage [39, 40], designed to match serverless applications' high scalability and low latency requirements while supporting short-term storage and flexible billing policies. FaaS applications can employ these storage systems, but they can require resource provisioning that significantly limits the flexibility of resource allocation and billing.

2.2 Serverless Components

We define serverless function models and propose synchronization primitives necessary for serverless services, including systems such

	ZooKeeper	**Cloud Storage**	**FaaSKeeper**
Scaling	Semi-automatic, min. 3 VMs	**Automatic**	**Automatic**
Cost	Pay upfront, 24h	Pay upfront, **pay-as-you-go**	**Pay-as-you-go**
Consistency	**Linearized writes**	Strong or eventual consistency	**Linearized writes**
High Availability and Reliability	Requires large deployment size.	**Relies only on the provider's SLA and automatic replication.**	**Relies only on the provider's SLA and automatic replication.**
Push Notifications	**Ordered watch events.**	None.	**Ordered watch events.**
Concurrency	**Sequential nodes**, conditional updates (version)	Conditional updates (simple expressions)	**Sequential nodes**, conditional updates (version).
Shutdown Processing Units	Not possible.	**Pay only storage fees**	**Pay only storage fees**
Failure Handling	**Sessions with ephemeral nodes**	None	**Sessions with ephemeral nodes**

Table 2. FaaSKeepervs ZooKeeper and cloud key-value storage: summary of features, requirements and limitations.

as FaaSKeeper. We do not limit ourselves to features currently available on commercial platforms. Instead, we propose characteristics and extensions that cloud providers could offer to enable building more efficient and robust serverless services.

Functions. We specify three distinct classes of functions that are necessary to implement a complex serverless service with multiple responsibilities. While each type provides the same elastic and abstract execution environments, the functions have divergent interfaces and fault-tolerance models and their semantics express different programming language constructs.

Free function. In a truly serverless fashion, this function is not bound to any cloud resource. It can be invoked synchronously anytime, by anyone, from any location, as long as the authorization succeeds and the constraints on concurrent invocations are satisfied. In practice, the most common invocation methods are API requests, and the result is returned as a response. Alternatively, functions can be invoked via RPC and RDMA connections [41]. Free functions express the semantics of *remote procedure calls* [42]: we interpret the application state in cloud storage as its address space and lift the restriction that the execution location must be known a priori to the caller. While the fault tolerance of non-idempotent remote calls is a difficult problem [43], we require the function to be neither side-effect-free nor idempotent, as it would put too many constraints on modern serverless functions with little transactional support. Thus, the caller is solely responsible for error handling.

Event function. The event-driven programming paradigm can be implemented by supplying functions to react to specific cloud events, such as new files, rows, and items in object storage, key-value database, and queue, respectively. Furthermore, a cloud service acting as a proxy allows preserving ordering and passing larger payloads to the function. From the client's point of view, sending a message to a queue-triggered function replaces passing requests over a TCP connection to a server of a non-serverless service. The cloud service triggers function invocations, and we expect the concurrency and batching of invocations to be configurable by the user, as the correctness and performance of the applications depend on it. Semantically, we interpret those functions as remote *asynchronous callbacks* to events in the system. Function executions can be co-located with queue and storage services when their computational workload is negligible compared to the invocation latency [44].

Requirement #1: Low invocation overhead. Invocation overheads dominate the execution time of short-running functions [28] and prohibit FaaS event processing with performance comparable to non-serverless applications. Thus, cloud operators should minimize the critical path of an invocation, and functions should be co-located with the source of the event as frequently as possible.

Scheduled function. Functions can be launched to perform regular routines such as garbage collection, fault detection, and discovery of clients that are no longer online. Such functions can be considered the serverless counterpart of a `cron` job in the family of Unix-based operating systems. This function class is necessary, as not every system change can be represented as a cloud event. In the event of an unexpected failure, the cloud should provide a default retry policy with a finite number of repetitions. Furthermore, if repeated failed invocations do not lead to a recovery, the cloud should report the problem to the user.

Requirement #2: Independent asynchronous invocations. The user cannot directly control asynchronous function invocations. We envision that this should be solved via user-defined *exception handlers*, allowing for unconstrainted error handling.

Synchronization Primitives. Concurrent operations require fundamental synchronization operations to guarantee safe state modifications [45]. Serverless needs synchronization primitives that operate on the shared storage instead of the shared memory.

A **timed lock** prevents concurrent modifications of the same item by serverless workers. It acts like a regular lock except for a limited holding time, a necessary feature to prevent system-wide deadlock caused by a failure of an ephemeral function. To *acquire* a lock on an item, users submit a timestamp stored in the item afterward. The lock is acquired if no timestamp is present or the difference between existing and new timestamp is larger than a user-defined maximum time, extended by 2 seconds to account for clock drift between functions. If the lock is not acquired, the operation returns immediately and can be repeated. When applying an update on a locked node, the stored timestamp is compared against user value to prevent accidental overwrite after losing the lock to another function. The lock *release* removes the timestamp.

An **atomic counter** supports atomic updates using standard arithmetic operations. Finally, an **atomic list** supports a single-step and safe expansion and truncation.

Requirement #3: Synchronization primitives. To become efficient for parallel applications, serverless needs concurrency and synchronization primitives - locks, atomic operations, monitors.

Challenge	Solution
System State	Total order through globally synchronized timestamps.
System Storage	Tables in key-value storage.
Data Storage	Object storage indexed by node path.
Read Requests	Clients access the storage directly.
Write Requests	Cloud queue between clients and writer functions.
Linearizability	Client uses a single FIFO queue.
Single Sys. Image	Strongly consistent storage.
Ordered Notifications	Active notifications define the epoch counter, client verifies that data does not involve awaited notification.
Parallel Updates	Multiple queues (but one per client), nodes are locked to prevent concurrent updates.
Ephemeral Nodes	Periodic invocation of *heartbeat* functions.
Cond. Updates	Writer functions apply user-provided condition.

Table 3. Summary of design requirements and solutions.

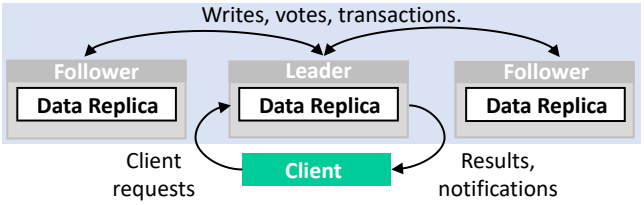


Figure 2. ZooKeeper distributes data across replicas but propagates all updates to a leader.

3 Designing Serverless Services

In this section, we first provide a short overview of the Zookeeper service. We then identify the critical features and requirements and build a serverless service, FaaSKeeper, to fulfill these using the abstract serverless platform model (Sec. 2.1),

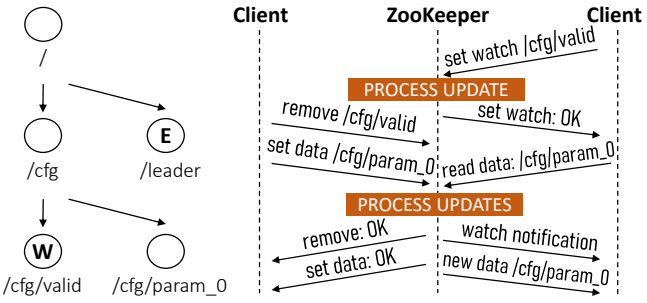
3.1 ZooKeeper

ZooKeeper guarantees data persistence and high read performance by allocating replicas of the entire system on multiple servers [9, 10, 46]. Figure 2 presents a ZooKeeper ensemble consisting of servers with an elected leader, whose role is verification and processing of write requests with the help of an atomic broadcast protocol ZAB [47]. In practice, the rule of $2f + 1$ servers is used: for three servers, two are required to accept change, and a failure of one server can be tolerated. While adding more servers increases the reliability of the service, it hurts write performance.

Consistency Model. ZooKeeper implements sequential consistency guarantees with four main requirements (Z). All operations of a single client are executed **atomically** (Z1), in FIFO order, and writes are **linearized** [48] (Z2). The order of transactions is *total* and *global*. Thus, all clients have a **single system image** (Z3), and they observe the same order of updates. Watch notifications guarantee that clients learn about a change before observing subsequent updates since notifications are **ordered** with read and write operations and other notifications (Z4). A formalization of consistency requirements is provided in appendix A.

Originally, ZooKeeper included an entirely static configuration of the service. Adding more servers to handle application scaling involved *rolling restarts*, a manual and error-prone process [49]. The dynamic reconfiguration [50] has been adopted for the first time in the 3.5.5 release. However, adding servers still requires manual effort [49], and the reconfiguration causes significant performance degradation when deploying across geographical regions [16].

Clients access ZooKeeper through a session mechanism with a selected server that guarantees the first-in-first-out (FIFO) order of



(a) Data structure. **(b)** Watch prevents incorrect reads.

Figure 3. ZooKeeper managing an application configuration.

processing requests. User data is stored in *nodes* that create a tree structure with parents and children. The leader handles updates to nodes, manages the voting process, and propagates changes to other servers. Read requests are resolved using a local data replica. ZooKeeper defines the order of transactions with a monotonically increasing counter *zxid*. While the requests from a single client cannot be reordered, the order of operations between different sessions is not specified. Moreover, sessions control the status of the client-server connection, periodically exchanging heartbeat messages. Finally, ZooKeeper supports push notifications through the concept of *watches*. Clients register watches on a node and receive a notification when that node changes.

Case Study. Figure 3 presents a classic example of storing multi-parameter application configuration in ZooKeeper [50]. The configuration can consist of many parameter nodes `/cfg/param_i`. An update must be visible as a single step and clients must not observe partially changed configuration. partially changed configuration. To that end, the `/cfg/valid` node is used - its existence guarantees that data is safe to read, and users set a watch on this node. When this node is updated, clients receive a *push notification* ordered with other operations and results. The configuration updater sends a pipeline of operations: removing the `valid` node, updating all parameters, and recreating the `valid` node. Thanks to the FIFO order of requests and notifications, clients will receive a watch notification from the deletion before seeing the new parameter data. Furthermore, application processes can select a leader among themselves to manage configuration. The `/leader` node is *ephemeral*, and it is automatically erased upon the termination of its owner's session. With a watch set on this node, other processes immediately learn that they must elect a new leader.

3.2 Design Principles

Figure 4 presents a general workflow that summarizes the steps involved in taking an IaaS service and creating a serverless version with the same functionality. We demonstrate the challenges and application of the workflow to building a serverless ZooKeeper.

ZooKeeper splits the responsibilities across the client library, servers, and the elected leader. Clients send requests to a server they select and wait for replies. Servers forward write requests to the leader, resolve the order of operations for each session, and control the connection status with a client. The leader resolves global ordering, manages the voting process, and propagates changes to its followers. In FaaSKeeper, we redistribute the responsibilities across various cloud services to accommodate its elastic design.

In the following paragraphs, we outline the key observations and design principles for a serverless coordination service.

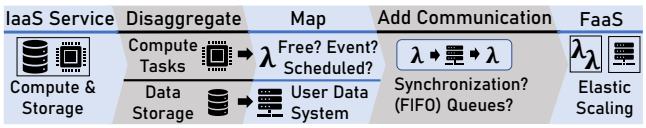


Figure 4. Workflow for designing serverless services.

Disaggregated compute and storage. Decoupling compute and storage is exploited in scalable cloud databases [20, 21], but the benefits of this approach have not been explored in ZooKeeper. Even though the servers manage connections and ordering, their primary responsibility is to provide low-latency read operations that can be replaced with cloud storage solutions. In a coordination system designed for high read-to-write ratios, the compute resources must be allocated more sparsely than data endpoints. In addition, the storage should distinguish between user data and the system data needed to control the ZooKeeper instance: the locality, cost, and consistency requirements of both types are wildly different.

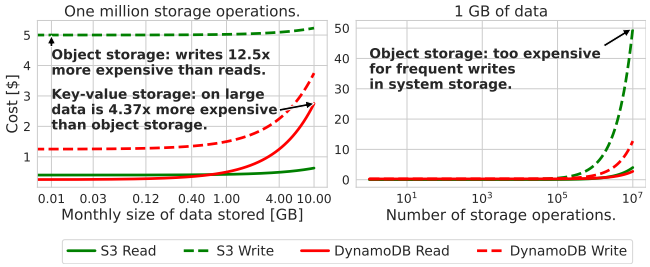
Elastic resource allocation. Serverless computing is centered around the idea of elastic resource allocation, and its primary advantage is the ability to scale the costs down to zero when there is no incoming traffic to the system. To accommodate the temporal and spatial irregularity of workloads [15], FaaSKeeper attempts to scale the resource allocation linearly with the demand. In the case of a *shutdown*, the user should pay only for keeping the data in the cloud. Therefore, we employ a pay-as-you-go billing scheme for the storage and queue services, dependent only on the number of operations performed and not on the resources provisioned.

Direct data access. ZooKeeper servers are responsible for ordering results of reading operations and watch notifications. With cloud storage used as the backend, eliminating intermediaries and direct data access provide the lowest cost and overhead. Thus, FaaSKeeper implements extended timestamps and additional logic in the client library to order read results with notifications.

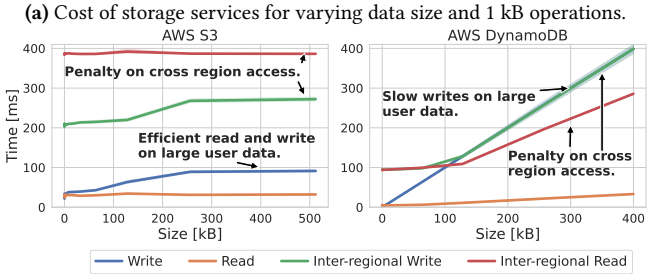
Horizontal scaling. Prior work attempted to increase the write throughput of ZooKeeper through automatic partitioning of the data tree [51, 52]. Instead, FaaSKeeper improves horizontal scaling by increasing the concurrency of functions handling the write process. While write functions must be serialized for a single client, different users can invoke them independently. Thanks to the locking primitives, FaaSKeeper can process write requests concurrently while guaranteeing safe and correct updates of nodes. Thus, the scalability of the process is bounded by the performance of cloud storage. Further performance increases are brought by processing batched write requests in a pipelined manner. Scalability of read operations is achieved using the auto-scaling of cloud storage.

Efficient reading of user data. The size of the user-defined tree node hierarchy can easily exceed a few gigabytes as each node can be up to 1 MB. User data is primarily read, and it requires strongly consistent operations to uphold ZooKeeper's consistency properties. The cost-performance analysis reveals that object storage is more efficient than key-value storage (Fig. 5). Storing large user data is 4.37x cheaper, and updating nodes scales much better with their size. Furthermore, read operations in object storage are billed per access and in 4 kB increments in key-value storage, making it even more expensive for user nodes.

Efficient modifications of control data. The FaaSKeeper's control data includes frequently modified watches, client and node status, and synchronized timestamps. The system must use atomic timestamps and locks when concurrent updates are allowed. The very high cost of frequent write operations (Figure 5a) and the lack



(a) Cost of storage services for varying data size and 1 kB operations.



(b) Latency of read and write operations in AWS storage services.

Figure 5. Storage microbenchmarks in AWS.

of synchronization primitives make object storage unsuitable for the control data. Thus, we use key-value storage instead.

User data locality. Cloud applications balance the resource allocation across geographical regions to support changing workloads [16, 53]. In addition, they aim to minimize the distance between the service and its users, as a cross-region network transmission adds significant performance and cost overheads (Fig. 5b). The automatic reconfiguration of ZooKeeper across regions causes temporal performance degradation, and the mitigation technique requires the live migration of a virtual machine [16]. In FaaSKeeper, we propose replicating data storage across a set of specified regions. Clients connect to the closest auto-scaling storage service in the same region, minimizing access latencies. Control data is placed in a pre-defined *leader* region only, as managing cross-regional modifications is non-trivial, and the replication of the control plane is not the priority in FaaSKeeper. The write requests are forwarded to the FaaSKeeper deployment in the *leader* region.

Cross-regional replication. The updated contents of data nodes must be made available in all replicas after changes to the FaaSKeeperstate. To that end, we could use the cross-region replication of cloud service. However, the latency of automatic replication in the object storage is in the order of minutes and even hours [54].

Cloud Agnosticity The vendor lock-in [55] is a serious limitation in serverless [56, 57], and dependency on queueing and storage services is of particular concern [58]. FaaS applications targeted for a specific cloud often include provider-specific solutions in their design, requiring a redesign and reevaluation of the architecture when porting to another cloud. In a *cloud-agnostic* design, we aim to abstract programming differences and allow moving between providers without a major system overhaul [59]. We define only the requirements on each used service and introduce new abstractions such as synchronization primitives to encapsulate cloud-specific solutions. We specify expectations on serverless services at the level of semantics and guarantees. This limits our exposure to vendor lock-in to the implementation layer. Later, we evaluate how available cloud services fulfill the design requirements and map components to compatible services offered by a selected cloud. While we present implementation for the AWS cloud (Sec. 4), FaaSKeeper's design

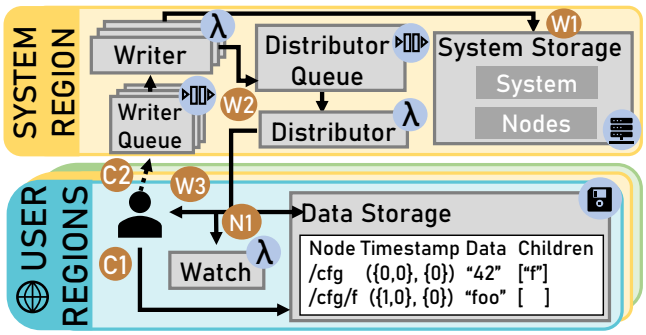


Figure 6. Read and write operations in FaaSKeeper. Queues and timestamps provide strong consistency guarantees.

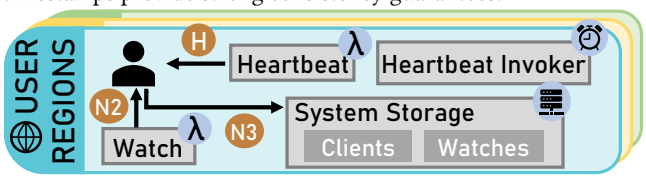


Figure 7. Watch notification delivery and client status verification in FaasKeeper.

and logic are not affected by porting, and the reimplementation effort is limited to incorporating new cloud APIs (Sec. 8).

3.3 System

Figures 6 and 7 present the system design of FaaSKeeper. Following the design principles discussed above, we decouple ZooKeeper data to tailor storage solutions to different requirements and increase the user data locality. The compute tasks are now implemented entirely in a serverless fashion, and the system does not require any resource provisioning. We summarize the components of FaaSKeeper, before going into more detail about design and implementation challenges in the next Section. A detailed discussion on how FaaSKeeper provides the same consistency guarantees as Zookeeper can be found in Appendix B.

Storage. The system state is served by a single instance of key-value storage in the region of FaaSKeeper deployment. Thanks to the strong consistency requirement, parallel instances of writer functions can modify the contents without the risk of reading stale data later. On the other hand, the data storage is optimized to handle read requests in a scalable and cost-efficient manner.

Timestamps. To guarantee the consistency of updates, we need to define a total ordering on the "happened before" relation [60]. Atomic counters are used to implement this necessary feature.

Writer. The single-writer system has been replaced with parallel *event* ***writer*** functions invoked by client queue with new update requests (C2). The writer function obtains exclusive access to the selected node and modifies system storage (W1). The validated and confirmed changes of a writer are propagated through a FIFO queue to the *event* ***distributor*** function (W2), ensuring that the changes to the data storage are not reordered.

Distributor. When the change caused by a writer triggers a watch, the notification is distributed (N1) parallel to the replication of the updated node (W3), thanks to the extended timestamp system that prevents clients from reading updated data before observing all consequences of the update. The *free function* ***watch*** delivers the notification to clients (N2) who registered the given watch (N3). The separation of this functionality from the writer is motivated by performance concerns: hundreds of clients could register a single watch, and the delivery must parallelize efficiently. It is more efficient to invoke the *watch* function in each region instead of sending copies of the same message across regions.

Client. FaaSKeeper clients use an API similar to ZooKeeper. *Read* operations are served through the regional replica of data storage (C1). *Write* operations that modify data and start a new session are sent to a cloud queue (C2). Invocation of a single function instance at a time upholds the ordering of requests within a session. FaaSKeeper implements the same standard read and write operations as ZooKeeper, except for the sync operation. We implement the synchronous and asynchronous variants, and the results are delivered in the usual FIFO order.

Heartbeat. Validation of connection status within a session is necessary to keep ephemeral nodes alive and guarantee notification delivery. Therefore, we implement a *scheduled* ***heartbeat*** function to prune inactive sessions and notify clients that the system is online (H). A time trigger invokes the heartbeat function regularly.The trigger must support dynamic arming and disarming to handle the disconnection of the last and arrival of the first client. Therefore, we replace the heartbeat messages of ZooKeeper with periodically invoked heartbeat functions.

4 Design and Implementation of FaaSKeeper

In this Section, we detail both design and implementation choices made while creating FaaSKeeper, as the serverless design enables scalability and elasticity but comes with a set of new challenges that are not present in ZooKeeper. The parallel processing of updates by the **writer** function requires additional ordering in the **distributor** function to prevent clients from observing an incorrect order of updates. The decomposition of the system and user **data storage** and direct data access by **clients** helps tailor storage solutions to different usage patterns. However, it requires **extended timestamps** to deliver notifications in the correct order. Finally, the **heartbeat** function conducts the verification of the client's status.

We provide a FaaSKeeper implementation for the AWS cloud with the following mapping of the design concepts to cloud services: system storage with synchronization primitives to DynamoDB tables, user data storage to S3 buckets, and FIFO queues to the SQS. We implement the four FaaSKeeper functions in 1,350 lines of Python code in AWS Lambda. Our implementation is standalone and does not reuse the server-centric ZooKeeper codebase since that is written in Java, and would be affected by large cold startup overheads in FaaS [29, 61]. The implementation and testing required an equivalent of three man-months. Additionally, we implement 260 lines of *Infrastructure-as-a-Code* in the multi-cloud framework Serverless [62]. Furthermore, we provide a client library with 1,400 lines of Python code, where we implement all relevant methods of the API specified by ZooKeeper [9]. We offer a compatible interface for existing applications by modeling our API after kazoo [63], a Python client for ZooKeeper. Our implementation provides a high level of compatibility with ZooKeeper (Sec. 4.7).

4.1 Storage

System storage contains the current timestamp and all active sessions, with the clients' addresses and a list of their ephemeral nodes. We use tables in the cloud-native key-value storage, and we store the list of all data nodes to allow lock operations by writer functions. We use only strongly consistent operations.

Algorithm 1 A pseudocode of the writer function.

```
function Writer(updates)
    for each client, node, op, args in updates do
        oldData = Lock(node)  ①
        if not IsValid(op, args, oldData) then  ②
            Notify(client, FAILURE)
        s' = TimestampIncrease  ③
        newData = CommitUnlock(node, op, args, s')  ④
        DistributorPush(client, node, newData, s')
```

Data storage is indexed by node paths, each object corresponds to a single node, and updates to it are atomic. The object contains user node data, modification timestamps, and a list of node children. Eventually consistent reads neither guarantee read–your–write consistency [38], nor consider a dependency between different writes, breaking ZooKeeper guarantees (Linearized Writes Z2, Single System Image Z3). Therefore, we require strong consistency, even though it can be more expensive.

Synchronization Primitives are implemented using condition and update expressions of DynamoDB system storage [64]. Each operation requires a single write to the storage, and the correctness is guaranteed because updates to a single item are always atomic. We cannot use DynamoDB transactions because they do not support mixing read and write operations.

Scalability. The scalability of system and data storage is handled automatically by the cloud provider.

Design goal. ZooKeeper achieves high availability with multiple replicas of the dataset. We achieve the same goal by using automatically replicated cloud storage with a flexible billing model.

4.2 Timestamps

To guarantee the consistency of updates, we need to define a total ordering of modifications in the system. The system ***state* counter** s is an integer representing the **timestamp** of each system change in FaaSKeeper. Each transaction modifies the state counter atomically, providing a total ordering of all processed modification requests. The ***epoch* counter** is associated with a *state* counter and includes the watch notifications pending while the transaction represented by the state counter was in progress. The epoch counter provides an ordering between notifications and changes in a system with decentralized processing of write and read requests.

Implementation. The state and epoch counter are implemented using the atomic primitives (Sec. 2.2).

Design goal. The timestamps provide a total order over system transactions, similar to the *zxid* state counter in ZooKeeper. Epoch counters do not have a corresponding entity since ZooKeeper servers handle watch notifications and read requests for each client.

4.3 Writer

FaaSKeeper replaces the single-leader design of ZooKeeper with concurrently operating **writer functions** to enhance the system's reliability and performance.

Function. The writer function processes user requests in a FIFO order (Alg. 1), The function acquires a lock on the node (①) to prevent concurrent updates, verifies the correctness of the operation (②), e.g., checking that a newly created node does not exist and the conditional update can be applied. Finally, the system timestamp is increased s += 1 (③), the data is secured in the system storage (④), and the results are propagated to the **distributor queue** to update user data storage. Data is not updated if the lock has expired.

Example. The timestamp is $s = 5$, and two writer instances accept updates simultaneously. They request new timestamps from the system, and depending on the order the transactions are processed, the possible timestamp assignments are 5, 6 and 6, 5.

Scalability. FaaSKeeper parallelizes the processing of write requests, as it requires multiple operations on the system storage and relies on optimistic concurrency. When ZooKeeper clients modify different nodes, FaaSKeeper can apply state modifications *concurrently*. Then, the throughput of write requests is limited by the scalability of cloud storage only and atomic timestamp increments. When clients apply changes to the same node, no parallelization is possible in ZooKeeper's consistency model. FaaSKeeper prevents resource starvation in this scenario by applying exponential backoff to locking [45].

Queue. Clients send requests in parallel, and queue invoking functions provides the ordering for a single client. A function concurrency limit of one instance per queue gives an ideal batching opportunity while upholding ordering guarantees. Consecutive requests cannot be reordered, but the first stages of a request (①, ②) can be executed while its predecessor is committed to the storage (③, ④). Thus, the writer function is a sequence of operations on the system storage that can be *pipelined*, improving the system's efficiency. The pipelining corresponds to ZooKeeper's processing of subsequent requests while ensemble servers accept preceding requests. Unfortunately, available serverless queues prevent functions from handling new requests before finishing the previous invocation, forcing *draining* of the pipeline.

Implementation. We select a cloud queue that fulfills the following requirements: (a) invokes functions on messages, (b) upholds FIFO order, (c) allows limiting the concurrency of functions to a single instance, (d) support batching of data items. The requirements guarantee that requests are not reordered (Z3), while (d) ensures efficient processing of frequent invocations in a busy system. We use the AWS SQS with batched Lambda invocations [65] because it performs better than DynamoDB Streams (Sec. 5.2).

Design goal. The concurrent writer functions and queues enable scaling up the processing to match the increasing number of clients in the system while upholding the FIFO order for a single client.

Requirement #4: Queues. Serverless functions require queues to support the ordering and reliability of invocations. However, queues that use discrete batches prevent efficient stream processing with serverless functions. Instead, functions should continuously poll for new items in the queue to keep the pipeline saturated.

4.4 Distributor

The ***distributor*** queue and function are responsible for updating user data storage and sending watch notifications. A distributor queue is necessary to ensure that changes to user data stores are not reordered since concurrent updates could violate consistency (Z3). FaaSKeeper uses the additional region-wide *epoch* counter to ensure that client's consistent view of the data is not affected.

Function. The *distributor* function (Algorithm 2) delivers the updates to each regional data store (❶). The function must ensure that updates are not reordered in a single storage unit, but the process can be parallelized across regions. Since parallel *writers* apply the updates independently, it could happen that two updates x, y processed by different writers are ordered such that $x < y$, but they are received in a reversed order by the distributor and cannot be applied directly (Z2). The unlikely interleaving is handled by sorting the updates (❷) according to their timestamp. However, it

Algorithm 2 A pseudocode of the distributor function.

```
global epoch, MRD  ❸
function DISTRIBUTOR(state, updates)
    (epoch, MRD) = LOAD(state)
    SORT(updates)  ❷
    for each region in parallel do
        for each client, node, data, s', writerID in updates do
            w = WATCHES(node)  ③
            if s' < MRD[region] then  ❹
                s' = TIMESTAMPADJUST(node, s', writerID)  ❺
            DATAUPDATE(region, data, s', epoch)  ❶
            INVOKEWATCH(region, w, WATCHCALLBACK)  ❼
            epoch[region] = epoch[region] + w  ❻
            MRD[region] = s'
            if FINISHED(node) then
                NOTIFY(client, SUCCESS)
    WAITALL(WATCHCALLBACK)
    SAVE(state, (epoch, MRD))
function WATCHCALLBACK(epoch, region, w)
    epoch[region] = epoch[region] − w  ❽
```

can still happen that updates x, y are reordered and scheduled in subsequent batches, and the sorting does not solve the problem there. The "most recent data seen" *MRD* counter (❸) catches delayed transactions in such a case (❹), and their transaction counter s' is updated such that $s' > MRD$ (❺). Adjusting a timestamp is safe: the intra-session ordering is preserved, and the inter-session ordering is not specified and can be changed before the result is visible to the user (Z3). The node data in system storage is updated unless the writer has processed subsequent modifications. The consecutive updates from this writer are corrected thanks to the *MRD* counter, and the system timestamp is updated to propagate the timestamp shift to future requests. However, a *state* persists between serverless invocations to retain the value. The *MRD* value is stored in the storage, and the load operation is optimized away on a *warm* start.

Example. Writers $w1$, $w2$ push updates with $s_{w1} = 5$ and $s_{w2} = 6$. Later, $w1$ processes update $s'_{w1} = 7$. However, s_{w2} appears before s_{w1} in the distributor queue, and the latter is scheduled in the next function invocation. s_{w2} is committed with a timestamp 7 since $s_{w1} < MRD = 6$. The update with $s'_{w1} = 7$ will be readjusted to 8.

Implementation. When committing data to user storage, we attempt to update only changed data to avoid unnecessary network traffic. While DynamoDB offers this feature, the update operation of S3 requires the complete replacement of data. Thus, even if a change involves only node's children, the distributor function needs to download user node data to conduct the update operation.

Requirement #5: Statefulness. *Stateful* functions are necessary for some use-cases, and FaaS systems should support efficient retrieval of a function's state from storage.

Requirement #6: Partial updates. To increase efficiency of storage write operations, cloud storage could support partial updates by allowing writing data stream of a given size to specified object at the user-defined offset.

Scalability. The clients must not see new data before receiving notification preceding it (Ordered Notifications Z4), and stalling updates until all notifications are delivered would put severe performance limitations. Instead, we propose to use *epoch* counters to associate updates with active watches and help clients detect when

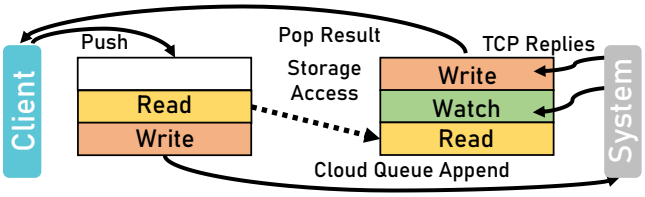


Figure 8. Client library: ordering requests and watch notifications according to the client FIFO and timestamp order.

the reordering of notifications and read operations occurs. Each watch is assigned a unique identifier, and multiple clients can be assigned to a single watch instance. The epoch counter is updated with identifiers of active watches (❻), and it can be updated independently between regions since it does not determine the global order (Z3). After committing the update, an independent function is invoked to deliver notifications in parallel (❼), and the counter is readjusted once all notifications are delivered (❽).

Design goal. The distributor provides the final component to guarantee that writes and notifications offer strong consistency when the single writer is replaced with independent functions and the storage is split between data and system state. ZooKeeper does not have a corresponding entity, as the servers resolve the ordering.

4.5 Client

The elimination of the ZooKeeper server from the data access path provides lower operating costs, but it puts on the client the responsibility of ordering read and write operations with watch notifications. We propose to implement a client-side, queue-like data structure (Figure 8). A background thread receives notifications and replies from the FaaSKeeper service and orders them in the receive queue. Operations return in the same fashion as if a ZooKeeper server processed them: a read following a write cannot return before its predecessor finishes. The client stores the timestamp for the most recent data seen (*MRD*) for all reads, writes, and notifications.

Notifications. Since data access is now independent of processing updates, the clients could violate consistency by reading data newer than a not yet delivered watch notification (Z4). When a read operation returns a node x and $s_x > s_{MRD}$, the client must determine if there are any pending notifications triggered by an update with timestamp y such that $s_{MRD} < s_y < s_x$. As clients are unaware of pending notifications, we use the *epoch* counter to determine if any of the watches registered by the client were being delivered when an update to x was applied. If yes, the read is stalled until the watch notification is delivered. Otherwise, the data can be safely returned to the user.

Example. When a client attempts to read the updated configuration (Fig. 3), it detects that the new node timestamp is $(s, e) = ((42, 7), (0))$. Since $MRD = (40, 7)$ and the client registered watch $w = 0$ before, the result is ready only after receiving the notification.

Implementation. The operations `get`, `exists`, and `get_children`, are implemented with direct access to user storage. Each client runs three background threads: one for sending requests to the system, one for managing incoming TCP connections and messages, and one for processing and ordering results (Fig. 8). Direct storage operations are processed in a non-blocking manner with a thread pool. Each client passes its public address to the system, and serverless functions return the results through a TCP message. The failure of serverless workers is detected through a timeout on each operation.

Requirement #7: Inbound and outbound channels. While the trigger system provides *inbound* communication, the functions lack an ordered, push-based, and high-performance *outbound* communication channel. Such a channel would significantly simplify the design of serverless services such as FaaSKeeper.

Design goal. We replace the event coordination on ZooKeeper servers with a lightweight queueing system on the client. We enable direct low-latency access and a flexible pricing model for non-provisioned resources. However, the extended timestamps are required to ensure that writes are linearized.

4.6 Heartbeat

Besides ordering guarantees, sessions play another significant role in ZooKeeper: their status ensures timely updates and eventual data synchronization (Z3). The disappearance of a client must be detected to handle ephemeral nodes, and we replace the heartbeat messages with the **heartbeat function**.

Implementation. The cloud system periodically invokes the function which sends heartbeat messages to each client in parallel. If a client does not reply before a timeout, he function begins an eviction process for the session by placing a deregistration request in the processing queue. The function is parameterized with the *heartbeat frequency* parameter H_{fr}.

Scalability. Since the verification process has a straightforward parallelization, the solution scales very well with active clients.

Design goal. The verification of client status does not need a persistently allocated server, and FaaSKeeper replaces it with a serverless function that scales according to the number of clients.

4.7 Compatibility with ZooKeeper

While FaaSKeeper aims to provide full compatibility with ZooKeeper, we make minor adjustments due to the limitations of cloud services and the serverless model. The user data storage in S3 supports the ZooKeeper limit of 1 MB of user data in a node. However, DynamoDB and SQS have the size restriction of 400 and 256 kB, respectively, limiting the maximum size of data sent by users to the system. FaaSKeeper can support larger nodes by using temporary S3 objects to transmit write requests and splitting nodes in the system storage. Furthermore, Zookeeper nodes can have their permissions defined using access control lists (ACLs). In FaaSKeeper, write permissions are implemented in functions thanks to the protection boundary between caller and callee, and read permissions can be enforced with ACL of cloud storage. Finally, FaaSKeeper requires additional logic on the client side to handle concurrent write and read requests. We encapsulate it in the client library and allow users to switch to a more complex client at no additional cost and work. There, we provide the same ordering and consistency guarantees for multithreaded applications as ZooKeeper [10].

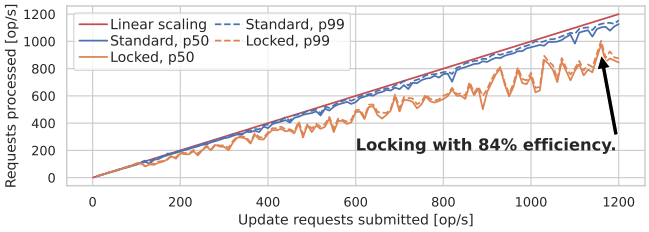
5 Evaluation

The major design goal of FaaSKeeper is a flexible cost model with affordable performance overheads. In this section, we present a detailed evaluation of cloud-native serverless primitives and FaaSKeeper features, focusing on system latencies and cost models. We answer the following evaluation questions.

§ 5.1 Can synchronization primitives enable efficient processing?
§ 5.2 Do serverless queues provide cheap and fast invocations?
§ 5.3 How fast are cloud-native read requests in FaaSKeeper?
§ 5.4 How expensive is the processing of write requests?
§ 5.5 What are the cost savings in service monitoring?

Primitive	Size	Min	p50	p95	p99	Max
Regular	1 kB	3.95	4.35	4.79	6.33	60.26
DynamoDB **write**	64 kB	6.54	66.31	70.28	77.23	121.64
Timed lock	1 kB	6.13	6.8	8.13	14.14	65.32
acquire	64 kB	7.82	67.16	72.71	90.56	177.02
Timed lock	1 kB	6.03	6.62	7.94	12.52	78.44
release	64 kB	6.38	65.2	70.33	92.15	222.64
Atomic counter	8	4.88	5.59	7.01	11.69	62.4
Atomic list	1	5.14	5.89	8.0	10.71	21.12
append	1024	16.72	76.01	184.02	187.47	249.23

(a) Latency of synchronization primitives for varying item size (lock) and list append length (atomic list).



(b) Throughput of standard and locked DynamoDB updates.

Figure 9. Synchronization primitives on AWS DynamoDB.

§ 6 What is the cost break-even point for FaaSKeeper?

Evaluation Platform We deploy FaaSKeeper in the AWS region `us-east-1`. The deployment consists of four functions, AWS SQS queue, and three `DynamoDB` tables storing system state, a users list, and a watch list. Functions are allocated with 2048 MB of memory if not specified otherwise. Additionally, we use a `DynamoDB` table or an S3 bucket for user data storage. FaaSKeeper is implemented in Python with the help of standard library tooling for multithreading. Benchmarks are implemented using Python 3.8.10, and we run microbenchmarks and FaaSKeeper clients from a `t3.medium` virtual machine with `Ubuntu 20.04` in the same cloud region. Furthermore, we deploy ZooKeeper 3.7.0 on a cluster of three `t3.small` EC2 virtual machines running `Ubuntu 20.04`.

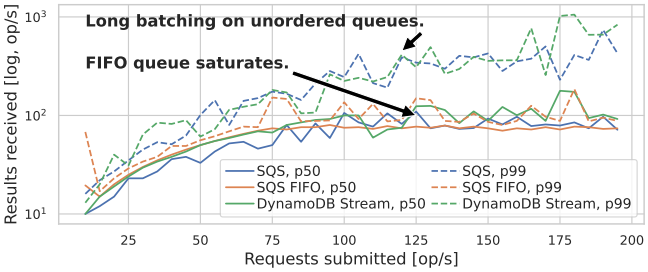
5.1 Synchronization primitives

The serverless synchronization primitives are a fundamental building block for FaaSKeeper operations that allow concurrent and safe updates. Primitives are implemented using conditional update expressions of DynamoDB [64], and we evaluate the overheads and scalability on this datastore system.

Latency. We evaluate the latency of each operation by performing 1000 repetitions on warmed-up data and present results in Table 9a. Each **timed lock** operation requires adding 8 bytes to the timestamp. However, the operation time increases significantly with the item size, even though large data attributes are neither read nor written in this operation. This conditional and custom update adds 2.5 ms to the median time of a regular DynamoDB write, and large outliers further degrade the performance. This result further proves the need to disaggregate the frequently modified system storage from the user data store, where items can store hundreds of kilobytes of data. Then, we evaluate the **atomic counter**, and **atomic list** expansion by expanding the list with a varying number of new items, each one of 1 kB size. This operation allows users to efficiently add new watches by extending the list of watches in storage with a single operation. We observe the same outlier problem on both operations as with timed locks.

	Direct		SQS		SQS FIFO		DynamoDB Stream	
	64B	64 kB	64B	64 kB	64B	64 kB	64B	64 kB
p50	39.0	42.97	39.83	44.22	24.22	28.15	242.65	240.96
p95	73.92	89.09	78.29	95.71	84.29	54.76	270.63	264.95
p99	124.01	129.16	125.24	180.92	162.42	162.41	417.21	364.16
Max	210.11	215.67	295.01	295.01	172.48	244.56	749.16	749.16

(a) End-to-end latency of function invocation with a TCP response.



(b) Throughput of function invocations with 64B payload.

Figure 10. Function invocations with serverless queues.

Throughput. Timed locks should allow FaaSKeeper users to conduct independent updates concurrently. We evaluate a pair of read and write operations and compare our locks with a standard version that does not permit safe parallelization. We measure the median throughput over a range of five seconds and vary the workload, as well as the number of processes sending requests. We use the `c4.2xlarge` VM as a client to support this multiprocessing benchmark (Figure 9b). Even though locks increase the latency of the update operation, the locked version still achieves up to 84% efficiency when handling over 100 requests per second from ten clients concurrently. This result agrees with previous findings that DynamoDB scales up to thousands of transactions per second [66], and the throughput of serverless operations on DynamoDB is limited by Lambda's parallelism and not by storage scalability [67].

Takeaway. Our synchronization primitives introduce a few milliseconds of overhead per operation and allow for efficient parallel processing of FaaSKeeper writes up to 1200 requests per second.

5.2 Serverless Queues

Queues improve performance by batching requests for the pipelined writing process are necessary to provide ordering (Sec. 4.4) AWS offers two cloud-native queues with pay-as-you-go billing and function invocation on new messages: SQS and DynamoDB Streams. For FaaSKeeper, we need to select a queue that adds minimal invocation overhead and allows users to achieve sufficient throughput.

Queue. For SQS [68], we need to enable the FIFO property that comes with the restriction of a maximum batch size of 10. We compare against the standard version to estimate potential overheads of ordering on a queue with a small batch size. For AWS DynamoDB streams, we configure database sharding to guarantee that all new items in a table are processed in order [69]. We restrict the function's concurrency to permit only one instance at a time.

Latency. First, we measure the end-to-end latency triggering an empty function that only returns a dummy result to the user with a TCP connection. We consider the best-case scenario of warm invocations that use a cached TCP connection to the same client. The median round-trip latency to client was 864 µs. In addition to queues, we measure direct function invocations to estimate the effects of user-side batching without cloud proxies, and we present results in Table 10a. Surprisingly, the FIFO queue achieves the lowest latency and is significantly faster than a direct Lambda

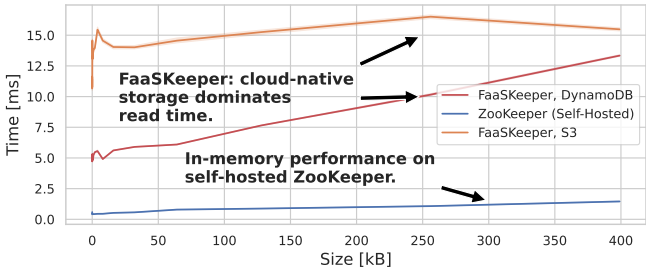


Figure 11. Read operations in FaaSKeeper and ZooKeeper.

invocation. Thus, offloading FaaSKeeper requests using SQS-based invocation comes with approximately 20ms of overhead.

Throughput. Here, we verify how well queues perform with batching and high throughput loads. The queue triggers a function that establishes a connection to the client, and the client measures the median throughput across 10 seconds (Figure 10b). FIFO queues saturate at the level of a hundred requests per second. Meanwhile, DynamoDB and standard SQS experience huge variance, leading to message accumulation and bursts of large message batches. Thus, we cannot expect to achieve higher utilization in FaaSKeeper with a state-of-the-art cloud-native queue, even with ideal pipelining and low-latency storage. However, we can assign one queue per user, which helps to alleviate some of the scalability concerns.

Cost. Lambda invocations cost $0.2 for 1 million requests, regardless of the source. SQS messages are billed in 64 kB increments, and 1 million of them costs $0.5. DynamoDB write units are billed in 1 kB increments, and 1 million of them costs $1.25. Thus, processing requests via SQS is 160x cheaper than with DynamoDB streams.

Takeaway. SQS provides ordering with cost-efficient invocations. Nevertheless, it could be the bottleneck for individual clients.

5.3 Read Operations

ZooKeeper is a system designed for fast read operations, and our serverless FaaSKeeper must also offer efficient reads. We evaluate the `get_data` operation that retrieves a node from storage, timing the retrieval on the user side. In addition to the persistent S3 storage selected as the user data store, we evaluate DynamoDB and compare FaaSKeeper against ZooKeeper. We repeat the measurements 100 times for each node size and present results in Figure 11.

While using DynamoDB can be tempting to provide lower latency on small nodes, this NoSQL database is not designed to serve as a persistent and frequently read data store. Even though it is price efficient up to 4 kB of data, the cost grows quickly: reading 128 kB data is 20x more expensive than S3 since the latter costs only $0.4 for one million reads. ZooKeeper offers much lower latency as it serves data from memory over a warm TCP connection. FaaSKeeper could offer a microsecond-scale read latency by incorporating an in-memory database [39]: these are only now becoming available in a pay-as-you-go billing model [70].

Sorting results, handling watches, and deserializing data adds between 1.9 and 2.5% overhead in our Python implementation.

Takeaway. FaaSKeeper offers fast reads whose performance is bounded by the latency and throughput of the underlying cloud storage. The cost is stable and depends on users' activity only.

5.4 Write Operations

We evaluate the performance of the `set_data` operation that replaces node contents. In addition to measuring total operation time as visible by the client, we measure the execution times of `writer` and `distributor` functions. We measure functions for

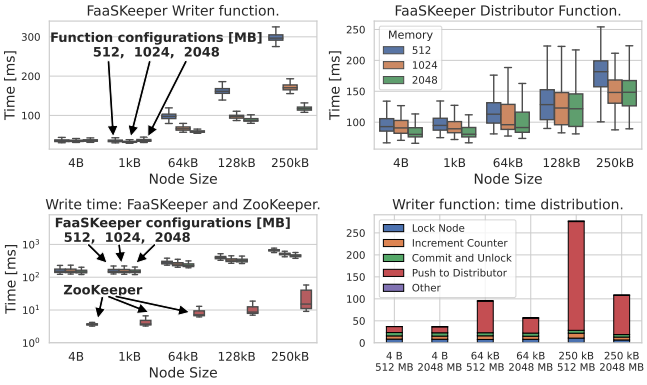


Figure 12. Write operations in FaaSKeeper and ZooKeeper.

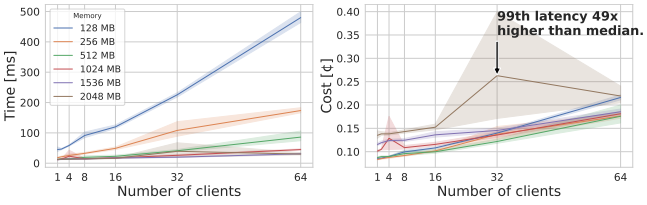


Figure 13. Heartbeat function: performance-cost tradeoffs.

varying memory allocations, and compare our framework against ZooKeeper. The results in Figure 12 show that while the distributor and writer functions run for less than 200 ms each, most of the time is spent on queueing, moving data, invoking functions.

ZooKeeper achieves lower write latency because of the overheads of queueing, invocations, and operations on DynamoDB. To locate the bottleneck of parallel processing in FaaSKeeper, we measure the relative time distribution of the `writer` function. While synchronization operations stay relatively constant, operations with large data dominate the runtime of functions due to pushing updated nodes to the distributor queue.

Takeaway. The performance of write operations is bounded by data transmission to the distributor queue. FaaSKeeper motivates the need for more efficient serverless queues.

5.5 Service Monitoring

An essential part of ZooKeeper is monitoring the status of clients. Automatic removal of non-responsive clients allows for reliable leader election and synchronization protocols. We estimate the time and resources needed for for FaaSKeeper to periodically verify client's status using the `heartbeat` function. We present results averaged from 100 invocations in Figure 13.

The execution time decreases significantly with the memory allocation, corresponding with previous findings on I/O in serverless [28, 29]. While we observe significant variance and long tail latency on the function with the largest memory allocation, this is not a concern for an operation that does not lie on the critical path.

We estimate the cost of monitoring over the entire day, with the highest available frequency on AWS Lambda of an execution every minute. The cost of the function is defined by the computation time and the cost of scanning a DynamoDB table storing the list of users. With the function taking less than 100ms for most configurations, the overall allocation time over 24 hours is less than 0.2% of the entire day. Thus, even for more frequent invocations and more clients, we offer status monitoring for a fraction of VM price.

Parameter	Description	Value
$W_{S3}(s)$	Writing data to S3	$5 \cdot 10^{-6}$
$R_{S3}(s)$	Reading data from S3	$4 \cdot 10^{-7}$
$W_{DD}(s)$	Writing data to DynamoDB	$s \cdot 1.25 \cdot 10^{-6}$
$R_{DD}(s)$	Reading data from DynamoDB	$\lceil \frac{s}{4} \rceil \cdot 0.25 * 10^{-6}$
$Q(s)$	Push to queue	$0.5 \cdot 10^{-6}$
$F_{W/D}(s)$	Execution of writer and distributor function.	Linear models.

Table 4. Parameters of cost model of FaaSKeeper.

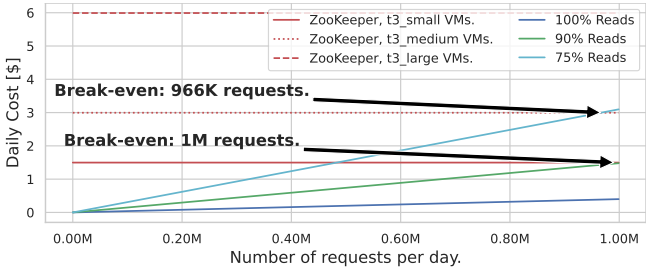


Figure 14. Cost comparison of ZooKeeper and FaaSKeeper.

Takeaway. The serverless heartbeat function replaces a persistent VM allocation and achieves the same goal of client monitoring while reducing the resource allocation time by a huge margin.

6 FaaSKeeper Cost

The final an most important evaluation is comparing the price of running an elastic FaaSKeeperinstance compared to Zookeeper.

FaaSKeeper Costs We present cost models for read and write operations of size of s kilobytes, as the daily monitoring costs are very low. Model parameters are summarized in the Table 4.

Reading. The cost of reading operation is limited to storage access, as all computing is performed by the client library.

$$\text{Cost}_R = R_{S3}(s) \tag{1}$$

Writing. The cost of writing is separated into computing and storing data. There are two queue operations, synchronization in the `writer`, writing data to user store, and two function executions. For both functions, we use regression to estimate linear cost models using data from Sec. 5.4, with R^2 scores between 0.97 and 0.999.

$$\text{Cost}_W = 2 \cdot Q(s) + 3 \cdot W_{DD}(1) + W_{S3}(s) + F_W + F_D \tag{2}$$

Storage. The databases and queues do not generate any inactivity charges except for retaining data in the cloud. Storing user data in S3 with FaaSKeeper is 3.47x cheaper than storing the same data in block storage attached to the EC2 virtual machines hosting ZooKeeper.

ZooKeeper Cost The cost of ZooKeeper is constant and it includes the cost of a persistent allocation of virtual machines. The smallest number of virtual machines is three. Depending on the VM selection, the daily cost of running ZooKeeper changes from \$1.5 on `t3.small`, through 3\$ on the `t3.medium` used for our experiments, up to \$6 on `t3.large`. Additionally, the machines must be provisioned with block storage to store OS, ZooKeeper and user data. 20GB of storage adds a monthly cost of \$4.8.

Comparison We compare ZooKeeper's cost against different read–to–write scenarios in FaaSKeeper, with 1kB writes and functions configured with 512 MB of memory, and we present results in Figure 14. A workload of 100,000 daily read operations costs \$0,04, which is 37.5x cheaper than the most affordable ZooKeeper configuration. For high–read–to–write scenarios, FaaSKeeper can process between 1 and 3.75 million requests daily before the costs are equal to ZooKeeper. Function execution and storage operations are responsible for 13% and 78% of write costs, respectively. When

the memory configuration is increased to 2048 MB, the cost contribution of functions increases to 31%. Since many user nodes do not contain a large amount of data, FaaSKeeper can handle the daily traffic of hundreds of thousands of requests while providing lower costs than ZooKeeper. Contrary to the standard ZooKeeper instance, the serverless design allows us to *shut down* the processing components while not losing any data: the heartbeat function is suspended after the deregistration of the last client, and the only charges come from the durable storage of the system and user data.

7 Related Work

Serverless for Storage Wang et al. [71] use serverless functions as cheap and ephemeral in-memory storage for elastic cache systems. Functions have been used to build data analytics [72] and query engines [73] on top of cloud storage. Boki provides stateful computing on shared logs [74]. DynamoDB is used to build transactional workflows with locks in Beldi [67] and a fault-tolerance shim AFT [66].

Elastic Storage Cloud-native storage solutions are known for elastic implementations that scale well with changes in workload [75]. Examples of reconfiguration controllers include a reactive model using CPU utilization as the primary metric for scaling of the Hadoop Distributed File System[76], a feedforward and feedback controller for key-value storage to resize the service and minimize SLO violations [77], a workload-aware heterogenous reconfiguration engine for HBase [78], a workload predictor with cost-aware reconfiguration [79], latency monitoring and forecasting in database-agnostic replication techniques [8]. PolarDB is a recent example of a database with disaggregated CPU and memory allocations and a serverless billing model [21]. In contrast, ZooKeeper requires autoscaling procedures that integrate the state ordering guarantees. FaaSKeeper does not require a dedicated elasticity controller since our service scales up and down automatically thanks to the auto-provisioning of serverless functions and databases.

ZooKeeper Other authors explored different ideas to improve the performance, availability, and reliability of ZooKeeper. Stewart et al. [80] replicated ZooKeeper contents on multiple nodes to provide predictable access latencies. Distler et al. [11] introduced pluggable extensions into ZooKeeper to increase the performance of coordination patterns by performing additional work atomically on the server. The performance of ZooKeeper's atomic broadcast protocol has been increased with a direct hardware implementation: István et al. [81] presented an FPGA implementation, and Stalder [82] presented an implementation offloading to network interface card (NIC) with the help of PsPIN [83]. Shen et al. [16] proposed a system for live VM migration to improve the performance of geographical migration and reconfiguration of ZooKeeper nodes.

8 Discussion

We now discuss how our requirements are supported in emerging platforms, how our blueprint can help design other applications, and how the cloud-agnostic design facilitates cloud portability.

Can serverless systems support our requirements? We specify seven requirements to define features currently missing in the commercial FaaS systems that are necessary to support distributed, stateful, and scalable applications. The requirements align with the major serverless challenges [27, 84] and are supported in research FaaS platforms. Emerging systems provide microsecond-scale latency [41, 85]. New storage systems satisfy the latency, consistency, and flexibility requirements of functions [39, 40, 44, 86]. Furthermore, stateful serverless is becoming the new norm in clouds [67, 74, 87–89]. Finally, we note that research systems can support many of our requirements already: Cloudburst (R1, R5) [86], PraaS (R1, R5, R7) [89], Boki (R3-R5) [74].

How do our design principles generalize to other applications? The paper demonstrates how the blueprint allows us to design a fully serverless variant of ZooKeeper. This distributed service demonstrates many critical issues in FaaS, such as stateful functions, fast and flexible storage, and synchronization. However, the design requirements (Tab. 1, p. 1) and principles of decoupling state, computations, and communication (Fig. 4, p. 5) can also improve other FaaS applications. Serverless machine learning [90, 91] would benefit from decoupling training data and gradient exchange storage and mapping them to best fitting serverless storage layers. The serial gradient accumulation could be parallelized with our locking primitives. Porting microservices to FaaS [4] requires mapping function's state and cross-function sharing to cloud storage layers and mapping function communication to serverless queues. Message queues are used already for communication and batching inputs for functions [92, 93], and one-third and 60% of serverless applications need messaging and event-based programming, respectively [94]. With the increase towards incorporating serverless computing into unmodified parallel applications [56, 92], the pressure will increase on supporting efficient synchronization primitives that do not require dedicated, non-serverless storage [74, 87].

How can FaaSKeeper be ported to other clouds? The cloud-agnostic design makes it easy to port FaaSKeeper to another cloud system. To that end, we need only to map functions, queue, storage, and synchronization primitives to available cloud services. We demonstrate it on the examples of Microsoft Azure and Google Cloud (GC). First, Azure Functions and GC Functions support pay-as-you-go billing and the three major trigger types [95–97]. FIFO queues with function invocations are offered by GC Pub/Sub [98] and Azure Service Bus [99], albeit the latter does include an upkeep fee with a granularity of one hour. Azure Blob Storage [33] and GC Storage [34] are strongly consistent user data stores. Finally, we need to find a service that satisfies the system's storage latency and cost requirements while permitting the implementation of synchronization primitives. We can use Azure CosmosDB [37] with its optimistic concurrency control [100], and GC Datastore [101] with read-write transactions [102]. Alternatively, the synchronization primitives can be implemented as scripts and transactions in low-latency stores such as Redis [103] and Anna [104].

9 Conclusions

As the tools and mechanisms of cloud computing adapt to the needs of an ever-growing FaaS landscape, creating powerful, fast, and fairly priced serverless services is becoming possible. To facilitate the development of serverless services, we identified seven requirements that cloud providers must fulfill to ensure functionality and good performance. In this work, we have introduced a design blueprint for creating efficient serverless services and built FaaSKeeper, a serverless coordination service offering the same consistency guarantees and interface as Zookeeper. FaaSKeeper allows for elastic deployment that matches system activity, reducing the cost of some configurations by a factor of 37,5.

A ZooKeeper

Below we summarize the provided consistency requirements [9, 10, 46] briefly, considering the case of M clients $C_1, \ldots, C_M$ using a ZooKeeper instance consisting of N servers $S_1, \ldots, S_N$.

Z1 ***Atomicity.*** Write requests never lead to partial results. They are accepted and persistently committed by ZooKeeper or they fail.

Z2 ***Linearized Writes.*** If a client C_i sends update request u before request v, and both are accepted, then it must hold that u "happens before" v, i.e., $u < v$. The guarantee holds for a single session. When clients C_i and C_j send requests $u_1, u_2, \ldots$ and $v_1, v_2, \ldots$, respectively, the ordering between any u_i and v_j is not defined.

Z3 ***Single and Reliable System Image.*** The order of successful updates is visible as identical to every client: for any updates u and v, if a client C connected to a server S observes that $u < v$, it must hold that $u < v$ for any client C' connected to any server S'. Furthermore if a client C observes node Z with version V, it cannot later see the node Z with version V' such that $V' < V$, even if session mechanism switched servers due to failure or network outage. Each view of the system will become up-to-date after bounded time, or a disconnection notification will be delivered (*timeliness*). Accepted updates are never rolled back.

Z4 ***Ordered Notifications.*** Watch notifications are delivered in the order of updates that triggered them. Their ordering with respect to other notifications and writes must be preserved. If an update u triggers a watch notification for a client C, the client must observe the notification before seeing any data touched by transaction v such that $u < v$. In particular, if a client C has a watch registered on any node Z with version V, it will receive watch notification before seeing any data associated with node Z with version V' such that $V < V'$. The property outlined above is global, i.e., it affects all changes preceded by the notification, not only changes related to watches registered by the client.

B FaaSKeeper Consistency Model

Z1 ***Atomicity.*** The updates in the system storage are performed in a single operation on the key-value storage that is guaranteed to be atomic. The results of the operation are eventually propagated to all data replicas. Incorrect operations have no effect on the system.

Z2 ***Linearized Writes.*** Updates are processed in a FIFO order by the *writer* function. The queue guarantees that only a single writer instance can be active at a time, and the function is not allowed to reorder any two requests unless they come from a different session. Therefore, any two update requests u, v in the same session cannot be assigned timestamp value such that $u \geq v$. The single *distributor* instance guarantees that clients reading from user data never observe v before u. Different sessions can use different queues and see their respective requests be reordered, which conforms to ZooKeeper's undefined ordering of requests between clients.

Z3 ***Single and Reliable System Image.*** Nodes are stored in a cloud storage with automatic replication and a strongly consistent read must always return the newest data. Thus, if a client C observes updates u, v such that $u < v$, all other clients must read either the same or newer data. Furthermore, strongly consistent reads prevent clients from observing an order of updates V, V', V.

Z4 ***Ordered Notifications.*** FaaSKeeper guarantees that transactions with timestamp v are not visible before receiving all notifications corresponding to updates v' such that $v' < v$. When a read returns node with timestamp v, it is first compared with the *MRD* value of the current session. If $v < MRD$, then by the transitive property of the total order, any pending watch notifications must be newer than v, and data is safe to read. Otherwise, the are two possible situations: (a) a watch notification relevant to the client was active but not yet deliveed (❺ in Alg. 2) before storing v (❶ there), and (b) no relevant watch notifications are being processed.

In the former case, if a transaction v' triggers watch w, it is added to the *epoch* counter (❹) before committing v, Thus, for each transaction following v', watch w must be included in the *epoch* unless the notification is delivered to each client (❻). This prevents the client from seeing transaction v unless notification of watch w is delivered. In the latter case, the client library releases the data immediately because watch w is not present in the *epoch*.

References

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (April 2010), 50–58. https://doi.org/10.1145/1721654.1721672

[2] James M Kaplan, William Forrest, and Noah Kindler. 2008. Revolutionizing data center energy efficiency. *McKinsey & Company* (2008), 1–13.

[3] Robert Birke, Lydia Y. Chen, and Evgenia Smirni. 2012. Data Centers in the Cloud: A Large Scale Performance Study. In *2012 IEEE Fifth International Conference on Cloud Computing*. 336–343. https://doi.org/10.1109/CLOUD.2012.87

[4] Zewen Jin, Yiming Zhu, Jiaan Zhu, Dongbo Yu, Cheng Li, Ruichuan Chen, Istemi Ekin Akkus, and Yinlong Xu. 2021. Lessons Learned from Migrating Complex Stateful Applications onto Serverless Platforms. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hong Kong, China) *(APSys '21)*. Association for Computing Machinery, New York, NY, USA, 89–96. https://doi.org/10.1145/3476886.3477510

[5] 2018. Amazon DynamoDB On-Demand – No Capacity Planning and Pay-Per-Request Pricing. https://aws.amazon.com/blogs/aws/amazon-dynamodb-on-demand-no-capacity-planning-and-pay-per-request-pricing/. Accessed: 2022-01-30.

[6] 2020. Azure Cosmos DB serverless now in preview. https://devblogs.microsoft.com/cosmosdb/serverless-preview/. Accessed: 2022-01-30.

[7] 2021. DataStax Serverless: What We Did and Why It's a Game Changer. https://www.datastax.com/blog/2021/02/datastax-serverless-what-we-did-and-why-its-game-changer. Accessed: 2022-01-30.

[8] Sean Barker, Yun Chi, Hakan Hacigümüs, Prashant Shenoy, and Emmanuel Cecchet. 2014. ShuttleDB: Database-Aware Elasticity in the Cloud. In *11th International Conference on Autonomic Computing (ICAC 14)*. USENIX Association, Philadelphia, PA, 33–43. https://www.usenix.org/conference/icac14/technical-sessions/presentation/barker

[9] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 11.

[10] Flavio Junqueira and Benjamin Reed. 2013. *ZooKeeper: Distributed Process Coordination* (1st ed.). O'Reilly Media, Inc.

[11] Tobias Distler, Christopher Bahn, Alysson Bessani, Frank Fischer, and Flavio Junqueira. 2015. Extensible Distributed Coordination. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 10, 16 pages. https://doi.org/10.1145/2741948.2741954

[12] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[13] Paolo Viotti, Dan Dobre, and Marko Vukolić. 2017. Hybris: Robust Hybrid Cloud Storage. *ACM Trans. Storage* 13, 3, Article 27 (Sept. 2017), 32 pages. https://doi.org/10.1145/3119896

[14] Cuong Manh Pham, Victor Dogaru, Rohit Wagle, Chitra Venkatramani, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2014. An Evaluation of Zookeeper for High Availability in System S. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering* (Dublin, Ireland) *(ICPE '14)*. Association for Computing Machinery, New York, NY, USA, 209–217. https://doi.org/10.

1145/2568088.2576801
[15] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. 2007. Workload Analysis and Demand Prediction of Enterprise Data Center Applications. In *2007 IEEE 10th International Symposium on Workload Characterization*. 171–180. https://doi.org/10.1109/IISWC.2007.4362193
[16] Zhiming Shen, Qin Jia, Gur-Eyal Sela, Ben Rainero, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. 2016. Follow the Sun through the Clouds: Application Migration for Geographically Shifting Workloads. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 141–154. https://doi.org/10.1145/2987550.2987561
[17] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3132747.3132772
[18] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. 2009. Server Workload Analysis for Power Minimization Using Consolidation. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (San Diego, California) *(USENIX'09)*. USENIX Association, USA, 28.
[19] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2013. ElasTraS: An Elastic, Scalable, and Self-Managing Transactional Database for the Cloud. *ACM Trans. Database Syst.* 38, 1, Article 5 (April 2013), 45 pages. https://doi.org/10.1145/2445583.2445588
[20] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus Database: How to Be Fast, Available, and Frugal in the Cloud. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1463–1478. https://doi.org/10.1145/3318464.3386129
[21] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. *PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers*. Association for Computing Machinery, New York, NY, USA, 2477–2489. https://doi.org/10.1145/3448016.3457560
[22] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache
[23] 2014. AWS Lambda. https://aws.amazon.com/lambda/. Accessed: 2022-01-30.
[24] 2016. Azure Functions. https://azure.microsoft.com/en-us/services/functions/. Accessed: 2022-01-30.
[25] 2017. Google Cloud Functions. https://cloud.google.com/functions/. Accessed: 2022-01-30.
[26] 2016. IBM Cloud Functions. https://cloud.ibm.com/functions/. Accessed: 2022-01-30.
[27] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383
[28] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. https://doi.org/10.1145/3464298.3476133
[29] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 133–145.
[30] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 923–935.
[31] Eric Jonas, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. *CoRR* abs/1702.04024 (2017). arXiv:1702.04024 http://arxiv.org/abs/1702.04024
[32] 2006. AWS S3. https://aws.amazon.com/s3/. Accessed: 2022-01-30.
[33] 2008. Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/. Accessed: 2022-01-30.
[34] 2010. Google Cloud Storage. https://cloud.google.com/storage. Accessed: 2022-01-30.
[35] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 143–157. https://doi.org/10.1145/2043556.2043571
[36] 2012. AWS Dynamo DB. https://aws.amazon.com/nosql/key-value/. Accessed: 2022-01-30.
[37] 2017. Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/. Accessed: 2022-01-30.
[38] Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (2009), 40–44.
[39] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 789–794. https://www.usenix.org/conference/atc18/presentation/klimovic-serverless
[40] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 427–444.
[41] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. 2021. RFaaS: RDMA-Enabled FaaS Platform for Serverless High-Performance Computing. arXiv:2106.13859 [cs.DC]
[42] Bruce Jay Nelson. 1981. *Remote procedure call*. Carnegie Mellon University.
[43] Andrew Stuart Tanenbaum and Robbert Van Renesse. 1987. *A critique of the remote procedure call paradigm*. Vrije Universiteit, Subfaculteit Wiskunde en Informatica.
[44] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the Gap Between Serverless and Its State with Storage Functions. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3357223.3362723
[45] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[46] 2020. ZooKeeper Programmer's Guide. https://zookeeper.apache.org/doc/current/zookeeperProgrammers.html. Accessed: 2022-01-30.
[47] F. P. Junqueira, B. C. Reed, and M. Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. 245–256. https://doi.org/10.1109/DSN.2011.5958223
[48] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972
[49] 2020. ZooKeeper Dynamic Reconfiguration. https://zookeeper.apache.org/doc/current/zookeeperReconfig.html. Accessed: 2022-01-30.
[50] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio P. Junqueira. 2012. Dynamic Reconfiguration of Primary/Backup Clusters. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 425–437. https://www.usenix.org/conference/atc12/technical-sessions/presentation/shraer
[51] Rainer Schiekofer, Johannes Behl, and Tobias Distler. 2017. Agora: A Dependable High-Performance Coordination Service for Multi-cores. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 333–344. https://doi.org/10.1109/DSN.2017.23
[52] Raluca Halalai, Pierre Sutra, Étienne Rivière, and Pascal Felber. 2014. ZooFence: Principled Service Partitioning and Application to the ZooKeeper Coordination Service. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. 67–78. https://doi.org/10.1109/SRDS.2014.41
[53] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N. Calheiros. 2010. InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services. In *Algorithms and Architectures for Parallel Processing*, Ching-Hsien Hsu, Laurence T. Yang, Jong Hyuk Park, and Sang-Soo Yeo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 13–31.
[54] 2021. AWS S3: Troubleshooting replication. https://docs.aws.amazon.com/AmazonS3/latest/userguide/replication-troubleshoot.html. Accessed: 2022-01-30.
[55] Benjamin Satzger, Waldemar Hummer, Christian Inzinger, Philipp Leitner, and Schahram Dustdar. 2013. Winds of Change: From Vendor Lock-In to the Meta Cloud. *IEEE Internet Computing* 17, 1 (2013), 69–73. https://doi.org/10.1109/MIC.2013.19
[56] Josep Sampe, Pedro Garcia-Lopez, Marc Sanchez-Artigas, Gil Vernik, Pol Roca-Llaberia, and Aitor Arjona. 2021. Toward Multicloud Access Transparency in Serverless Computing. *IEEE Software* 38, 1 (2021), 68–74. https://doi.org/10.1109/MS.2020.3029994

[57] Ataollah Fatahi Baarzi, George Kesidis, Carlee Joe-Wong, and Mohammad Shahrad. 2021. *On Merits and Viability of Multi-Cloud Serverless.* Association for Computing Machinery, New York, NY, USA, 600–608. https://doi.org/10.1145/3472883.3487002

[58] Gojko Adzic and Robert Chatley. 2017. Serverless Computing: Economic and Architectural Impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 884–889. https://doi.org/10.1145/3106237.3117767

[59] Dana Petcu. 2011. Portability and Interoperability between Clouds: Challenges and Case Study. In *Towards a Service-Based Internet*, Witold Abramowicz, Ignacio M. Llorente, Mike Surridge, Andrea Zisman, and Julien Vayssière (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 62–74.

[60] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[61] David Jackson and Gary Clynch. 2018. An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 154–160. https://doi.org/10.1109/UCC-Companion.2018.00050

[62] 2020. Serverless Framework. https://github.com/serverless/serverless. Accessed: 2022-01-30.

[63] 2021. Kazoo: high-level Python library for ZooKeeper. https://github.com/python-zk/kazoo. Accessed: 2022-01-30.

[64] 2021. Using Expressions in DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Expressions.html. Accessed: 2022-01-30.

[65] 2021. Using AWS Lambda with Amazon SQS. https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html. Accessed: 2022-01-30.

[66] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. 2020. A Fault-Tolerance Shim for Serverless Computing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/3342195.3387535

[67] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1187–1204. https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

[68] 2021. AWS SQS High Throughput Mode for SQS. https://aws.amazon.com/about-aws/whats-new/2021/05/amazon-sqs-now-supports-a-high-throughput-mode-for-fifo-queues/. Accessed: 2022-01-30.

[69] 2020. Using AWS Lambda with Amazon DynamoDB. https://docs.amazonaws.cn/en_us/lambda/latest/dg/with-ddb.html. Accessed: 2022-01-30.

[70] 2021. Upstash: Serverless Database for Redis. https://upstash.com/redis. Accessed: 2022-01-30.

[71] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 267–281. https://www.usenix.org/conference/fast20/presentation/wang-ao

[72] Ingo Müller, Renato Marroquin, and Gustavo Alonso. 2019. Lambada: Interactive Data Analytics on Cold Data using Serverless Cloud Infrastructure. *ArXiv* abs/1912.00937 (2019).

[73] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3318464.3380609

[74] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 691–707. https://doi.org/10.1145/3477132.3483541

[75] Ioannis Konstantinou, Evangelos Angelou, Christina Boumpouka, Dimitrios Tsoumakos, and Nectarios Koziris. 2011. On the Elasticity of NoSQL Databases over Cloud Management Platforms. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (Glasgow, Scotland, UK) *(CIKM '11)*. Association for Computing Machinery, New York, NY, USA, 2385–2388. https://doi.org/10.1145/2063576.2063973

[76] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. 2010. Automated Control for Elastic Storage. In *Proceedings of the 7th International Conference on Autonomic Computing* (Washington, DC, USA) *(ICAC '10)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/1809049.1809051

[77] Ahmad Al-Shishtawy and Vladimir Vlassov. 2013. ElastMan: Elasticity Manager for Elastic Key-Value Stores in the Cloud. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference* (Miami, Florida, USA) *(CAC '13)*. Association for Computing Machinery, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/2494621.2494630

[78] Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, and Ricardo Vilaça. 2013. MeT: Workload Aware Elasticity for NoSQL. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/2465351.2465370

[79] Ashraf Mahgoub, Paul Wood, Alexander Medoff, Subrata Mitra, Folker Meyer, Somali Chaterji, and Saurabh Bagchi. 2019. SOPHIA: Online Reconfiguration of Clustered NoSQL Databases for Time-Varying Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 223–240. https://www.usenix.org/conference/atc19/presentation/mahgoub

[80] Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. 2013. Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs. In *10th International Conference on Autonomic Computing (ICAC 13)*. USENIX Association, San Jose, CA, 265–277. https://www.usenix.org/conference/icac13/technical-sessions/presentation/stewart

[81] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a Box: Inexpensive Coordination in Hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 425–438. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/istvan

[82] Elias Stalder. 2020. *Zoo-Spinner: A Network-Accelerated Consensus Protocol.* Master's thesis. ETH Zurich.

[83] Salvatore Di Girolamo, Andreas Kurth, Alexandru Calotoiu, Thomas Benz, Timo Schneider, Jakub Beránek, Luca Benini, and Torsten Hoefler. 2020. PsPIN: A high-performance low-power architecture for flexible in-network compute. *arXiv preprint arXiv:2010.03536* (2020).

[84] Pedro Garcia Lopez, Aleksander Slominski, Michael Behrendt, and Bernard Metzler. 2021. Serverless Predictions: 2021-2030. arXiv:2104.03075 [cs.DC]

[85] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing forLatency-Sensitive, Interactive Microservices. In *Proceedings ofthe 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3445814.3446701

[86] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 12 (July 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[87] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/3361525.3361535

[88] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[89] Marcin Copik, Alexandru Calotoiu, Rodrigo Bruno, Roman Böhringer, and Torsten Hoefler. [n.d.]. Process-as-a-Service: FaaSt Stateful Computing with Optimized Data Planes. ([n. d.]). https://spcl.inf.ethz.ch/Publications/.pdf/praas_2022.pdf Accessed: 2022-01-10.

[90] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A Serverless Framework for End-to-End ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3357223.3362711

[91] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. *Towards Demystifying Serverless Machine Learning Training.* Association for Computing Machinery, New York, NY, USA, 857–871. https://doi.org/10.1145/3448016.3459240

[92] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020. Kappa: A Programming Framework for Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/3419111.3421277

[93] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. *CoRR* abs/1812.03651 (2018). arXiv:1812.03651 http://arxiv.org/abs/1812.03651

[94] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2021. Serverless Applications: Why, When, and How? *IEEE Software* 38, 1 (2021), 32–39. https://doi.org/10.1109/MS.2020.3023302

[95] 2021. Azure Functions triggers and bindings concepts. https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings. Accessed: 2022-01-30.

[96] 2021. Google Cloud Functions Events and Triggers. https://cloud.google.com/functions/docs/concepts/events-triggers. Accessed: 2022-01-30.

[97] 2021. Google Cloud: Using Pub/Sub to trigger a Cloud Function. https://cloud.google.com/scheduler/docs/tut-pub-sub. Accessed: 2022-01-30.
[98] 2021. Google Cloud Pub/Sub: Ordering messages. https://cloud.google.com/pubsub/docs/ordering. Accessed: 2022-01-30.
[99] 2021. Azure Service Bus: Message sessions. https://docs.microsoft.com/en-us/azure/service-bus-messaging/message-sessions. Accessed: 2022-01-30.
[100] 2021. Azure Cosmos DB: Transactions and optimistic concurrency control. https://docs.microsoft.com/en-us/azure/cosmos-db/sql/database-transactions-optimistic-concurrency. Accessed: 2022-01-30.
[101] 2013. Google Cloud Datastore. https://cloud.google.com/datastore. Accessed: 2022-01-30.
[102] 2021. Google Cloud Datastore: Transactions. https://cloud.google.com/datastore/docs/concepts/transactions. Accessed: 2022-01-30.
[103] 2013. Redis in Action: Scripting Redis with Lua. https://redis.com/ebook/part-3-next-steps/chapter-11-scripting-redis-with-lua/. Accessed: 2022-01-30.
[104] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. 2018. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 401–412. https://doi.org/10.1109/ICDE.2018.00044