FaaSKeeper: Learning from Building Serverless Services with ZooKeeper as an Example

Marcin Copik
ETH Zürich

Alexandru Calotoiu
ETH Zürich

Konstantin Taranov
Microsoft

Torsten Hoefler
ETH Zürich

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 functions in the serverless paradigm while still fulfilling functional requirements is challenging. In this work, we demonstrate how ZooKeeper, a centralized coordination service that offers a safe and wait-free consensus mechanism, can be redesigned to benefit from serverless computing. We define synchronization primitives to extend the capabilities of scalable cloud storage and contribute a set of requirements for efficient and scalable FaaS computing. We introduce FaaSKeeper, the first coordination service built on serverless functions and cloud-native services, and share serverless design lessons based on our experiences of implementing a ZooKeeper model deployable to clouds today. *FaaSKeeper provides the same consistency guarantees and interface as ZooKeeper*, with a serverless price model that lowers costs up to 37.5 times on infrequent workloads.

1 Introduction

FaaS is a new paradigm that combines elastic and on-demand resource allocation with an abstract programming model. In FaaS, the cloud provider invokes stateless functions, freeing the user from managing the software and hardware resources. Flexible resource management and a pay-as-you-go billing help with 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 high-performance applications and require storage to support state and communication. However, stateful applications can benefit from serverless services [4], and even databases adapt on-demand offerings to handle infrequent workloads more efficiently [5–7].

Apache ZooKeeper [8] is a prime example of a system that has been widely adopted by many applications but is not available as a serverless service. ZooKeeper provides a coordination service for distributed applications to control the shared state and guarantee data consistency and availability. Compared to cloud key-value storage, ZooKeeper provides additional semantics of total order with linearizable writes, atomic updates, and ordered push notifications (Table 1).

	ZooKeeper	Cloud Storage	FaaSKeeper
⏫	Semi-automatic, ≥ 3 VMs	**Automatic**	**Automatic**
⏬	Not possible.	**Only storage fees**	**Only storage fees**
💲	Pay upfront	**Pay-as-you-go**	**Pay-as-you-go**
🛡️	Depends on cluster size	**Cloud SLA**	**Cloud SLA**
🔄	**Linearized writes**	Strong consistency	**Linearized writes**
✉️	**Watch events**	None	**Watch events**
🔀	**Sequential nodes**, conditional updates	Conditional updates.	**Sequential nodes**, conditional updates
🐞	**Ephemeral nodes**	None	**Ephemeral nodes**

Table 1: **FaaSKeeper combines the best features of cloud storage: scale–to–zero scalability (⏫⏬) and reliability (🛡️), with ZooKeeper's consistency (🔄), push notifications (✉️), and support for concurrency and fault tolerance (🔀 🐞).**

Services are expected to match the temporal and geographical variability of production workloads [9–11]. Workloads are often bursty and experience rapid changes: the maximum system utilization can be multiple times higher than even the 99th percentile [9, 12]. However, the static ZooKeeper architecture makes the readjustment to the burst workload difficult. Even when ZooKeeper is co-located as a part of a larger system, it still contributes to the overprovisioning of resources for the peak workload. Serverless applications built on top of cloud storage could adapt to diurnal changes in workload and handle thousands of requests daily at lower cost. A *serverless* service with the same consistency guarantees would offer the opportunity to consolidate variable workloads, helping users and cloud operators increase efficiency. Unfortunately, the path to serverless for such distributed applications is unclear due to the restricted and vendor-specific nature of FaaS.

In this work, **we chart the path needed to build a complex serverless service** — serverless ZooKeeper. We choose ZooKeeper **because** it is a complex, **reliable** service, and

therefore challenges both the capabilities and the limitations of inherently unreliable FaaS systems. First, we decouple the system from the application state and compute from storage tasks [13–15], and derive a *design guide* for the serverless system architecture (Sec. 3). We focus on the semantics of building components and abstract away differences in interfaces and services, helping design *cloud-agnostic* systems that are easily portable between clouds (Sec. 4).

Finally, we introduce and evaluate **FaaSKeeper**, the first coordination service with a serverless scaling and billing model. 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 1). Standing on the shoulders of ZooKeeper, we show how consensus can be implemented as a FaaS application, and create a prototype of the provider-agnostic system on AWS (Sec. 4). FaaSKeeper offers a pay-as-you-go cost model while upholding consistency and ordering properties (Sec. 6).

In summary, we make the following contributions:

- Serverless design lessons on how to compose functions with cloud services to support synchronization, message ordering, and event-based communication.
- The first complex serverless solution that offers the same level of service as its IaaS counterpart without provisioned resources.
- A serverless ZooKeeper consistency model with a compatible interface that achieves 37.5 times lower costs against the smallest ZooKeeper deployment.

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. Storage services are necessary for FaaS to maintain state and implement reliable computing, but they differ in performance, consistency, and costs. We first provide an overview of FaaS (Figure 1), and then introduce a set of fundamental building blocks for the design of serverless services.

2.1 Background

Serverless functions replace persistent virtual machines with elastic and dynamic execution of fine-grained tasks. The management of the software and hardware stack becomes the sole responsibility of the cloud provider, and users are charged only for the time and resources consumed during the function execution (*pay-as-you-go*). In place of cloud resource management and orchestration systems, functions offer various *triggers* to process internal cloud events and external REST requests (1). A function scheduler (2) routes the invocation to a selected server [16], and the function executes within an isolated sandbox on a multi-tenant server (3). The cloud scheduler aims to increase performance by reusing sandboxes, since *warm* execution in an existing container is faster than *cold* invocations that wait for sandbox allocation.

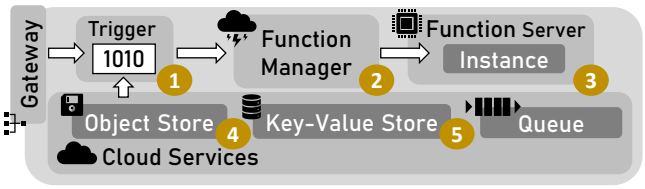


Figure 1: **A high-level overview of a FaaS platform.**

Users adopt the FaaS computing model to improve efficiency in irregular workloads. Cloud operators provide serverless platforms [17–20], as they benefit from increased resource utilization when running functions on multi-tenant and oversubscribed servers. The challenges in serverless computing are high-latency warm and cold invocations, low data locality, variable performance, and limited I/O bandwidth [21–24]. Furthermore, stateless functions require a persistent service for messaging in distributed applications [25, 26].

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

Object. Object storage is designed to store large amounts of data for a long time while providing high throughput and durability (4). The cloud operator manages replication across multiple instances in physically and geographically separated data centers, providing high availability and reliability. Modern object stores offer strong consistency on read operations [27], guaranteeing that successful writes are immediately visible to other clients. The billing model is linear in both the data amount and the number of performed operations.

Key-Value. The second type of cloud storage common in serverless applications is a nonrelational database (5). In addition to strong consistency, read operations can be executed with eventual consistency [28], trading consistency for lower costs, improved latency, and higher availability. Storage can offer optimistic concurrency with *conditional* updates that apply atomic operations to existing attributes.

Other. FaaS can employ additional storage systems, but these often introduce resource provisioning. *Ephemeral* storage [29, 30] is designed to meet serverless requirements for scalability and flexibility. In-memory caches bring lower latency and are being adapted to serverless scalability [31, 32].

2.2 Serverless Components

We define function models and propose synchronization primitives necessary for serverless services, including FaaSKeeper. We do not limit ourselves to features currently available on cloud platforms and propose extensions that providers could offer, enabling more efficient and robust serverless services.

Functions We specify three distinct classes of functions that are necessary to implement a serverless application or a microservice and have divergent interfaces and fault-tolerance models — their semantics express different programming language constructs. A **free function** is not bound to any cloud resource and is invoked via an API request. Free functions express the semantics of *remote procedure calls* [33]. The event-driven programming paradigm is implemented by

providing **event functions** to react to specific changes in cloud storage, databases, or queues. There, API requests are replaced by sending a message to a queue that triggers the function. Furthermore, using such a proxy allows coalescing many invocations into a larger batch and preserving their internal ordering. We expect the batching and concurrency of invocations to be configurable for each trigger, as batching improves the throughput of serverless processing, and restricting concurrency to a single instance is necessary to ensure FIFO order. Semantically, we interpret these functions as remote *asynchronous callbacks* to events in the system.

Functions can be launched to perform regular routines such as garbage collection and detecting system faults. **Scheduled functions** are the serverless counterpart of a `cron` job in Unix-based operating systems. In the event of an unexpected failure, the cloud should provide a retry policy with a finite number of repetitions. Furthermore, users should be notified after repeated failed invocations to detect system-wide failures, even when they do not directly control such functions.

Synchronization Primitives Concurrent operations require fundamental synchronization operations to ensure safe state modifications [34]. Serverless requires synchronization primitives that operate on storage instead of shared memory.

A **timed lock** extends a regular lock with a limited holding time, similarly to leases [35]. It is a necessary feature to prevent a system-wide deadlock caused by a failure of an ephemeral function. Lock operations are submitted with a user timestamp. The lock is *acquired* if no timestamp is present, and when the difference between the existing and new timestamps is greater than a predefined maximum time. To prevent accidental overwriting after losing the lock, each update to a locked resource compares the stored timestamp with the user value. The lock *release* removes the timestamp. An **atomic counter** supports single-step updates while **atomic list** provides safe expansion and truncation.

3 From ZooKeeper to FaasKeeper

FaaSKeeper implements the distributed coordination of ZooKeeper (Sec. 3.1). To build such a serverless service, we introduce a multi-step design guideline (Sec. 3.2): we disaggregate computing from storage and incorporate different types of cloud storage. We decompose ZooKeeper into separate cloud services, tailoring resource requirements to each component and enabling serverless scalability (Sec. 3.3).

3.1 ZooKeeper

ZooKeeper guarantees data persistence and high read performance by allocating replicas of the entire system on multiple servers [8, 36, 37]. ZooKeeper ensemble consists of servers with an elected leader whose roles are verification and processing write requests with the help of the ZAB atomic broadcast protocol [38]. In practice, the rule of $2f+1$ servers is used: for three servers, two are required to accept change and failure

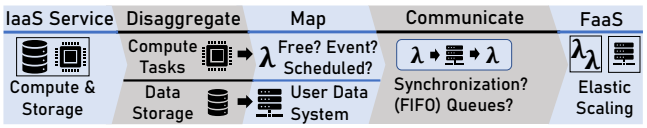


Figure 2: **Workflow for designing a serverless service.**

of one can be tolerated. While adding more servers increases reliability, it hurts write performance.

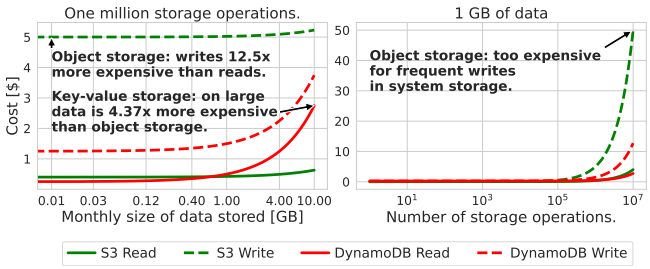
In the static ZooKeeper configuration, changing the deployment size involved *rolling restarts*, a manual and error-prone process [39]. While ZooKeeper has been later enhanced with dynamic reconfiguration [40], it still requires manual effort [39], and reconfiguration causes significant performance degradation when deploying across geographical regions [10].

ZooKeeper splits the responsibilities between the client library, servers, and the elected leader. User data is stored in *nodes*, which create a tree structure with parents and children. Clients send requests to a server through a session mechanism that guarantees the first-in-first-out (FIFO) order of processing requests, achieved over reliable and fast TCP connections. Read requests are resolved using a local data replica, while write operations are forwarded to the leader. The leader updates nodes, manages the voting process, and propagates changes to other servers. ZooKeeper defines the order of transactions with a monotonically increasing counter *zxid*. While requests from a single client cannot be reordered, the order of operations between different sessions is not specified. ZooKeeper supports push notifications with *watches*. Clients register watches on a node and receive a notification when that node changes. Finally, clients exchange heartbeat messages with a server to keep the session alive.

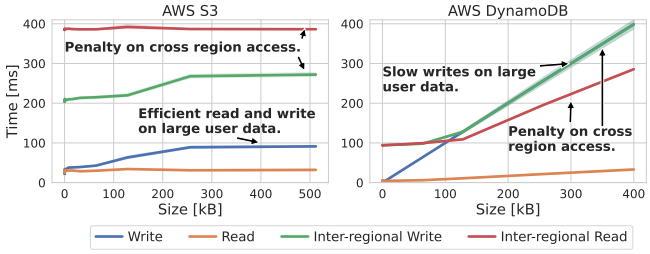
Consistency Model. ZooKeeper implements sequential consistency guarantees with four main requirements (Z). All operations of a single client are executed **atomically** (Z1), in FIFO order, and writes are **linearized** (Z2). The order of transactions is *total* and *global*. Thus, all clients have a **single system image** (Z3), and they observe the same order of updates. Watches ensure that clients know about a change before observing subsequent updates since notifications are **ordered** with read and write operations (Z4). Formal definitions of each guarantee can be found in the appendix (Sec. A).

3.2 Challenges in FaaSKeeper

The main advantage of FaasKeeper is that compute tasks can now be fully serverless, and the system does not require any resource provisioning. However, the design and implementation of ZooKeeper are incompatible with the serverless paradigm and require us to build a reliable service on top of a fundamentally unreliable foundation. Therefore, we design FaasKeeper from the ground up to replicate the complex ZooKeeper functionality and overcome the inherent challenges of the serverless worlds. We follow a general workflow for turning an



(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 3: **Cost and performance of storage in the AWS cloud. Python benchmarks executed on an EC2 instance.**

IaaS system into FaaS (Fig. 2): disaggregate compute and data map them onto cloud storage and functions, and let these components communicate.

Disaggregate. Although ZooKeeper servers manage connections and ordering, their primary responsibility is to provide low-latency data access that can be replaced with cloud storage. In a coordination system designed for high read-to-write ratios, data endpoints require larger allocations than computing tasks. Furthermore, storage should distinguish between user data and the system data needed to control ZooKeeper: their locality and cost requirements are different.

User data locality. Cloud applications balance resource allocation across geographical regions to support changing workloads [10, 41]. In addition, they aim to minimize the distance between the service and its users, as the cross-region transmission adds major performance and cost overheads (Fig. 3b). While ZooKeeper requires the live migration of a virtual machine across regions [10], FaaSKeeper can serve data from endpoints local to the user. Clients connect to the closest autoscaling storage in their region, minimizing access latency.

Map. Now that storage and computing tasks are decoupled, we can map them to services that fit best their access patterns and computational requirements.

Efficient reading of user data. ZooKeeper is optimized for high read performance, and the size of the user-defined node hierarchy can easily exceed a few gigabytes as each node stores up to 1 MB of data. Thus, we must use storage with strongly consistent, cheap, and fast read operations. The cost-performance analysis reveals that object storage is more efficient than key-value storage (Fig. 3). Storing large user data is 4.37x cheaper, and updating nodes scales much better with their size. Furthermore, read operations are billed per access and per 4 kB read in object and key-value storage, respectively, making the latter even more expensive for user nodes.

Efficient modifications of control data. The control data includes frequently modified watches, client and node status, and synchronized timestamps. FaaSKeeper must use atomic operationes and locks to support concurrent updates. We use the key-value store as object store is limited by expensive writes (Fig. 3a) and lack of synchronization primitives.

Communicate Finally, we define the FaaSKeeper functions that connect with both client and storage, using triggers that scale automatically with an increasing workload.

Vertical scaling. ZooKeeper improves throughput by *pipelining* client requests over a single TCP connection to the server. Requests are sent before previous operations finish, and the implementation ensures that operations from a single session are not reordered in the pipeline. However, serverless functions are designed for fine–grained invocations. Thus, FaaSKeeper employs cloud queues to batch invocations and continuously feeds the processing pipeline.

Horizontal scaling. ZooKeeper achieves high read scalability with more servers, but write scalability is limited by design with a single leader. Prior attempts to increase write performance focused on partitioning the ZooKeeper data tree [42, 43]. Instead, FaaSKeeper delegates requests from different client sessions to concurrently operating functions. While write requests of a single session are serialized, we exploit the parallelism of operations from different users. To determine global ordering and handle concurrent modifications to the same data node, FaaSKeeper uses synchronization primitives on the storage (Sec. 2.2). Thus, the scalability of read and write operations is bounded by storage throughput.

FaaS Service. To finalize the design of a serverless version of an IaaS service, we need to incorporate an elastic scaling model and ensure portability between clouds.

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 [9], 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 aim to use 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.

Cloud agnosticity. Vendor lock-in [44] is a serious limitation in serverless [45, 46], and dependency on queueing and storage services is of particular concern [47]. FaaS applications implemented in a specific cloud often include provider-specific solutions, requiring a redesign and reevaluation of the architecture when porting to another cloud. In a *cloud-agnostic* design, we define only the requirements for each service used 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 dependency to the implementation layer and allows moving between providers without a major system overhaul [48].

3.3 FaaSKeeper Design

Figures 4 and 5 present the system design of FaaSKeeper. In this Section, we describe each component of FaaSKeeper by first following the path of a write request through the system and then discussing all additional components. Compared to ZooKeeper, FaaSKeeper must overcome several challenges. First, we do not have a direct ordered channel to the server, such as a TCP connection, and we need to rely on queues for ordering, which imposes a higher latency on the system. The second challenge is that different storage solutions have vastly different costs and latencies – especially if multiple regions are considered, as seen in Fig. 3. This requires us to investigate and differentiate which components can use what storage type. Finally, having different types of storage for system and user data means we have separate data read and write paths. Therefore, the final challenge is extending serverless functions with logic to handle watch notifications, guarantee consistency, and handle fault tolerance.

Client. FaaSKeeper clients use an API similar to ZooKeeper. FaaSKeeper implements the same standard read and write operations as ZooKeeper, except for the `sync` operation. We implement both synchronous and asynchronous variants and deliver results in the usual FIFO order. *Read* operations are served through the regional replica of data storage (C1). *Write* operations are sent to a cloud queue (C2).

Queues. Each session is assigned a queue to send new requests and invoke processing functions (C2). The FIFO order guarantees the ordering of requests within a session. Queues batch requests to improve throughput and pipelining.

Writer. The single-writer system has been replaced with parallel *event* ***writer*** functions invoked by the writer queue with session 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 are not reordered.

Distributor. When the change caused by a writer triggers a watch, the notification is distributed (N1) parallel to replicating the updated node (W3). The extended timestamp system prevents clients from reading updated data before observing all consequences of the update. The distributor invokes a *free function* ***watch*** to deliver notifications to clients (N2) who registered the given watch (N3). Since hundreds of clients can register a single watch, using a serverless function allows us to adjust resource allocation to the workload.

Timestamps. To guarantee the consistency of updates, we define a total ordering with the "happened before" relation [49] using atomic counters.

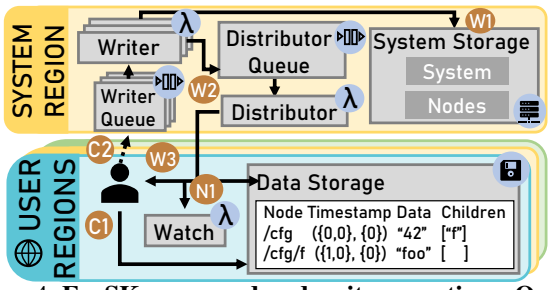


Figure 4: **FaaSKeeper read and write operations. Queues and timestamps provide strong consistency guarantees.**

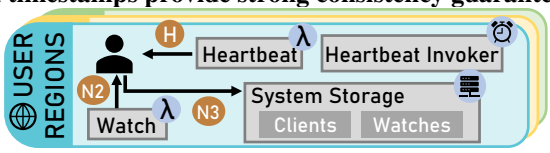


Figure 5: **Notifications and heartbeats in FaaSKeeper.**

	Requirements	AWS	Azure	Google	Other
Function	Free	✓	✓	✓	—
	Event	✓	✓	✓	
	Scheduled	✓	✓	✓	
User Store		S3	Blob Storage	Storage	Redis
	Consistency	✓	✓	✓	✓✳
	Throughput	✓	✓	✓	✓✳
System Store		DynamoDB	CosmosDB	Datastore	Redis
	Reliability	✓	✓	✓	✗
	Consistency	✓	✓	✓	✓✳
	Concurrency Primitives	Conditional Updates	Optimistic Locking	Transactions	Lua Scripts
Queue		SQS	Service Bus	Pub/Sub	
	FIFO	✓	✓	✓	—
	Serverless	✓	✓✳	✓	

Table 2: **Mapping FaaSKeeper design to cloud and user-managed services. ✳ indicates additional constraints.**

Storage. The system state is served by 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, data storage is optimized to handle read requests in a scalable and cost-efficient manner.

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). We replace the ZooKeeper heartbeat messages with periodically invoked heartbeat functions.

4 Building FaasKeeper

To implement FaaSKeeper, we need to map functions, queue, storage, and synchronization primitives to cloud services (Table 2). We detail the implementation choices made as the serverless design comes with a set of new challenges that are not present in ZooKeeper.

We select the AWS cloud and translate design concepts to cloud systems: system storage with synchronization primitives to DynamoDB tables, user data storage to S3 buck-

Algorithm 1 A pseudocode of the new writer function.

```
function WRITER(updates)
    for each client, node, op, args in updates do
        lock, oldData = LOCK(node)  ①
        if not ISVALID(op, args, oldData) then  ②
            NOTIFY(client, FAILURE)
            continue
        txid = DISTRIBUTORPUSH(client, lock, node, newData)  ③
        COMMITUNLOCK(lock, node, op, args, txid)  ④
```

Algorithm 2 A pseudocode of the new distributor function.

```
function DISTRIBUTOR(state, updates)
    for each region in parallel do
        for each client, lock, node, data, txid, writerID in updates do
            nodeStatus = GETNODE(node)  ❶
            if nodeStatus.transactions[0] != txid then
                if not TryCommit(lock, node) then  ❷
                    NOTIFY(client, FAILURE)
                    continue
            DATAUPDATE(region, data, s', epoch)  ❸
            w = WATCHES(node)
            INVOKEWATCH(region, w, WATCHCALLBACK)  ❹
            epoch[region] = epoch[region] + w
            NOTIFY(client, SUCCESS)
            POPTRANSACTION(node)  ❺
    WAITALL(WATCHCALLBACK)
function WATCHCALLBACK(epoch, region, w)
    epoch[region] = epoch[region] − w  ❻
```

ets, and FIFO queues to the SQS. We implement the four FaaSKeeper functions in 1,350 lines of Python code in AWS Lambda. Furthermore, we provide a client library with 1,400 lines of Python code with the relevant methods of the API specified by ZooKeeper [8]. Each component has a corresponding alternative in other cloud systems that provides the same semantics and guarantees, and storage can be improved with in-memory caches such as Redis.

Using the same ordering for components used in Section 3, we now discuss each component in detail and start with the path of a client performing a write operation. For each component we provide a description, discuss its implementation as well as our design goals, and where appropriate we also consider scalability concerns.

4.1 Client

Eliminating the ZooKeeper server from the data access path provides lower operating costs, but puts the responsibility of ordering read and write operations with watch notifications on the client. We implement a queue-like data structure that orders notifications and replies from the FaaSKeeper service with results of read operations. Operations return according to the semnatics of ZooKeeper: 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. Formal analysis of consistency guarantees can be found in the appendix (Sec. B).

Implementation. Each client runs three background threads to send requests to the system, manage incoming responses, and order results. Read operations are implemented with direct access to user storage.

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, extended timestamps are required to ensure that writes are linearized, and cloud queues are needed to replace TCP communication.

4.2 Writer

FaaSKeeper replaces the single-leader design of ZooKeeper with concurrently operating **writer functions** to enhance the system's reliability and performance. The writer function processes client 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. The results are propagated to the **distributor queue** to update user data storage (③). Finally, the new node version is secured in the system storage (④) and extended with the current transaction's index. This operation is combined with a lock release and applied conditionally, and no changes are made if the lock expires. At that point, the client request has been committed to the system (Z1), and pushing to the distributor queue before committing ensures that changes will be propagated to the user storage. In some operations, the ZooKeeper model requires locking more than one node — for example, creating a node also requires locking the parent node. There, the commit creates a transaction from multiple atomic operations that will fail or succeed simultaneously.

Each client sends requests to a single queue instance that invokes the processing function. A function concurrency limit of one instance per queue provides 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*.

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, and (e) assigns monotonically increasing values to consecutive messages (*txid*). 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 [50] as it performs better than DynamoDB Streams (Sec. 5.2).

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

4.3 Distributor

A distributor queue is necessary to ensure that changes in user data stores are not reordered since concurrent updates could violate consistency (Z3), and notifications must be delivered in order (Z3). Committing changes to the user-visible storage must be serialized in ZooKeeper's consistency model, and clients cannot observe newer data before receiving watch notifications (Z4). FaaSKeeper uses the additional region-wide *epoch* counter to ensure that the client's consistent view of the data is not affected.

Function. The *distributor* function (Algorithm 2) delivers updates to each user storage. Updates cannot be reordered for a single storage unit, but the process is parallelized across regions. Since the writer cannot push to the distributor queue and commit the node atomically, distributor verifies that the node has been committed successfully (❷). In the case of the writer's failure or unlikely interleaving between both functions, the distributor tries to commit nodes when possible to improve the system availability. Otherwise, the update is rejected - the request has never been committed, and a failure of one writer function does not impact the system consistency. Then, the data is replicated to user storage (❸), and the distributor sends watch notifications (❹). Once all steps are completed, the current transaction is removed from the node to ensure that all notifications are always delivered (❺). In the case of failure, the queue will retry the invocation automatically. A transaction index ensures that the distribution of updates can be retried.

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 the node's children, the distributor function needs to download user node data to conduct the update operation.

Notifications. The clients must not see new data before receiving a notification preceding it. If we stall updates until all notifications are delivered, this would put severe performance limitations. In our implementation, the path of reads and writes are different, a significant departure from ZooKeeper, where all reads and writes are processed by the same entity. Instead, we use *epoch* counters to associate updates with active watches and help clients detect when 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, an independent function is invoked to deliver notifications in parallel (❹), and the counter is readjusted once all notifications are delivered. As clients are unaware of pending notifications, they use the *epoch* counter to determine if any of the watches registered by the client were being delivered when an update to the read node was applied. In such a case, the read operation must be stalled until the pending notification is delivered.

Design goal. The distributor guarantees 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.

4.4 Storage

We use tables in the cloud-native key-value database for the **system storage**, containing the current timestamp and all active sessions, and we store the list of all data nodes to allow lock operations by writer functions. **Data storage** is indexed by node paths, and each object corresponds to a single node. The object contains user node data, modification timestamps, and a list of node children. Eventually consistent reads neither guarantee read–your–write consistency [28], nor consider a dependency between different writes, breaking ZooKeeper guarantees (Linearized Writes Z2, Single System Image Z3). Therefore, we require strong consistency, although it can be more expensive.

Synchronization Primitives are implemented with *update expressions* of DynamoDB system storage [51]. Each operation requires a single write, and the correctness is guaranteed by the atomicity of updates to a single item.

Timestamps are used to provide an order over system transactions. The system ***state* counter** *txid* is an integer that represents the **timestamp** of each change in FaaSKeeper, similar to the *zxid* state counter in ZooKeeper and provides total order over the system. The ***epoch* counter** is specific to FaaSKeeper and contains watch notifications pending while the transaction represented by the state counter was in progress. In a system with decentralized processing of write and read requests, the epoch counter provides an ordering between notifications and changes applied to the system. Counters are implemented using the atomic counters and lists (Sec. 2.2).

Design goal. ZooKeeper achieves high availability with multiple replicas of the dataset. We achieve the same goal by using automatically replicated and scalable cloud storage.

4.5 Heartbeat

In addition to ordering guarantees, sessions play another significant role in ZooKeeper: their status defines lifetime of ephemeral nodes. We replace the heartbeat messages with the **heartbeat function** to discover client disappearance and remove ephemeral nodes.

Implementation. The cloud system periodically invokes the function which sends in parallel heartbeat messages to clients that own ephemeral nodes. If a client does not respond before a timeout, the 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}.

Design goal. The verification of client status does not need a persistently allocated server, and FaaSKeeper replaces it

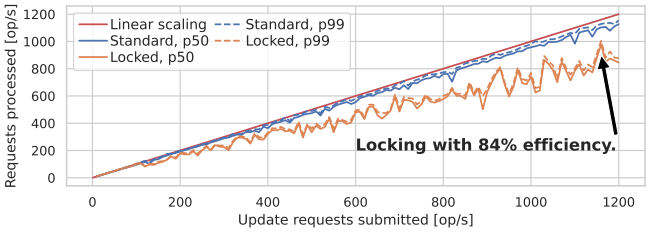
with a serverless function that scales accordingly.

4.6 Compatibility with ZooKeeper

Our implementation is standalone and does not reuse the server-centric ZooKeeper codebase since it would be affected by large cold startup overheads in FaaS [23, 52]. We offer a compatible interface for existing applications by modeling our API after kazoo [53], a Python client for ZooKeeper. While FaaSKeeper aims to provide a consistency model and interface compatible 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. The size restrictions of 400 and 256 kB in DynamoDB, respectively, can be avoided by splitting larger nodes and using temporary S3 objects. Furthermore, Zookeeper clients can define node permissions with 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.

Primitive	Size	**Min**	**p50**	**p95**	**p99**	**Max**
Regular DynamoDB **write**	1 kB	3.95	4.35	4.79	6.33	60.26
	64 kB	6.54	66.31	70.28	77.23	121.64
Timed lock **acquire**	1 kB	6.13	6.8	8.13	14.14	65.32
	64 kB	7.82	67.16	72.71	90.56	177.02
Timed lock **release**	1 kB	6.03	6.62	7.94	12.52	78.44
	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 **append**	1	5.14	5.89	8.0	10.71	21.12
	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 6: **Synchronization primitives on AWS DynamoDB.**

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, answering the following questions:

§ 5.1 Are synchronization primitives efficient?
§ 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?
§ 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. Benchmarks use 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

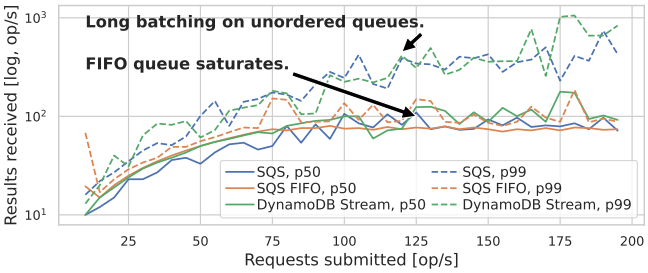
The serverless synchronization primitives are a fundamental building block for FaaSKeeper operations that allow concurrent, safe updates. Primitives are implemented using conditional update expressions of DynamoDB [51], and we evaluate the overheads and scalability on this datastore system.

Latency. We evaluate the latency of each operation by performing 1000 repetitions on warmed-up data and present results in Table 6a. 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 adding a varying number of items of 1 kB size. This allows users to efficiently add new watches by extending the list of watches in storage with a single operation.

Throughput. Timed locks allow FaaSKeeper to conduct independent updates concurrently. We evaluate a pair of read and write operations, and compare our locks with a version without a 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 (Fig. 6b). 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 [54], and the throughput of serverless operations on DynamoDB is limited by Lambda's parallelism and not by storage scalability [55].

	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 7: **Function invocations with serverless queues.**

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

5.2 Serverless Queues

Queues improve the pipelined writing process by batching requests and are necessary to provide ordering (Sec. 4.3) 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 select a queue that adds minimal invocation overhead and allows to achieve sufficient throughput. For SQS [56], we enable the FIFO property that comes with the restriction of a maximum batch size of 10. We compare against the standard version to estimate the potential overheads of of small batch sizes. For AWS DynamoDB streams, we configure database sharding to guarantee that all new items in a table are processed in order [57]. We restrict the function's concurrency to permit only one instance at a time.

Latency. First, we measure the end-to-end latency by triggering an empty a 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 the 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 7a. Surprisingly, the FIFO queue achieves the lowest latency and is significantly faster than a direct Lambda 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 (Fig. 7b). FIFO queues saturate at the level of a hundred requests per second. Meanwhile, DynamoDB and standard SQS experience huge variances, 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 scalability concerns partially.

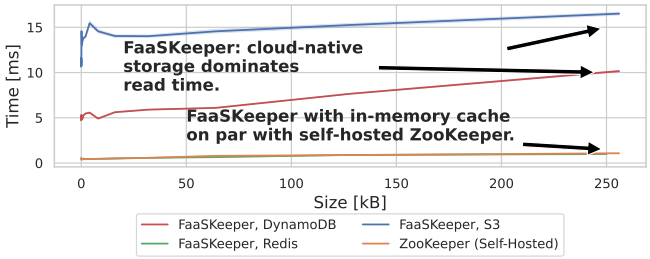


Figure 8: **Read operations in FaaSKeeper and ZooKeeper.**

Cost. 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.

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 Redis (`t3.small` VM), and compare FaaSKeeper against ZooKeeper. We repeat the measurements 100 times for each node size and present results in Figure 8.

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 matches its performance with an in-memory store.

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

FaaSKeeper offers fast reads whose performance is bounded by the latency and throughput of the underlying cloud storage, with a stable cost proportional to workload.

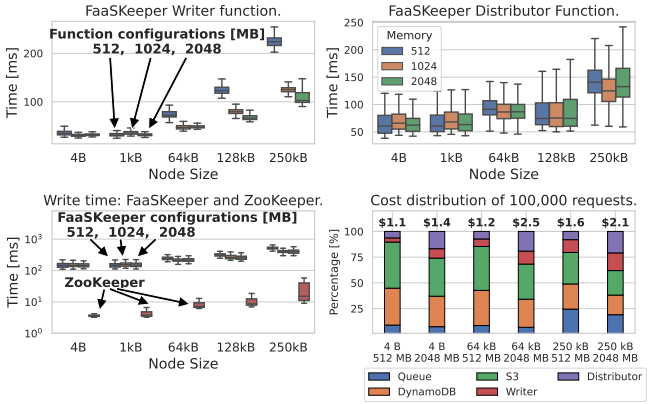


Figure 9: **Write operations in FaaSKeeper and ZooKeeper.**

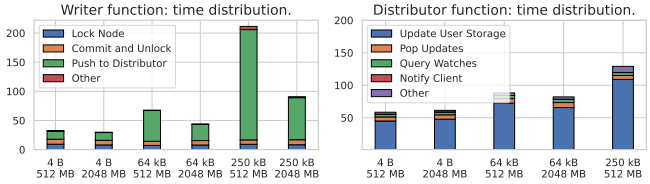


Figure 10: **Time distribution of FaaSKeeper functions.**

5.4 Write Operations

We evaluate the performance and cost of writing in FaaSKeeper and compare our framework against ZooKeeper. We measure `set_data` operation that replaces node contents with base64-encoded data of different sizes (Figure 9). The write latency of FaaSKeeper consists of function runtimes and the overheads of queue-based invocations. ZooKeeper achieves lower write latency because it benefits from a direct connection to the client and can perform some operations on the local memory. In addition to measuring total operation time as visible by the client, we study the execution times of writer and distributor functions. In FaaSKeeper, storage operations are responsible for the 40-80% of writing cost. The cost of computing with functions is noticeably lower, even though the CPU time of a serverless function is 8x more expensive than in a VM. Both functions use no more than 100 MB of memory but require large allocations to increase I/O performance [22], leading to increased cost and resource underutilization.

To locate the bottleneck of parallel processing in FaaSKeeper, we inspect where functions spend time. The results in Figure 10 show that the impact of synchronization operations is limited, and the runtime of distributor and writer functions are dominated by moving data to queues and storage. This impacts both the latency and cost, as there is no *yield* operation in serverless - functions waiting on I/O and external services keep consuming resources and accruing costs.

Finally, we examine the tail latency of the most important operations (Table 3). We observe significant performance degradation at the tail percentiles for pushing to the distributor queue in *writer* and updating nodes in S3 in *distributor*.

	Writer	Size	Min	p50	p90	p95	p99
Writer	Total	4B	27.29	31.81	38.55	41.88	58.78
		250 kB	30.24	102.53	142.35	163.15	183.49
	Lock	4B	7.38	8.02	9.47	12.69	26.8
		250 kB	6.77	8.36	15.38	17.79	28.48
	Push	4B	9.65	13.35	15.55	17.28	38.15
		250 kB	62.73	72.18	96.82	118.62	148.61
	Commit	4B	7.31	7.93	9.41	11.91	26.83
		250 kB	6.61	8.59	14.31	18.81	32.83
Distributor	Total	4B	42.02	62.16	92.01	103.65	138.28
		250 kB	58.94	132.62	213.5	294.01	465.47
	Get Node	4	4.67	5.09	5.68	6.92	11.83
		250 kB	4.58	4.97	7.31	11.13	19.83
	Update Node	4	24.4	42.73	70.7	84.94	118.13
		250 kB	32.51	102.07	183.17	265.42	432.92
	Watch Query	4	3.88	4.48	5.45	7.0	28.64
		250 kB	4.68	5.13	6.76	7.59	18.38

Table 3: **Variability of functions performance, 2048 MB.**

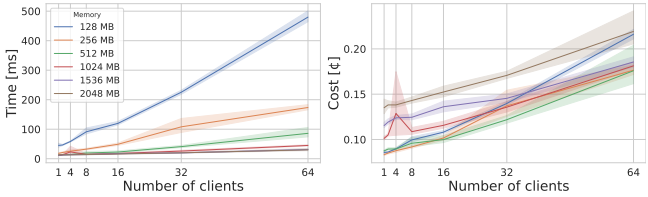


Figure 11: **Heartbeat function performance and cost.**

The performance of write operations is bounded by data transmission to the distributor queue and object storage, motivating 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 removal of ephemeral ZooKeeper resources. We estimate the time and resources needed by FaaSKeeper to periodically launch the `heartbeat` function and verify status of clients owning ephemeral nodes . We present results averaged from 100 invocations in Figure 11. The execution time decreases with the memory allocation, corresponding with previous findings on I/O in serverless [22, 23].

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.

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

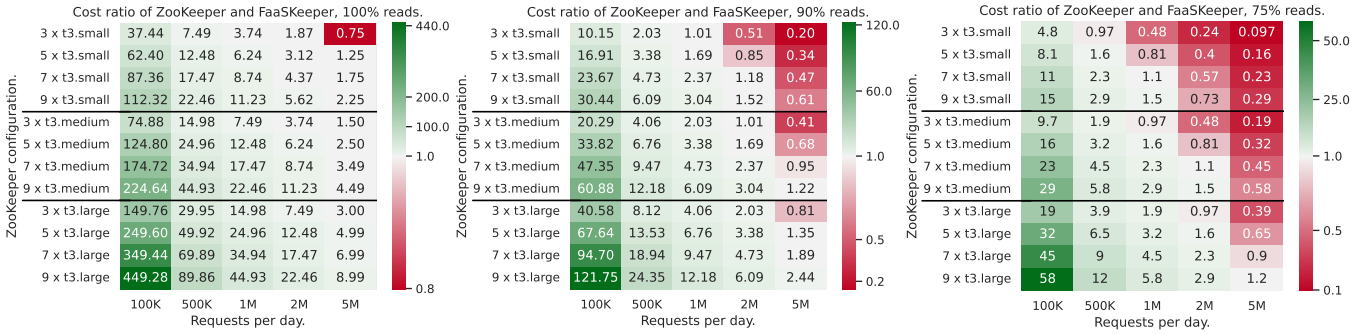


Figure 12: **Cost of ZooKeeper and FaaSKeeper, running a workload mix of 1 kB reads and writes with `set_data`.**

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	$\lceil \frac{s}{64} \rceil \cdot 0.5 \cdot 10^{-6}$
$F_{W/D}(s)$	Execution of writer and distributor function.	Linear models.

Table 4: **Parameters of FaaSKeeper cost model.**

6 FaaSKeeper Cost

The most important evaluation compares the price of running an elastic FaaSKeeper instance to Zookeeper. We consider a scenario of 512 MB, with reads and writes of 1 kB writes, and the optimistic case that we experience no failures and therefore no retries.

FaaSKeeper cost We focus on read and write operations of s kilobytes, as the daily monitoring costs are low. Watch and heartbeat functions add charges only when notifications and ephemeral nodes are used. We model the cost of modifying node data (`set_data` in ZooKeeper), and summarize model parameters in Table 4.

Reading. The cost of operation is limited to storage access.

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

A workload of 100,000 read operations costs $0.04.

Writing. The cost of writing is separated into computing and storing data: two queue operations and function executions, synchronization in the `writer` and `distributor`, and writing data to the user store. For both functions, we use regression to estimate linear cost models using data from Sec. 5.4, with R^2 scores of 0.98 (writer) and 0.84 (distributor).

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

A workload of 100,000 write operations costs $1.12.

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 the block storage `gp3` attached to the EC2 virtual machines hosting ZooKeeper.

ZooKeeper cost The cost of ZooKeeper is constant and includes the cost of a persistent allocation of virtual machines. The smallest number of virtual machines is three. However, a single machine with an attached EBS block storage has an annual durability of 99.9%. To match the annual durability of S3 used as the user store in FaaSKeeper (11 9's), the ZooKeeper ensemble requires nine machines.

Depending on the VM selection, the daily cost changes from $0.5 on `t3.small`, through $1 on the `t3.medium` used for our experiments, up to $2 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 between $4.8 (three VMs) and $14.4 (nine VMs).

Comparison We compare ZooKeeper's cost against FaaSKeeper with different read–to–write scenarios, using 1kB writes and functions configured with 512 MB of memory, and present results in Figure 12. In high–read–to–write scenarios for which ZooKeeper has been designed, FaaSKeeper can process between 1 and 3.75 million requests daily before the costs are equal to the smallest possible ZooKeeper deployment. Since many user nodes do not contain large amounts 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 limit expensive computing time to processing writes only. Furthermore, we can *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 Building Serverless Services

We now discuss the main issues we encountered while building FaasKeeper and highlight the limitations of serverless in its current form. We collect several requirements that cloud providers could easily support and pave the way for future improvements. They would make complex systems such as FaaSKeeper more efficient and serverless services in general more performant — simplifying their implementation and increasing adoption. We finally discuss how well these requirements are supported in current cloud architectures.

7.1 Areas of improvement

Using the lessons learned while creating FaaSKeeper, we propose a list of requirements for serverless environments that

would allow complex services to flourish. However, many of these requirements are not limited to FaaSKeeper, and improvements will benefit other serverless applications, such as microservices [4] and serverless ML [58, 59].

Requirement #1: Fast invocations. Invocation overheads dominate the execution time of short-running functions [22] and prohibit FaaS processing with performance comparable to non-serverless applications. ZooKeeper often requires multiple round trips to finish an operation, and when each one takes milliseconds rather than microseconds, the overheads quickly accrue, as seen in Fig. 8.

Requirement #2: Exception handling. The user cannot control asynchronous function invocations such as notifications of completed operations (Sec. 4.3). We envision this should be solved via *exception handlers*, allowing for easier and more efficient error handling.

Requirement #3: Synchronization primitives. To efficiently implement distributed applications, serverless needs synchronization, such as locks and atomic operations (Sec. 2.2).

Requirement #4: FIFO Queues. Serverless functions require queues to support the ordering and reliability of invocations (Sec. 4.3). 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.

Requirement #5: Statefulness. *Stateful* functions are necessary for some use cases, and FaaS systems should support a reliable function state with low-latency access (Sec. 4.2).

Requirement #6: Partial updates. To increase the efficiency of write operations, cloud storage could support partial updates where data is written at a user-defined offset to the specified object (Sec. 4.2).

Requirement #7: Outbound channels. While the trigger system provides *inbound* communication, functions lack an ordered, push-based, and fast *outbound* communication channel. Such a channel would significantly simplify the design of serverless services such as FaaSKeeper (Sec. 4.2).

7.2 Discussion

Can serverless systems support our requirements? We specify seven requirements to define features currently missing in commercial FaaS systems that are necessary to support distributed, stateful, and scalable applications. The requirements align with the major serverless challenges [21, 60] and are supported in research FaaS platforms. Emerging systems provide microsecond-scale latency [24, 61]. New storage systems satisfy the latency, consistency, and flexibility requirements of functions [29, 30, 62, 63]. Furthermore, stateful serverless is becoming the new norm in clouds [55, 64–67]. Finally, we note that research systems can support many of our requirements already: Cloudburst (R1, R5) [63], PraaS (R1, R5, R7) [67], Boki (R3-R5) [65].

What are the design trade-offs of FaaSKeeper? FaaSKeeper achieves elastic scaling and a serverless price model by accepting the increased latency of FaaS systems. However, performance overheads are isolated to specific services and their impact will decrease with the adoption of more efficient serverless platforms. FaaSKeeper could offer read latency on par with ZooKeeper by incorporating an in-memory database as the user endpoint [29]: these are only now becoming available in a serverless billing model [31, 32]. The increased processing time of write requests is caused primarily by performance variations of cloud queues and object storage.

8 Related Work

Serverless for Storage Wang et al. [31] use functions for elastic in-memory cache. DynamoDB is used in transactional workflows with locks in Beldi [55] and in a fault-tolerance shim AFT [54]. In contrast, FaaSKeeper is designed as a service and not a backend for serverless functions. We offer coordination for general-purpose applications while optimizing resource allocation.

Elastic Storage Cloud-native storage is known for elastic implementations that scales with changes in workload [68]. Examples include reconfiguration controllers [69, 70], workload predictors [71], and latency monitoring [72]. PolarDB is an example of a disaggregated database that offers a serverless billing model [15]. However, ZooKeeper requires autoscaling procedures that integrate the state ordering guarantees. FaaSKeeper achieves that by using the auto-provisioning of serverless functions and databases.

ZooKeeper Other authors explored different approaches to improve the performance and availability of ZooKeeper. Stewart et al. [73] replicated ZooKeeper contents on multiple nodes to provide predictable access latencies. Shen et al. [10] proposed live VM migration for geographical reconfiguration of ZooKeeper. The performance of ZooKeeper has been improved with hardware implementations, using FPGAs [74] and offloading to network adapters [75] with PsPIN [76].

9 Conclusions

As the tools and mechanisms of cloud computing adapt to the needs of an ever-growing FaaS landscape, creating a powerful, fast, and efficient serverless application is becoming possible. In this work, we present FaaSKeeper, a serverless coordination service offering the same consistency model and interface as Zookeeper. FaaSKeeper allows for an elastic deployment that matches system activity, reducing the cost of some configurations by a factor of up to 450x. We discuss the lessons learned in creating FaaSKeeper, and identify seven requirements that clouds should fulfill to ensure functionality and performance.

Acknowledgments

This project received funding from EuroHPC-JU under grant agreements DEEP-SEA, No 95560 and RED-SEA, No 055776. We thank Amazon Web Services for supporting this research with credits through the AWS Cloud Credit for Research program.

A ZooKeeper

Below we summarize the provided consistency requirements [8, 36, 37] 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 operation results are propagated to the distributor queue before the commit. The queue triggers the distributor function and retries it upon failure, guaranteeing the eventual propagation of changes to all data replicas. Since the distributor function verifies node status before propagating changes (❶ in Alg. 2), incorrect operations do not affect 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 a 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 a 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, there are two possible situations: (a) a watch notification relevant to the client was active but not yet delivered (❹ in Alg. 2) before storing v (❷ 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 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 watch w is notified. 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: 2022-10-15.

[6] 2020. Azure Cosmos DB serverless now in preview. https://devblogs.microsoft.com/cosmosdb/serverless-preview/. Accessed: 2022-10-15.

[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: 2022-10-15.

[8] 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.

[9] 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

[10] 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

[11] 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

[12] 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.

[13] 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

[14] 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

[15] 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

[16] 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

[17] 2014. AWS Lambda. https://aws.amazon.com/lambda/. Accessed: 2022-10-15.

[18] 2016. Azure Functions. https://azure.microsoft.com/en-us/services/functions/. Accessed: 2022-10-15.

[19] 2017. Google Cloud Functions. https://cloud.google.com/functions/. Accessed: 2022-10-15.

[20] 2016. IBM Cloud Functions. https://cloud.ibm.com/functions/. Accessed: 2022-10-15.

[21] 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

[22] 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

[23] 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.

[24] 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]

[25] 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

[26] 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.

[27] 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

[28] Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (2009), 40–44.

[29] 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

[30] 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.

[31] 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

[32] 2021. Upstash: Serverless Database for Redis. https://upstash.com/redis. Accessed: 2022-01-30.

[33] Bruce Jay Nelson. 1981. *Remote procedure call*. Carnegie Mellon University.

[34] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[35] Cary Gray and David Cheriton. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review* 23, 5 (1989), 202–210.

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

[37] 2020. ZooKeeper Programmer's Guide. https://zookeeper.apache.org/doc/current/zookeeperProgrammers.html. Accessed: 2022-10-15.

[38] 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

[39] 2020. ZooKeeper Dynamic Reconfiguration. https://zookeeper.apache.org/doc/current/zookeeperReconfig.html. Accessed: 2022-10-15.

[40] 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

[41] 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.

[42] 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

[43] 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

[44] 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

[45] 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

[46] 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

[47] 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

[48] 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.

[49] 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

[50] 2021. Using AWS Lambda with Amazon SQS. https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html. Accessed: 2022-10-15.

[51] 2021. Using Expressions in DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Expressions.html. Accessed: 2022-10-15.

[52] 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

[53] 2021. Kazoo: high-level Python library for ZooKeeper. https://github.com/python-zk/kazoo. Accessed: 2022-10-15.

[54] 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

[55] 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

[56] 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: 2022-10-15.

[57] 2020. Using AWS Lambda with Amazon DynamoDB. https://docs.amazonaws.cn/en_us/lambda/latest/dg/with-ddb.html. Accessed: 2022-10-15.

[58] 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

[59] 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

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

[61] 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

[62] 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

[63] 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

[64] 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

[65] 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

[66] 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

[67] 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/index.php?pub=458 Accessed: 2022-10-15.

[68] 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

[69] 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

[70] 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

[71] 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

[72] 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

[73] 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

[74] 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

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

[76] 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).