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

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

Alexandru Calotoiu
ETH Zurich
Switzerland
alexandru.calotoiu@.inf.ethz.ch

Pengyu Zhou
University of Toronto
Canada
ericpengyu.zhou@mail.utoronto.ca

Konstantin Taranov
Microsoft
Switzerland
kotaranov@microsoft.com

Torsten Hoefler
ETH Zurich
Switzerland
htor@inf.ethz.ch

ABSTRACT

FaaS (Function-as-a-Service) revolutionized cloud computing by replacing persistent virtual machines with dynamically allocated resources. This shift trades locality and statefulness for a pay-as-you-go model more suited to variable and infrequent workloads. However, the main challenge is to adapt services to the serverless paradigm while meeting functional, performance, and consistency requirements. In this work, we push the boundaries of FaaS computing by designing a serverless variant of ZooKeeper, a centralized coordination service with a safe and wait-free consensus mechanism. We define synchronization primitives to extend the capabilities of scalable cloud storage and outline a set of requirements for efficient computing with serverless. In FaaSKeeper, the first coordination service built on serverless functions and cloud-native services, we explore the limitations of serverless offerings and propose improvements essential for complex and latency-sensitive applications. We share serverless design lessons based on our experiences of implementing a ZooKeeper model deployable to clouds today. FaaSKeeper maintains the same consistency guarantees and interface as ZooKeeper, with a serverless price model that lowers costs up to 110-719x on infrequent workloads.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → *Software architectures*; **Cloud computing**.

KEYWORDS

serverless, function-as-a-service, faas, cloud computing, zookeeper

ACM Reference Format:
Marcin Copik, Alexandru Calotoiu, Pengyu Zhou, Konstantin Taranov, and Torsten Hoefler. 2024. FaaSKeeper: Learning from Building Serverless Services with ZooKeeper as an Example. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24), June 3–7, 2024, Pisa, Italy.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3625549.3658661

	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 (⏫⏬) and reliability (🛡), with ZooKeeper's consistency (⟳), push notifications (✉), and support for concurrency and fault tolerance (⤧🐞).

FaaSKeeper implementation: https://github.com/spcl/faaskeeper
FaaSKeeper Artifact: https://github.com/spcl/faaskeeper-paper-artifact
Extended paper version: https://arxiv.org/abs/2203.14859

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 [14, 19, 50]. 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 [46], and even databases adapt on-demand offerings to handle infrequent workloads more efficiently [1, 2, 7].

Apache ZooKeeper [40] 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 key-value storage, ZooKeeper adds semantics of total order with linearizable writes, atomic updates, and ordered push notifications (Table 1).

Cloud services are expected to match the temporal and geographical variability of production workloads [30, 36, 63]. Workloads are often bursty and experience rapid changes: maximum system utilization can be multiple times higher than even the 99th percentile [36, 70]. However, the static ZooKeeper architecture make the readjustment to the workload difficult, and ZooKeeper is often underutilized in practical deployments (Section 5.1). 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 cloud storage could adapt to diurnal changes in workload and handle thousands of requests at a lower cost. A *serverless* service with the same consistency as ZooKeeper 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, which lack fast communication channels, ordering, and statefulness. First, we decouple the system from the application state and compute from storage tasks [23, 32, 33], Similar to past results in building a database on cloud storage [20], we build a serverless architecture on top of auto-scalable storage, focusing on a *cloud-native* design that requires no user-managed and custom solutions, but is instead deployable to clouds available today (Section 3). We focus on the semantics of building components and abstract away differences in interfaces and services, helping design *cloud-agnostic* systems that are portable between clouds (Section 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). We implement ZooKeeper's model and API in FaaS, with a prototype of the provider-agnostic system on AWS and GCP (Section 4), demonstrating consensus in a serverless application on top of consistent and replicated cloud storage. FaaSKeeper offers a pay-as-you-go cost model while upholding consistency and ordering properties (Section 5.3.4). Another goal of this paper is to demonstrate the fundamental trade-offs of moving data-intensive services to new cloud paradigms such as serverless. We explore design choices in storing and updating cloud data, proving the cost and resource efficiency of serverless while enumerating limitations of current serverless offerings (Section 6).

In summary, we make the following contributions:

- Exploration of challenges and limitations in serverless and lessons for designing cloud-native services with synchronization, message ordering, and event-based communication.
- The first complex serverless solution that offers the same level of service as its IaaS counterpart without provisioning.
- An API-compatible implementation of the ZooKeeper consistency model that achieves up to 60 times lower costs against the smallest ZooKeeper deployment.

2 BACKGROUND AND MOTIVATION

While serverless systems differ between cloud providers, they can represented as fundamental building blocks needed to design serverless services (Sec. 2.1). These are necessary to implement in serverless the distributed coordination model of ZooKeeper (Sec. 2.2).

2.1 Serverless Components

We define function models and propose synchronization primitives necessary for serverless services, including FaaSKeeper. Then, we propose extensions to current serverless systems that providers could offer, enabling more efficient and robust serverless services.

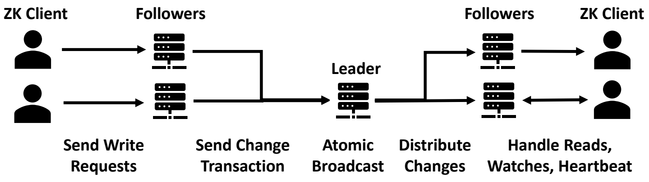
Cloud Storage. Cloud operators offer storage solutions that differ in elasticity, costs, reliability, and performance.
Object. Object storage is designed to store large amounts of data for a long time while providing high throughput and durability. The cloud operator manages replication across multiple instances in physically and geographically separated data centers, providing high availability and reliability. Modern object stores offer strong consistency on read operations [22], guaranteeing that successful writes are immediately visible to other clients. The billing model is linear in the data amount and the number of performed operations.
Key-Value. Nonrelational databases offer serverless billing where the costs depend only on the stored data and operations performed. They 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 [51, 52] is designed to meet serverless requirements for scalability and flexibility. In-memory caches bring lower latency and are being adapted to serverless scalability [10, 72].

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* [58]. 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 each trigger to have configurable batching and concurrency of invocations, as the former improves throughput and the latter is essential to ensure FIFO order. Semantically, we interpret these functions as remote *asynchronous callbacks* to events.

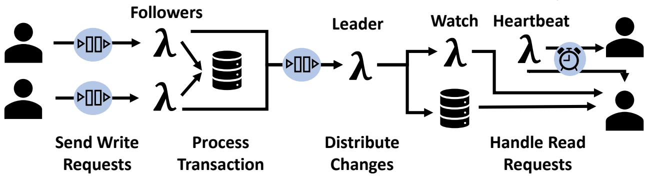
Functions can be launched to perform regular routines such as garbage collection and detecting system faults. **Scheduled functions** are the serverless counterpart of a `cron` job in Unix-based operating systems. In the event of an unexpected failure, the cloud should provide a retry policy with a finite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

Synchronization Primitives Functions operating in parallel require fundamental synchronization primitives to safely modify a global state [39], as it is the case in FaaSKeeper. In serverless, such primitives operate on storage instead of shared memory.

A **timed lock** extends a regular lock with a limited holding time, similarly to leases [37]. 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 or when the difference between the



(a) ZooKeeper processes requests by a set of servers *following* a leader.



(b) FaaSKeeper maps ZooKeeper to stateless functions and storage.

Figure 1: From ZooKeeper servers to functions in FaaSKeeper.

existing and new timestamp 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.

2.2 ZooKeeper

ZooKeeper guarantees data persistence and high read performance by allocating replicas of the entire system on multiple servers [5, 40, 48]. ZooKeeper ensemble consists of *followers* and an elected leader (Fig. 1a), whose roles are processing write requests with the help of the ZAB atomic broadcast protocol [49]. The smallest ZooKeeper deployment uses three servers, where two are required to accept a change and failure of one can be tolerated. While adding more servers increases reliability, it hurts write performance.

In ZooKeeper, changing the deployment size involved *rolling restarts*, a manual and error-prone process [4]. While it has been later enhanced with dynamic reconfiguration [65], it still requires manual effort [4], and reconfiguration causes significant performance degradation when deploying across geographical regions [63].

ZooKeeper splits the responsibilities between the client library, follower 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 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*. Clients register *watches* on a node to receive a push notification when that node changes. Finally, clients exchange heartbeat messages with a server to keep the session alive.

Consistency. 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 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).

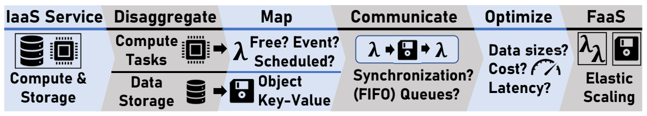


Figure 2: Workflow for designing a serverless service. The decoupled compute and storage are connected to cloud services, and later optimized for cost and performance (Sec. 4).

3 FROM ZOOKEEPER TO FAASKEEPER

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 FaaS foundation. Therefore, we designed FaaSKeeper from the ground up to replicate the complex ZooKeeper functionality and overcome the inherent challenges of the serverless world. We follow a general workflow for turning an IaaS system into FaaS (Fig. 2): disaggregate compute and data, replace servers with cloud storage and stateless functions, and let the new components communicate. By decomposing the system into separate cloud services - functions, storage, and queues (Fig. 1b) - we adjust the resource consumption to throughput and shut down processing instances when they are no longer needed. While the transition of many applications and microservices to serverless might be straightforward, ZooKeeper has non-trivial ordering and data visibility requirements. Thus, we design custom synchronization, queue communication, and new algorithms to implement ZooKeeper's data model in the serverless world.

We describe each component of FaaSKeeper by following the path of a client performing a write operation, then discuss all additional components. We map the computational logic of *follower* and *leader* servers to separate functions (Sec. 3.1, 3.2). Since functions are stateless, the entire state of a system must be stored in a replicated cloud storage (Sec. 3.3). Serverless has new challenges: it does not have direct and ordered communication channels such as TCP connection, and we need to use ordered cloud queues and extend functions with logic to handle watch notifications and guarantee consistency. Different types of storage for system and user data means we have separate data read and write paths, requiring extended system counters (Sec. 3.4) and additional ordering in FaaSKeeper client library (Sec. 3.5). Finally, the periodic heartbeat verification is mapped to a scheduled function (Sec. 3.6).

3.1 Follower

FaaSKeeper replaces ZooKeeper servers preparing update transactions with concurrently operating **follower functions**. A cloud queue invokes functions, and the function processes requests of each client in a FIFO order (Algorithm 1). The follower 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 validated and confirmed changes are propagated through a FIFO queue to the *event **leader*** function (③), ensuring that the changes

Algorithm 1 A pseudocode of the new follower function.

```
function FOLLOWER(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
        $txid$ = LEADERPUSH(client, lock, node, newData)  ③
        COMMITUNLOCK(lock, node, op, args, $txid$)  ④
```

Algorithm 2 A pseudocode of the new leader function.

```
function LEADER(state, updates)
    for each region in parallel do
        for each client, lock, node, data, $txid$, followerID 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
```

are not reordered. 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 queue before committing ensures that the leader will propagate the changes to the storage visible by users. 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.

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 follower function is a sequence of operations on the system storage that can be *pipelined.*

Implementation. Each client session is assigned a queue to send new requests and invoke processing functions. 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.

3.2 Leader

The *leader* function (Algorithm 2) delivers updates to the cloud storage visible by users, similar to ZooKeeper's leader that distributes confirmed changes to servers handling read requests. A FIFO queue between *followers* and *leader* 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 (Z4). Since a follower cannot push to the queue and commit the node atomically, leader verifies that the node has been committed successfully (❷). In the case of the follower's failure or unlikely interleaving between both functions, the leader 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 follower function does not impact the system consistency. Then, the data is replicated to user storage (❸), and the leader sends watch notifications (❹). 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). However, this process can be parallelized across cloud regions. Once all steps are completed, the current transaction is removed from the node (❺). The per-node transaction index allows the cloud queue to retry the function invocation automatically after a failure.

Implementation. When committing data to cloud storage, we attempt to update only changed data to avoid unnecessary costs and network traffic. While early visions of object storage assumed that write operations can access arbitrary offsets in an object [15, 35], these are not widely available in modern clouds. Thus, even if a change involves only part of node's metadata, the leader function needs to download entire node first to conduct the update operation.

3.3 Storage

ZooKeeper achieves high availability with multiple replicas of the dataset. We achieve the same goal by using automatically replicated and scalable cloud storage, which helps us to simplify the control plane of our system. We distinguish two types of storage in FaaSKeeper: **system storage** used by followers and leader to coordinate and modify the system state, and **user storage** optimized to handle read requests from FaaSKeeper clients in a scalable and cost-efficient manner. **System storage** contains the current timestamp, all active sessions, and the list of all data nodes to allow locking by follower functions. **Data storage** is indexed by node paths, and each item corresponds to a single ZooKeeper node, with user data, modification timestamps, and a list of node children.

When selecting an instance of cloud storage for FaaSKeeper, we consider not only the cost and performance but also the technical capabilities of the services. Eventually consistent reads neither guarantee read–your–write consistency [71], nor consider a dependency between different writes, breaking ZooKeeper guarantees (Linearized Writes Z2, Single System Image Z3). Therefore, we must use cloud storage that supports strongly consistent reads.

Synchronization Primitives are implemented in system storage and require that each update to a single item is atomic. Atomic counters are implemented as a single number, and an update adds a numerical constant to the current value. Atomic lists are represented as a list of numbers with an update that adds and removes elements from the list. Finally, the timed lock uses conditional updates to verify that each locking and unlocking operation does not invalidate any existing locks. A lock is stored in the node as a timestamp, allowing other functions to override an expired lock and prevent deadlocks caused by a failure in a function. Each operation requires a single write to a single item.

Timestamps 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. Counters are implemented using the atomic counters and lists (Sec. 2.1).

3.4 Watch Notifications

When a ZooKeeper client changes a node that has watches attached to it, the system sends a notification to watch owners, who must not see new data before receiving a notification. In serverless, the path of reads and writes are different, a significant departure from ZooKeeper, where all reads and writes are processed by the same entity, and the underlying TCP connection guarantees order. Instead, we use the additional region-wide **epoch counters** to provide an ordering between notifications and changes applied to the system. Each watch is assigned a unique identifier, and multiple clients can be assigned to a single watch instance. When a node is updated, the leader attaches to it the epoch counter containing the identifiers of all watch notifications still being delivered while the update was taking place. Once the client library finds the counter in a read node, it checks the epoch counter for any of the watches registered by this client. In such a case, the read operation must be stalled until the pending notification is delivered, preventing the client from seeing updated data before observing all preceding notifications.

3.5 Client

FaaSKeeper implements the same standard read and write operations as ZooKeeper and offers clients an API similar to ZooKeeper. *Read* operations are served with a direct access to the cloud storage. *Write* operations are sent to *follower* functions through a cloud queue. Eliminating the server from the data access path provides lower operating costs, but puts on the client the responsibility of ordering results with watch notifications. Thus, we replace the event coordination on ZooKeeper servers with a lightweight queue on the client: a read following a write cannot return before its predecessor.

Implementation. Each client runs three background threads to send requests, manage incoming responses, and order results. Epoch counters ensure the ordering of writes and notifications, and queues replace the ordered TCP communication of ZooKeeper.

3.6 Heartbeat

In addition to ordering guarantees, sessions play another significant role in ZooKeeper: their status defines the lifetime of ephemeral nodes, which are automatically deleted upon the closure of their owner's session. We replace the heartbeat messages with a *scheduled* ***heartbeat*** function to prune inactive sessions and notify clients that the system is online.

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}.

	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.

3.7 Summary

To finalize the serverless redesign of an IaaS service, we need to incorporate an elastic scaling model and ensure cloud portability.

Elastic resource allocation. To accommodate the temporal and spatial irregularity of workloads [36], 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 use the pay-as-you-go billing scheme of the storage and queue services, dependent only on the number of operations performed and not on the resources provisioned.

Cloud agnosticity. Vendor lock-in [61] is a serious limitation in serverless [16, 60], and dependency on queueing and storage services is of particular concern [13]. 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 the *cloud-agnostic* design of FaaSKeeper, 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 [59].

4 FROM FAASKEEPER DESIGN TO CLOUD

In the previous section, we mapped the ZooKeeper components to a cloud-agnostic design with separate services. Now, we map FaaSKeeper functions, queue, storage, and synchronization primitives to actual cloud services (Table 2), tailoring resource requirements to each component and enabling serverless scalability. Following the multi-step design guideline, we disaggregate computing (Section 4.1), incorporate different types of cloud storage (Section 4.2), and find the most optimal ways of exchanging data in the system (Section 4.3). Compute tasks can now be fully serverless, and the system requires no resource provisioning while providing compatible interface to ZooKeeper clients (Section 4.4). We scale FaaSKeeper up by adding more concurrent *follower* functions and placing data in different storage keys to benefit from sharding and cloud scalability.

Implementation. We select the AWS cloud and translate design concepts to cloud systems: system storage with DynamoDB tables, synchronization primitives to DynamoDB *update expressions* [12], user data storage to S3 buckets, and FIFO queues to the SQS. We use the AWS SQS with batched Lambda invocations [11] as it performs better than DynamoDB Streams (Section 5.2.2). 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 [40]. 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.

4.1 Disaggregating

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, more resources should be allocated for data endpoints rather than servers handling computing tasks. In FaaSKeeper, we removed the need for separate *reader* function - clients access cloud storage directly, saving time and money. Both key-value and object storage implement replication, with DynamoDB using three-way replication of each partition and S3 guaranteeing 11 9's of durability for each object. For that reason, we do not have to implement any additional replication.

User data locality. Cloud applications balance resource allocation across geographical regions to support changing workloads [21, 63]. 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 (Figure 3b). While ZooKeeper requires migrating a virtual machine across regions [63], FaaSKeeper can serve data from endpoints local to the user. Clients connect to the closest storage in their region, minimizing access latency.

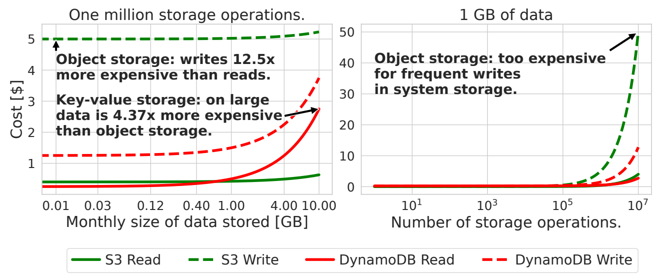
Decoupling Watch Delivery In FaaSKeeper, we moved the delivery of watch notifications into a separate *free function **watch***. Since hundreds of clients can register a single watch, using a serverless function allows us to adjust resource allocation to the workload. A standard writing pipeline includes only querying watch information in the system storage, adding insignificant cost and overhead.

4.2 Mapping Storage

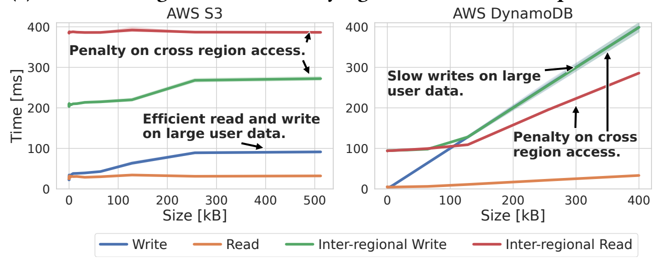
With storage and computing decoupled, we can map them to services that fit best their access patterns and computational requirements. Storage should distinguish between user data and the system data needed to control ZooKeeper: locality and cost requirements are different, and storage solutions have varied costs and latencies – especially when different sizes are considered (Figure 3).

Efficient reading of user data. ZooKeeper is optimized for high read performance. 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 (Figure 3a). 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 more expensive for large data in user nodes by even an order of magnitude.

Furthermore, we optimize ZooKeeper's `get_children` operation by storing the children list in the metadata of each node. This update does not add costs, as adding and removing nodes requires locking and updating the parent, and we avoid the expensive scan.



(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.

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

Hybrid storage While DynamoDB is cheaper for small nodes and faster (Section 5.3.1), the costs explode for large user data, restricting us to object storage. However, even though nodes store up to 1 MB of data, the dominant use case of ZooKeeper is to store small configuration objects. Thus, we optimize for the common case and place in DynamoDB all nodes up to 4 kB, and split node metadata and user data between DynamoDB and S3 for larger nodes. The client library begins by reading data from key-value storage, and only the infrequent large nodes incur the performance and cost penalty of a second storage request. This allows us to improve read latency by over 50% (Section 5.3.1) and decrease costs by 37.5%.

4.3 Communicating Functions

FaaSKeeper functions scale automatically with workload and emulate the TCP connection between the client and ZooKeeper servers.

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. However, cloud queues invoke functions in batches, preventing continuous streaming of new requests to the 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 [38, 62]. 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 (Section 2.1). Thus, the scalability of read and write operations is bounded by storage throughput.

4.4 Compatibility with ZooKeeper

Our implementation is standalone and does not reuse the server-centric ZooKeeper codebase since Java functions are by large cold startup overheads [42, 73]. We offer a compatible interface for existing applications by modeling our API after kazoo [9], a Python client for ZooKeeper. While FaaSKeeper aims to provide consistency model and interface compatible with ZooKeeper, we make minor adjustments due to the limitations of cloud services and the serverless model. While large ZooKeeper nodes are uncommon and impractical, we can support the 1MB node in cloud object storage. The size restrictions of 400 and 256 kB in DynamoDB, respectively, limit the maximum data sent by users. This 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, functions implement write permissions thanks to the protection boundary between caller and callee, and read permissions can be enforced with cloud storage ACLs.

4.5 Cloud Portability

To validate that FaaSKeeper design is cloud-agnostic and not locked to a single provider, we ported it to the Google Cloud Platform. We replace cloud services as specified in Table 2, and achieve the same semantics of a serverless service with pay-as-you-go-billing. The majority of the implementation effort was in adapting to new APIs and adding synchronization primitives as transactions [8], with changes in the system library (600 LoC), client code (200 LoC), and configuration (150 LoC). Google Cloud has size limits of 10 MB and 1 MB on queue and key-value storage operations, respectively, simplifying the implementation of large ZooKeeper nodes.

However, both platforms have different pricing models that affect the optimizations. While object storage costs the same, operations pricing on the key-value Datastore is independent of the item size. Compared to AWS DynamoDB, Datastore is 2.4x and 1.44x more expensive on read and write operations of up to 1 KB, respectively. While this simplifies the system design as we no longer need special treatment for large nodes, this is not the common case for ZooKeeper. On the other hand, the Pub/Sub queue charges clients based on the amount of data sent and received, but not less than 1 KB per message. At $40 per terabyte of data, the queue is 6.7x cheaper for small messages than AWS SQS.

5 EVALUATION

We begin with analyzing ZooKeeper utilization (Sec. 5.1) and benchmarking serverless components necessary to build a serverless service (Sec. 5.2). Then, we evaluate the performance-cost trade-offs of FaaSKeeper in relation to ZooKeeper (Sec. 5.3).

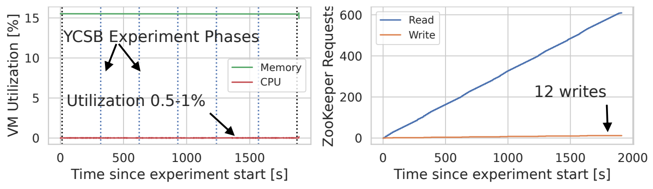


Figure 4: ZooKeeper utilization in HBase running YCSB.

Evaluation Platform The deployment in the AWS region `us-east-1` consists of four functions, SQS queues, and DynamoDB tables storing system state, user list, and watches. 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. We also deploy FaaSKeeper in the GCP region `us-central1`. Benchmarks use Python 3.8.10, and we run benchmarking clients from a `e2-medium` virtual machine with `Ubuntu 20.04` in the same cloud region. ZooKeeper 3.7.0 is deployed on three VMs running `Ubuntu 20.04`, using `t3.small` and `e2-small` machines on AWS and GCP, respectively.

5.1 ZooKeeper

To understand how ZooKeeper is used in practice, we profile its utilization in Apache HBase. We deploy HBase 2.5.6 with Hadoop 3.3.2 on four `t3.2xlarge` machines, with one holding HDFS NameNode and HBase HMaster, and others serving data, and ZooKeeper 3.9 on three `t3.medium` machines. From the benchmarking virtual machine, we execute the standard workloads from YCSB [26], each running for five minutes, and present results in Figure 4. The HBase service can handle thousands of requests while using ZooKeeper only to control the state of the cluster, With less than a thousand requests in over half an hour, replacing persistent ZooKeeper with a serverless system is a significant optimization opportunity.

Furthermore, we analyzed the size of ZooKeeper nodes after the experiment ended. HBase created 29 nodes, with a median and mean data size of 0 and 46 bytes, respectively. The largest node had 320 bytes of data and corresponded to each *RegionServer*.

5.2 Serverless Components

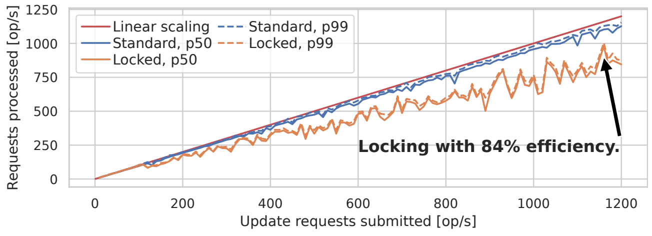
Now we evaluate the latency and throughput of components necessary to build a serverless service (Sec. 2.1).

5.2.1 Synchronization Primitives. The serverless synchronization primitives bring concurrent and safe updates to FaaSKeeper. Primitives are implemented in DynamoDB [12] and we evaluate the overheads and scalability of this datastore system.

Latency. We evaluate each operation by performing 1000 repetitions on warmed-up data and present results in Table 5a. 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

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).

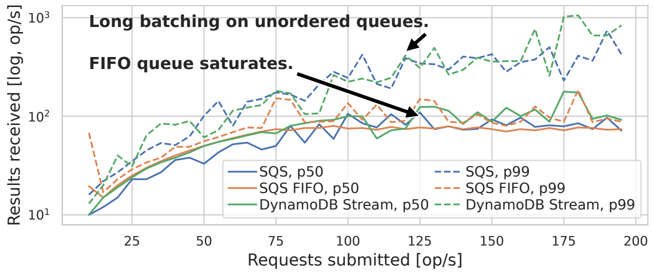


(b) Throughput of standard and locked DynamoDB updates.

Figure 5: Synchronization primitives on AWS DynamoDB.

	Direct		SQS		SQS FIFO		DynamoDB Stream	
	64B	64 kB	64B	64 kB	64B	64 kB	64B	64 kB
p50	39.0	48.69	39.83	51.68	24.22	34.47	242.65	237.75
p95	73.92	83.36	78.29	138.94	84.29	44.7	270.63	262.61
p99	124.01	117.25	125.24	184.91	162.42	55.26	417.21	464.52
Max	210.11	129.15	295.01	211.55	172.48	109.61	749.16	610.96

(a) End-to-end latency of FaaS invocation on AWS with a TCP reply.



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

	Direct		PubSub		PubSub FIFO	
	64B	64 kB	64B	64 kB	64B	64 kB
p50	83.29	85.29	38.04	29.23	201.22	206.62
p95	94.63	95.61	95.77	39.46	234.8	250.46
p99	112.74	97.49	114.43	46.32	581.19	263.0
Max	1115.14	112.73	643.96	57.66	588.95	280.84

(c) End-to-end latency of FaaS invocation on GCP with a TCP reply.

Figure 6: Function invocations with serverless queues.

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 add new watches in storage with a single operation.
Throughput. Timed locks allow FaaSKeeper to conduct independent updates concurrently. We evaluate a pair of regular reads and writes, compare ing them against our locks with 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 (Figure 5b). 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 operations on DynamoDB is limited by Lambda's parallelism and not by storage scalability [74].

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

5.2.2 *Serverless Queues.* Queues improve the writing process by batching requests and are necessary to provide ordering (Section 3.2) 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 the minimal invocation overhead and allows to achieve sufficient throughput. For SQS [6], 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 overhead of small batch sizes. For DynamoDB streams, we configure database sharding to guarantee that all items in a table are processed in order [3]. We restrict the function's concurrency to permit only one instance at a time.
Latency. We measure the end-to-end latency by triggering an empty a function that returns a dummy result to the user with a TCP connection. We consider the best-case scenario of warm invocations with a cached 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 potential of user-side request batching without cloud proxies, and present AWS and GCP results in Tables 6a and 6c. Surprisingly, the FIFO queue achieves the lowest latency and is faster than a direct Lambda invocation. Thus, offloading requests using SQS-based invocation comes with approximately 20ms of overhead. However, the ordered PubSub subscription is slower than the direct cloud function invocation and unordered subscription, adding over 170 ms 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 6b). 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.

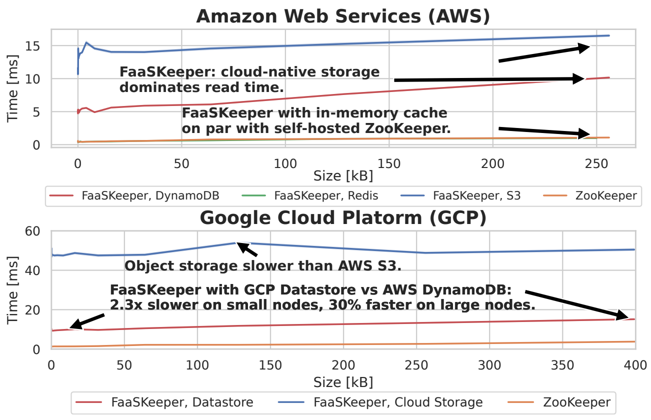


Figure 7: Read operations in FaaSKeeper and ZooKeeper.

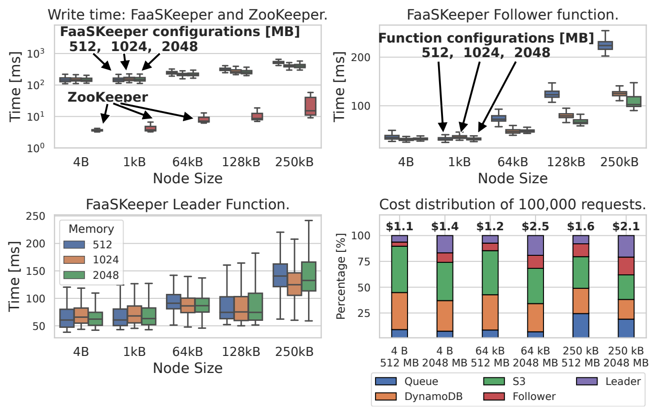


Figure 8: Write operations in FaaSKeeper and ZooKeeper.

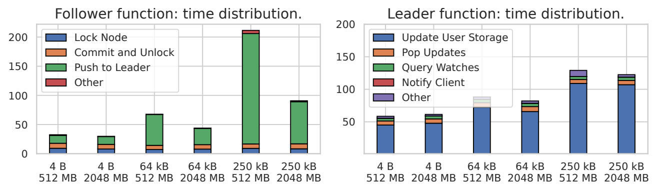


Figure 9: Time distribution of FaaSKeeper functions.

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 FaaSKeeper vs ZooKeeper

We evaluate FaaSKeeper and compare against ZooKeeper in four domains: read performance, write latency, service monitoring, and cost trade-offs.

5.3.1 Read Operations. ZooKeeper is designed for efficient read operations, and our FaaSKeeper must offer the same. We evaluate the `get_data` operation that retrieves a ZooKeeper node, timing the retrieval on the user side. On AWS, we evaluate S3, DynamoDB and Redis (`t3.small` VM) as the user data store. On GCP, we evaluate Cloud Storage and Datastore. We repeat the measurements 100 times for each node size and present results in Figure 7. We compare FaaSKeeper against ZooKeeper, placing the benchmarking client in the same cloud region zone as one of ZooKeeper's nodes.

Hybrid storage distributes nodes between both storage options (Section 4.2), allowing us to benefit from the low latency of DynamoDB on small nodes while placing large user data to S3. This avoids the cost explosion as reading 128 kB data from DynamoDB is 20x more expensive than S3. 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 in the client library 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.

5.3.2 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 8), pushing to the size limit of 250 kB. First, we notice that ZooKeeper achieves lower write latency due to the direct connection with a client and operating on a local state in memory. The latency in FaaSKeeper is bounded by the functions and the overheads of queue-based invocations. Then, we study the execution times of follower and leader functions. The leader function contributes more to the total write latency, especially on small input sizes, and exhibits a strong variance. Finally, we look at the writing cost and find that storage operations are responsible for 40-80% of it, with functions contribution 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 larger allocations to increase I/O performance [28], leading to increased cost and resource underutilization.

Overhead To locate the bottleneck of writing in FaaSKeeper, we inspect where functions spend time. Figure 9 shows the impact of synchronization operations is limited, and the runtime of leader and follower 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.

Variability To understand the sources of performance variability observed in Figure 8, we examine tail latencies of the important operations (Table 3). We observe significant performance degradation at the tail percentiles when pushing to queue in *follower* and updating S3 nodes in *leader*. This result aligns with the previous subsection: distributed applications in serverless are particularly affected by inefficient queues and remote storage.

Hybrid Storage We evaluate the impact of hybrid storage on the node size range typical for ZooKeeper applications (Figure 10). By

	Follower	Size	Min	p50	p90	p95	p99
Follower	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
Leader	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.

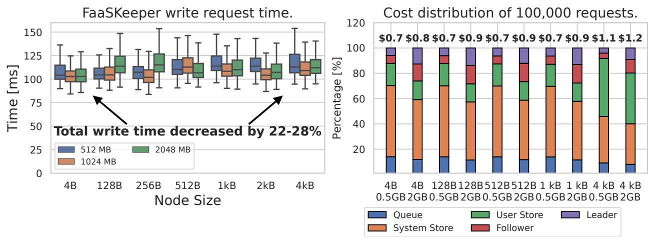


Figure 10: FaaSKeeper writes with hybrid storage.

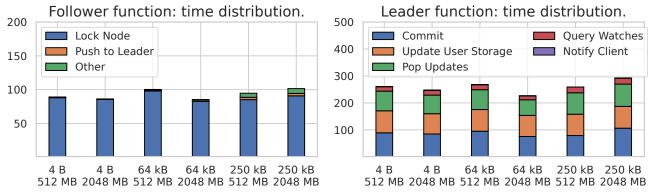


Figure 11: FaaSKeeper writes on Google Cloud.

replacing S3 with DynamoDB for user storage, we improve not only the cost and performance of reading, but also decrease the write time while keeping costs for infrequent large nodes under control.
Google Cloud Finally, we evaluate the write performance on Google Cloud (Figure 11). Compared to AWS, FaaSKeeper achieves worse performance due to significantly more expensive synchronization with transactions on key-value storage. However, the hybrid storage optimization does not apply here since the cost of reading from the NoSQL storage is larger than from object storage.
Resource Configuration Finally, we explore new configuration options available in serverless. In Google Cloud, we test the ability to change CPU allocation independently from the memory allocation. When comparing functions with 512MB memory and 0.33 or 1 virtual CPU, we notice performance change of 2-10%, often favoring the smaller allocation. However, a smaller CPU allocation translates to a 54-62% cost decrease. Applications like FaaSKeeper are I/O-bound and benefit from flexible allocation of CPU resources.

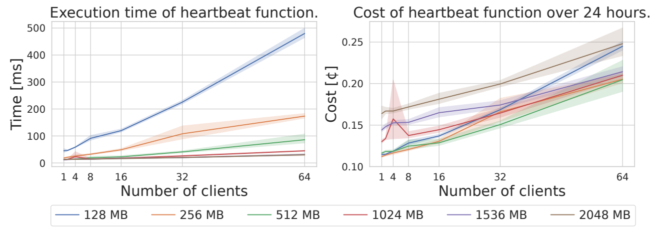


Figure 12: Heartbeat function performance and cost.

Then, we compare the x86 and ARM instances of AWS Lambda. There, ARM functions perform better on follower functions but experience significant slowdowns of up to 94% on the leader function. Depending on the configuration, ARM functions can decrease costs of follower functions by up to 32%.

Write operations are limited by data transmission to queues and object storage, motivating the need for more efficient queues.

5.3.3 Service Monitoring. 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 12. Execution time decreases with the allocation, corresponding with previous findings on serverless I/O [28, 73].

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.

5.3.4 Cost. The most important evaluation compares the price of running an elastic FaaSKeeper instance to Zookeeper with standard and hybrid storage on AWS, with x86 functions. We consider a scenario of 512 MB, with reads and writes to one node of 1 kB, and the optimistic case that we experience no failures and, therefore, no retries.
FaaSKeeper 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.

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

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, function executions, synchronization

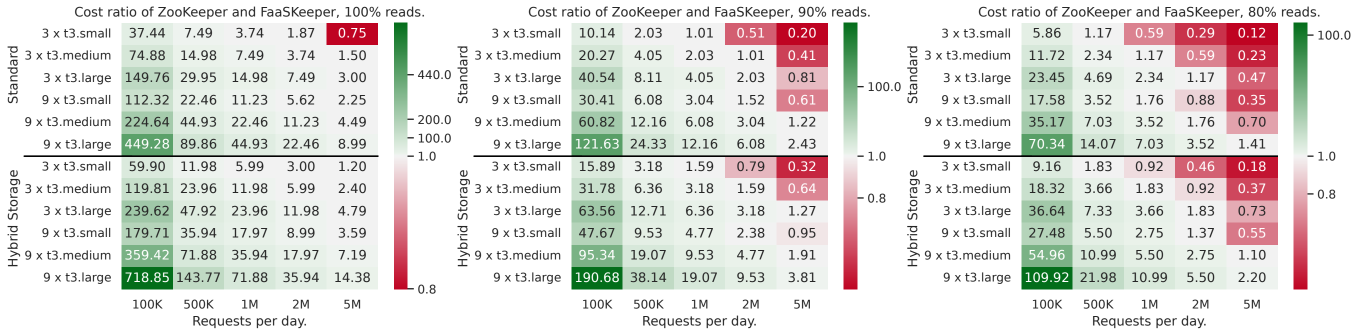


Figure 13: Cost ratio of ZooKeeper and FaaSKeeper, running a workload mix of 1 kB reads and writes with `set_data`.

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 follower and leader function.	-

Table 4: Parameters of FaaSKeeper cost model.

in the `follower` and `leader`, and writing data to the user store.

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

A workload of 100,000 write operations costs \$1.12. With hybrid storage, the cost of user storage write $W_{S3}(s)$ becomes $W_{DD}(s)$ There, a workload of 100,000 write operations costs \$0.72.
Storage. The databases and queues do not generate any inactivity charges except for retaining data. 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. The hybrid storage incurs higher costs, as retaining data in DynamoDB is 3.125x more expensive than block storage. However, the size of ZooKeeper data is not high as nodes are usually small, and data access costs dominate the long-term storage.
ZooKeeper The cost 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 (3 VMs) and \$14.4 (9 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 13. 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 equal the smallest possible ZooKeeper deployment. With hybrid storage, this number grows to 5.99 million daily read requests. 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.

6 BUILDING SERVERLESS SERVICES

In the following section, we address the primary challenges encountered during the development of FaaSKeeper and highlight the current limitations of serverless technology. We compile a set of requirements cloud providers could easily support and pave the way for future improvements. They would make complex serverless systems more efficient and performant, simplifying their implementation and increasing adoption. In particular, they would allow FaaSKeeper to match ZooKeeper's performance when using off-the-shelf cloud services. We finally discuss how well these requirements are supported in research and emerging cloud architectures.

6.1 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, the rationale behind these requirements is not limited to our use case, and will improve other applications, such as microservices [46] and serverless ML [24, 45].
Requirement #1: Fast invocations. Invocation overheads dominate the execution time of short-running functions [28] and prohibit FaaS processing with performance comparable to non-serverless applications that can use a direct RPC call over a TCP connection. 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 Figure 8.
Requirement #2: Exception handling. The user cannot control asynchronous function invocations (Section 3.2). We envision this should be solved via user-defined *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 atomics (Section 2.1). In practice, sub-millisecond latency is needed, like the one offered by in-memory storage.

Requirement #4: FIFO Queues. Serverless functions require queues to support the ordering and reliability of invocations (Section 3.2). 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. Furthermore, they can be significantly slower than regular invocations (Section 5.2.2).
Requirement #5: Statefulness. While stateless functions are sufficient for many use cases, *stateful* functions are necessary to efficiently process requests that depend on each other (Section 3.2). FaaS should support a reliable and low-latency function state.
Requirement #6: Partial updates. To increase the efficiency of write operations, object storage could support partial updates where data is written at a user-defined offset to the specified object, avoiding the need for the read-update-write process (Section 3.2).
Requirement #7: Outbound channels. While the trigger system provides *inbound* communication, functions lack an ordered, push-based, and fast *outbound* communication channel with acknowledgment of delivery. Cloud queues are an order of magnitude slower than a TCP connection and do not validate that the recipient read the message. Such a channel would significantly simplify the design of serverless services such as FaaSKeeper (Section 3.2).
Requirement #8: Fast serverless storage. ZooKeeper is a data-intensive system focused on fast read operations. In FaaSKeeper, in-memory storage could deliver competitive performance, but it is not available as serverless and cloud-native service (Section 5.3.1).
Requirement #9: Decoupling I/O and compute. FaaSKeeper functions spend most of the time waiting on requests to cloud services – increasing CPU allocation alone has little effect on performance (Section 5.3.2). Furthermore, queue batching (R4) prevents effective handling of many requests within a single function. Instead, functions should be swapped out during idle periods to free up resources. While this is a fundamental change, improved I/O management would decrease user costs and allow cloud providers to increase utilization with a larger degree of oversubscription.

6.2 Discussion

Can serverless systems support our requirements? We specify nine requirements to define features missing in cloud-native FaaS systems that are necessary for distributed, stateful, and scalable applications. The requirements align with the major serverless challenges [47, 56] and are supported in research FaaS platforms. Emerging systems provide microsecond-scale invocation latency [29, 44] and I/O separation from functions [54]. New storage systems satisfy the latency, consistency, and flexibility requirements of functions [51, 52, 67, 76], including serverless in-memory caches [72, 75]. Furthermore, stateful serverless is becoming the new norm in clouds [17, 27, 43, 64, 74]. Finally, we note that research systems can support many of our requirements already: Cloudburst (R1, R5) [67], PraaS (R1, R5, R7, R9) [27], Boki (R3-R5) [43].

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 can match ZooKeeper's read performance by incorporating an in-memory database, but these are unavailable as a cloud-native serverless service and require third-party solutions [10, 72]. The increased processing time of write requests is caused primarily by performance variations of cloud queues and object storage.

7 RELATED WORK

Serverless for Storage Wang et al. [72] use functions for elastic in-memory cache. Boki provides stateful serverless on shared logs [43]. DynamoDB is used in transactional workflows with locks in Beldi [74] and in a fault-tolerance shim AFT [66]. 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.

λFS implements a serverless metadata layer of a distributed file system [25]. Similarly to FaaSKeeper, functions operate on top of strongly consistent data storage. However, the implementation of read operations is different in both systems. In λFS, metadata reading is handled by functions that use caching to avoid reaching to the data store. In FaaSKeeper, functions are removed entirely from the reading path. Finally, λFS also identified some of the requirements for efficient serverless applications (Sec. 6), such as fast invocations that bypass the slow interface of HTTP requests.
Elastic Storage Cloud-native storage is known for elastic implementations that scales with changes in workload [53]. Examples include reconfiguration controllers [31, 55], workload predictors [57], and latency monitoring [18]. PolarDB is an example of a disaggregated database that offers a serverless billing model [23]. However, ZooKeeper requires autoscaling procedures that integrate the state ordering guarantees. FaaSKeeper achieves that by using the autoprovisioning of serverless functions and databases.
ZooKeeper Other works explored different approaches to ZooKeeper's performance. Stewart et al. [69] replicated data on multiple servers to provide predictable access latencies. Shen et al. [63] proposed live migration for geographical reconfiguration. ZooKeeper has been improved with hardware implementations using FPGAs [41] and the PsPIN [34] network accelerator [68].

8 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 cloud-native and 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 719x. We discuss the lessons learned in creating FaaSKeeper, and identify nine requirements that clouds should fulfill to ensure functionality and performance.

ACKNOWLEDGMENTS

This project received funding from EuroHPC-JU under grant agreements DEEP-SEA, No 955606 and RED-SEA, No 955776. We thank Amazon Web Services for supporting this research with credits through the AWS Cloud Credit for Research, and Google Cloud Platform through the Google Cloud Research Credits program with the award GCP19980904.

REFERENCES

[1] 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: 2023-11-30.

[2] 2020. Azure Cosmos DB serverless now in preview. https://devblogs.microsoft.com/cosmosdb/serverless-preview/. Accessed: 2023-11-30.

[3] 2020. Using AWS Lambda with Amazon DynamoDB. https://docs.amazonaws.cn/en_us/lambda/latest/dg/with-ddb.html. Accessed: 2023-11-30.

[4] 2020. ZooKeeper Dynamic Reconfiguration. https://zookeeper.apache.org/doc/current/zookeeperReconfig.html. Accessed: 2023-11-30.

[5] 2020. ZooKeeper Programmer's Guide. https://zookeeper.apache.org/doc/current/zookeeperProgrammers.html. Accessed: 2023-11-30.

[6] 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: 2023-11-30.

[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: 2023-11-30.

[8] 2021. Google Cloud Datastore: Transactions. https://cloud.google.com/datastore/docs/concepts/transactions. Accessed: 2023-11-30.

[9] 2021. Kazoo: high-level Python library for ZooKeeper. https://github.com/python-zk/kazoo. Accessed: 2023-11-30.

[10] 2021. Upstash: Serverless Database for Redis. https://upstash.com/redis. Accessed: 2023-11-30.

[11] 2021. Using AWS Lambda with Amazon SQS. https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html. Accessed: 2023-11-30.

[12] 2021. Using Expressions in DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Expressions.html. Accessed: 2023-11-30.

[13] 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

[14] 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

[15] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi. 2003. Towards an object store. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings.* 165–176. https://doi.org/10.1109/MASS.2003.1194853

[16] 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

[17] 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

[18] 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

[19] 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

[20] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. 2008. Building a Database on S3. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 251–264. https://doi.org/10.1145/1376616.1376645

[21] 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.

[22] 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

[23] 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

[24] 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

[25] Benjamin Carver, Runzhou Han, Jingyuan Zhang, Mai Zheng, and Yue Cheng. 2024. λFS: A Scalable and Elastic Distributed File System Metadata Service using Serverless Functions. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (, Vancouver, BC, Canada,) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 394–411. https://doi.org/10.1145/3623278.3624765

[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[27] Marcin Copik, Alexandru Calotoiu, Rodrigo Bruno, Gyorgy Rethy, Roman Böhringer, and Torsten Hoefler. 2022. Process-as-a-Service: Elastic and Stateful Serverless with Cloud Processes. (2022). https://spcl.inf.ethz.ch/Publications/index.php?pub=458 Accessed: 2023-11-30.

[28] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: a serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 64–78. https://doi.org/10.1145/3464298.3476133

[29] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. 2023. rFaaS: Enabling High Performance Serverless with RDMA and Leases. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 897–907. https://doi.org/10.1109/IPDPS54959.2023.00094

[30] 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

[31] 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

[32] 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

[33] 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

[34] 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).

[35] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. 2005. Object storage: the future building block for storage systems. In *2005 IEEE International Symposium on Mass Storage Systems and Technology.* 119–123. https://doi.org/10.1109/LGDI.2005.1612479

[36] 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

[37] 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.

[38] 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

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

[40] 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.

[41] 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

[42] 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

[43] 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

[44] 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

[45] 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

[46] 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

[47] 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

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

[49] 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

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

[51] 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

[52] 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.

[53] 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

[54] Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. Function as a Function. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (, Santa Cruz, CA, USA,) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/3620678.3624648

[55] 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

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

[57] 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

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

[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] 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

[61] 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

[62] 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

[63] 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

[64] 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

[65] 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

[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] 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

[68] Elias Stalder. 2020. *Zoo-Spinner: A Network-Accelerated Consensus Protocol*. Master's thesis. ETH Zurich. https://doi.org/10.3929/ethz-b-000455366

[69] 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

[70] 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.

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

[72] 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

[73] 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.

[74] 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

[75] Jingyuan Zhang, Ao Wang, Xiaolong Ma, Benjamin Carver, Nicholas John Newman, Ali Anwar, Lukas Rupprecht, Vasily Tarasov, Dimitrios Skourtis, Feng Yan,

and Yue Cheng. 2023. InfiniStore: Elastic Serverless Cloud Storage. *Proc. VLDB Endow.* 16, 7 (mar 2023), 1629–1642. https://doi.org/10.14778/3587136.3587139

[76] 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