Boki: Stateful Serverless Computing with Shared Logs

Zhipeng Jia
The University of Texas at Austin

Emmett Witchel
The University of Texas at Austin and Katana Graph

Abstract

Boki is a new serverless runtime that exports a shared log API to serverless functions. Boki shared logs enable stateful serverless applications to manage their state with durability, consistency, and fault tolerance. Boki shared logs achieve high throughput and low latency. The key enabler is the *metalog*, a novel mechanism that allows Boki to address ordering, consistency and fault tolerance independently. The metalog orders shared log records with high throughput and it provides read consistency while allowing service providers to optimize the write and read path of the shared log in different ways. To demonstrate the value of shared logs for stateful serverless applications, we build Boki support libraries that implement fault-tolerant workflows, durable object storage, and message queues. Our evaluation shows that shared logs can speed up important serverless workloads by up to 4.7×.

CCS Concepts: • **Information systems** → **Distributed storage**; • **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; **Cloud computing**.

Keywords: Serverless computing, function-as-a-service, shared log, consistency

ACM Reference Format:
Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3477132.3483541

1 Introduction

Serverless computing has become increasingly popular for building scalable cloud applications. Its function-as-a-service (FaaS) paradigm empowers diverse applications including video processing [21, 32], data analytics [39, 47], machine learning [27, 51], distributed compilation [31], transactional workflows [56], and interactive microservices [38].

One key challenge in the current serverless paradigm is the mismatch between the stateless nature of serverless functions and the stateful applications built with them [36, 48, 52, 59]. Serverless applications are often composed of multiple functions, where application state is shared. However, managing shared state using current options, e.g., cloud databases or object stores, struggles to achieve strong consistency and fault tolerance while maintaining high performance and scalability [50, 56].

The shared log [23, 30, 55] is a popular approach for building storage systems that can simultaneously achieve scalability, strong consistency, and fault tolerance [7, 22, 24, 26, 35, 41, 54, 55]. A shared log offers a simple abstraction: a totally ordered log that can be accessed and appended concurrently. While simple, a shared log can efficiently support state machine replication [49], the well-understood approach for building fault-tolerant stateful services [24, 55]. The shared log API also frees distributed applications from the burden of managing the details of fault-tolerant consensus, because the consensus protocol is hidden behind the API [22]. Providing shared logs to serverless functions can address the dual challenges of consistency and fault tolerance (§ 2.1).

We present Boki (meaning bookkeeping in Japanese), a FaaS runtime that exports the shared log API to functions for storing shared state. Boki realizes the shared log API with a LogBook abstraction, where each function invocation is associated with a LogBook (§ 3). For a Boki application, its functions share a LogBook, allowing them to share and coordinate updates to state. In Boki, LogBooks enable stateful serverless applications to manage their state with durability, consistency, and fault tolerance.

The shared log API is simple to use and applicable to diverse applications [22, 24, 25, 55], so the challenge of Boki is to achieve high performance and strong consistency while conforming to the serverless environment (§ 2.2). Data locality is one challenge for serverless storage, because disaggregated storage is strongly preferred in the serverless environment [36, 48, 52]. Boki separates the read and write path, where read locality is optimized with a cache on function nodes and writes are optimized with scale-out bandwidth. Boki will scatter writes over variable numbers of shards while providing consistent reads and fault tolerance. In Boki, high performance, read consistency and fault tolerance are achieved by a single log-based mechanism, the *metalog*.

The metalog defines a total order of Boki's internal state that applications can use to enforce consistency when they need it. For example, monotonic reads are enforced by tracking metalog positions. The metalog contains metadata that totally orders a log's data records. Because Boki uses a compact format for the metalog, durability and consensus are

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483541

vital, but high data throughput is not. Therefore Boki stores and updates metalogs using a simple primary-driven design.

Boki handles machine failures by reconfiguration, similar to previous shared log systems [22, 30, 55]. Because the metalog controls Boki's internal state transitions, sealing the metalog (making it no longer writable) pauses state transitions. Therefore, Boki implements reconfiguration by sealing the metalog, changing the system configuration, and starting a new metalog.

Boki's metalog allows easy adoption of state-of-the-art techniques from previous shared log designs because it make log ordering, consistency, and fault tolerance into independent modules (§ 4.1). Boki adapts ordering from Scalog [30] and fault tolerance from Delos's [22] sealing protocol. Another benefit of the metalog is it decouples read consistency from data placement, enabling indices and caches for log records to be co-located with functions. Without interfering with read consistency, cloud providers can build simple caches which increase data locality when scheduling functions on nodes where their data is likely to be cached.

We implement Boki's shared log designs on top of Nightcore [38], a FaaS runtime optimized for microservices. Nightcore has no specialized mechanism for state management, Boki provides it; while Nightcore's design for I/O efficiency benefits Boki. Boki achieves append throughput of 1.2M Ops/s within a single LogBook, while maintaining a p99 latency of 6.4ms. With LogBook engines co-located with functions, Boki achieves a read latency of 121μs for best-case LogBook reads.

To make writing Boki applications easier, we build support libraries on top of the LogBook API aimed at three different serverless use cases: fault-tolerant workflows, durable object storage, and serverless message queues. Boki support libraries leverage techniques from Beldi [56], Tango [24], and vCorfu [55], while adapting them for the LogBook API. Boki and its support libraries are open source on GitHub `ut-osa/boki`.

This paper makes the following contributions.

- Boki is a FaaS runtime that exports a LogBook API for stateful serverless applications to manage their state with durability, consistency, and fault tolerance.
- Boki proposes a unified mechanism, the metalog, to address log ordering, read consistency, and fault tolerance. The metalog decouples the read and write path of LogBooks, letting Boki achieve high throughput and low latency.
- We build Boki support libraries that use the LogBook API to demonstrate the value of shared logs for stateful serverless applications. The libraries implement fault-tolerant workflows (BokiFlow), durable object storage (BokiStore), and serverless message queues (BokiQueue).
- Our evaluation shows: BokiFlow executes workflows 4.3–4.7× faster than Beldi [56]; BokiStore achieves 1.18–1.25× higher throughput than MongoDB, while executing transactions 1.5–2.3× faster; BokiQueue achieves 2.14× higher throughput and up to 15× lower latency than Amazon SQS [2], while achieving 1.23× higher throughput and up to 2.0× lower latency than Apache Pulsar [3].

2 Background and Motivation

Serverless functions, or function as a service (FaaS) [4, 6], allow developers to upload simple functions to the cloud provider which are invoked on demand. The cloud provider manages the execution environment of serverless functions.

State management remains a major challenge in the current FaaS paradigm [36, 48, 52, 59]. Because of the stateless nature of serverless functions, current serverless applications rely on cloud storage services (e.g., Amazon S3 and DynamoDB) to manage their state. However, current cloud storage cannot simultaneously provide low latency, low cost, and high throughput [42, 47]. Relying on cloud storage also complicates data consistency in stateful workflows [56], because functions in a workflow could fail in the middle which leaves inconsistent workflow state stored in the database.

2.1 Shared Log Approach for Stateful Serverless

In the current FaaS paradigm, stateful applications struggle to achieve fault tolerance and strong consistency of their critical state. For example, consider a travel reservation app built with serverless functions. This app has a function for booking hotels and another function for booking flights. When processing a travel reservation request, both functions are invoked, but both functions can fail during execution, leaving inconsistent state. Using current approaches for state management such as cloud object stores or even cloud databases, it is difficult to ensure the consistency of the reservation state given the failure model [56].

The success of log-based approaches for data consistency and fault tolerance motivates the usage of shared logs for stateful FaaS. For example, Olive [50] proposes a client library interacting with cloud storage, where a write-ahead redo log is used to achieve exactly-once semantics in face of failures. Beldi [56] extends Olive's log-based techniques for transactional serverless workflows. State machine replication (SMR) [49] is another general approach for fault tolerance, where application state is replicated across servers by a command log. The command log is traditionally backed by consensus algorithms [45, 46, 53]. But recent studies demonstrate a shared log can provide efficient abstraction to support SMR-based data structures [24, 55] and protocols [22, 25]. Boki provides shared logs to serverless functions, so that Boki's applications can leverage well-understood log-based mechanisms to efficiently achieve data consistency and fault tolerance.

By examining demands in serverless computing, we identify three important cases where shared logs provide a solution. Boki provides support libraries for these use cases (§ 5).

Fault-tolerant workflows. Workflows orchestrating stateful functions create new challenges for fault tolerance and

transactional state updates. Beldi [56] addresses these challenges via logging workflow steps. Beldi builds an atomic logging layer on top of DynamoDB. We adapt Beldi's techniques to the LogBook API without building an extra logging layer.

Durable object storage. Previous studies like Tango [24] and vCorfu [55] demonstrate that shared logs can support high-level data structures (i.e., objects), that are consistent, durable, and scalable. Motivated by Cloudflare's Durable Objects [17], we build a library for stateful functions to create durable JSON objects. Our object library is more powerful than Cloudflare's because it supports transactions across objects, using techniques from Tango [24].

Serverless message queues. One constraint in the current FaaS paradigm is that functions cannot directly communicate with each other via traditional approaches [31], e.g., network sockets. Shared logs can naturally be used to build message queues [30] that offer indirect communication and coordination among functions. We build a queue library that provides shared queues among serverless functions.

2.2 Technical Challenges for Serverless Shared Logs

While prior shared log designs [22, 23, 30, 55] provide inspiration, the serverless environment creates new challenges.

Elasticity and data locality. Serverless computing strongly benefits from disaggregation [20, 34], which offers elasticity. However, current serverless platforms choose physical disaggregation, which reduces data locality [36, 52]. Boki achieves both elasticity and data locality, by decoupling the read and the write paths for log data and co-locating read components with functions.

Resource efficiency. Boki aims to support a high density of LogBooks efficiently, so it multiplexes many LogBooks on a single physical log. Multiplexing LogBooks can address performance problems that arise from a skewed distribution of LogBook sizes. But this approach creates a challenge for LogBook reads: how to locate the records of a LogBook. Boki proposes a log index to address this issue, with the metalog providing the mechanism for read consistency (§ 4.4).

The ephemeral nature of FaaS. Shared logs are used for building high-level data structures via state machine replication (SMR) [24, 55]. To allow fast reads, clients keep in-memory copies of the state machines, e.g., Tango [24] has local views for its SMR-based objects. However, serverless functions are ephemeral – their in-memory state is not guaranteed to be preserved between invocations. This limitation forces functions to replay the full log when accessing a SMR-based object. Boki introduces auxiliary data (§ 3) to enable optimizations like local views in Tango (§ 5.4). Auxiliary data are designed as cache storage on a per-log-record basis, while their relaxed durability and consistency guarantees allow a simple and efficient mechanism to manage their storage (§ 4.4).

```
struct LogRecord {
   uint64_t seqnum;     string data;
   vector<tag_t> tags;  string auxdata;
};
// Append a new log record.
status_t logAppend(vector<tag_t> tags, string data,
                   uint64_t* seqnum);
// Read the next/previous record whose
// seqnum >=`min_seqnum`, or <=`max_seqnum`.
status_t logReadNext(uint64_t min_seqnum, tag_t tag,
                     LogRecord* record);
status_t logReadPrev(uint64_t max_seqnum, tag_t tag,
                     LogRecord* record);
// Alias of logReadPrev(kMaxSeqNum, tag, record).
status_t logCheckTail(tag_t tag, LogRecord* record);
// Trim log records until `seqnum`.
status_t logTrim(uint64_t seqnum, tag_t tag);
// Set auxiliary data for the record of `seqnum`.
status_t logSetAuxData(uint64_t seqnum, string auxdata);
```

Figure 1. Boki's LogBook API (§ 3).

3 Boki's LogBook API

Boki provides a LogBook abstraction for serverless functions to access shared logs. Boki maintains many independent LogBooks used by different serverless applications. In Boki, each function invocation is associated with *one* LogBook, whose *book_id* is specified when invoking the function. A LogBook can be shared with multiple function invocations, so that applications can share state among their function instances.

Like previous shared log systems [22, 23, 30, 55], Boki exposes *append*, *read*, and *trim* APIs for writing, reading, and deleting log records. Figure 1 lists Boki's LogBook API.

Read consistency. LogBook guarantees *monotonic reads* and *read-your-writes* when reading records. These guarantees imply a function has a monotonically increasing view of the log tail. Moreover, a child function inherits its parent function's view of the log tail, if two functions share the same LogBook. This property is important for serverless applications that compose multiple functions (§4.4).

Sequence numbers (seqnum). The `logAppend` API returns a unique seqnum for the newly appended log record. The seqnums determine the relative order of records within a LogBook. They are monotonically increasing but *not* guaranteed to be consecutive. Boki's `logReadNext` and `logReadPrev` APIs enable bidirectional log traversals, by providing lower and upper bounds for seqnums (§4.2).

Log tags. Every log record has a set of tags, that is specified in `logAppend`. Log tags enable selective reads and trims, where only records with the given tag are considered (see the `tag` parameter in `logReadNext`, `logReadPrev`, and `logTrim` APIs). Records with same tags form abstract streams within a single LogBook. Having sub-streams in a shared log for selective reads is important for reducing log replay overheads, that is used in Tango [24] and vCorfu [55] (§4.4).

Auxiliary data. LogBook's auxiliary data is designed as per-log-record cache storage, which is set by the `logSetAuxData` API. Log reads may return auxiliary data along with normal data if found. Auxiliary data can cache object views in a shared-log-based object storage. These object views can significantly reduce log replay overheads (§ 5.4).

As auxiliary data is designed to be used only as a cache, Boki does not guarantee its durability, but provides best effort support. Moreover, Boki does not maintain the consistency of auxiliary data, i.e., Boki trusts applications to provide consistent auxiliary data for the same log record. Relaxing durability and consistency allows Boki to have a simple yet efficient backend for storing auxiliary data (§ 4.4).

4 Boki Design

Boki's design combines a FaaS system with shared log storage. Boki internally stores multiple independent, totally ordered logs. User-facing LogBooks are multiplexed onto internal physical logs for better resource efficiency (§ 2.2). A Boki physical log has an associated *metalog*, playing the central role in ordering, consistency, and fault tolerance.

4.1 Metalog is "the Answer to Everything" in Boki

Every shared log system must answer three questions because they store log records across a group of machines. The first is how to determine the global total order of log records. The second is how to ensure read consistency as the data are physically distributed. The third is how to tolerate machine failures. Table 1 shows different mechanisms used by previous shared log systems to address these three issues, whereas in Boki, the *metalog* provides the single solution to all of them.

In Boki, every physical log has a single associated *metalog*, to record its internal state transitions. Boki sequencers append to the metalog, while all other components subscribe to it. In particular, appending, reading, and sealing the metalog provide mechanisms for log ordering, read consistency, and fault tolerance:

- *Log ordering.* The primary sequencer appends metalog entries to decide the total order for new records, using Scalog [30]'s high-throughput ordering protocol. (§ 4.3)
- *Read consistency.* Different LogBook engines update their log indices independently, however, read consistency is enforced by comparing metalog positions. (§ 4.4)
- *Fault tolerance.* Boki is reconfigured by sealing metalogs, because a sealed metalog pauses state transitions for the associated log. When all current metalogs are sealed, a new configuration can be safely installed. (§ 4.5)

The metalog is backed by a primary-driven protocol. Every Boki metalog is stored by n_{meta} sequencers (which is 3 in the prototype). One of the n_{meta} sequencers is configured as primary, and only the primary sequencer can append the metalog. To append a new metalog entry, the primary sequencer sends the entry to all secondary sequencers for replication.

Table 1. Comparison between vCorfu [55], Scalog [30], and Boki. Boki's metalog provides a unified approach for log ordering, read consistency, and fault tolerance (§ 4.1).

	Ordering Log Records	**Read Consistency**	**Failure Handling**
vCorfu	A dedicated sequencer	Stream replicas	Hole-filling protocol
Scalog	Paxos and aggregators	Sharding policy	Paxos
Boki	Appending *metalog* entries	Tracking *metalog* positions	Sealing the *metalog*

Once acknowledged by a quorum, the new metalog entry is successfully appended. The primary sequencer always waits for the previous entry to be acknowledged by a quorum before issuing the next one. Sequencers propagate appended metalog entries to other Boki components that subscribe to the metalog.

4.2 Architecture

Figure 2 depicts Boki's architecture, which is based on Nightcore [38], a state-of-the-art FaaS system for microservices. In Nightcore's design, there is a gateway for receiving function requests and multiple function nodes for running serverless functions. On each function node, an engine process communicates with the Nightcore runtime within function containers via low-latency message channels.

Boki extends Nightcore's architecture by adding components for storing, ordering, and reading logs. Boki also has a control plane for storing configuration metadata and handling component failures.

Storage nodes. Boki stores log records on dedicated storage nodes. Boki's physical logs are sharded, and each log shard is stored on n_{data} storage nodes (n_{data} equals 3 in the prototype). Individual storage nodes contain different shards from the same log, and/or shards from different logs, depending on how Boki is configured.

Sequencer nodes. Sequencer nodes run Boki sequencers that store and update metalogs using a primary-driven protocol (see § 4.1). Sequencers append new metalog entries to order physical log records as detailed in § 4.3. Similar to storage nodes, individual sequencer nodes can be configured to back different metalogs.

LogBook engines. In Nightcore, the engine processes running on function nodes are responsible for dispatching function requests. Boki extends Nightcore's engine by adding a new component serving LogBook calls. We refer the new part as LogBook engine, to distinguish it from the part serving function requests.

LogBook API requests are forwarded to LogBook engines by Boki's runtime, which is linked with user supplied function

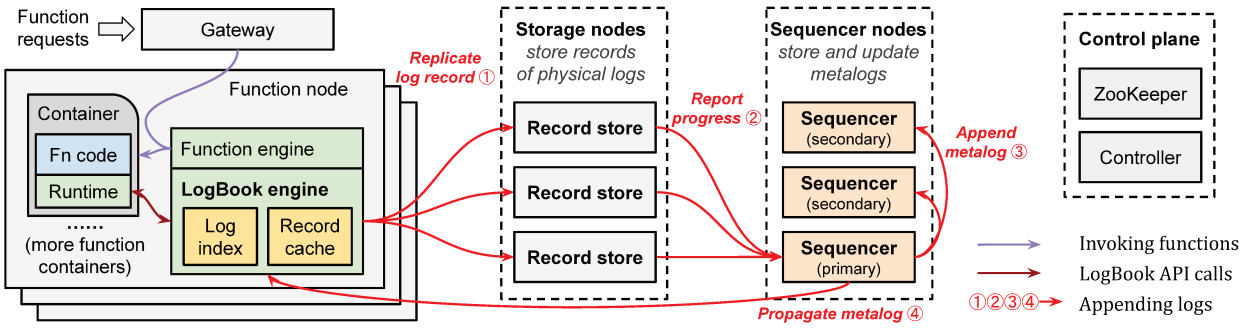


Figure 2. Architecture of Boki (§ 4.2), where red arrows show the workflow of log appends (§ 4.3).

code. LogBook engines maintain indices for physical logs, in order to efficiently serve LogBook reads (detailed in § 4.4). LogBook engines subscribe to the metalog, and incrementally update their indices in accordance with the metalog. LogBook engines also cache log records for faster reads, using their unique sequence numbers as keys. Co-locating LogBook engines with functions means that, in the best case, LogBook reads can be served without leaving the function node.

Control plane. Boki's control plane uses ZooKeeper [37] for storing its configuration. Boki's configuration includes (1) the set of storage, sequencers, and indices constituting each physical log; (2) addresses of gateway, function, storage, and sequencer nodes; (3) parameters of consistent hashing [40] used for the mapping between LogBooks and physical logs. Every Boki node maintains a ZooKeeper session to keep synchronized with the current configuration. ZooKeeper sessions are also used to detect failures of Boki nodes.

Boki's controller (see the control plane in Figure 2) is responsible for global reconfiguration. Reconfiguration happens when node failures are detected, or when instructed by the administrator to scale the system, e.g., by changing the number of physical logs (see §7.1 for reconfiguration latency measurements). We define the duration between consecutive reconfigurations as a *term*. Terms have a monotonically increasing *term_id*.

Structure of sequence numbers (seqnum). In Boki, every log record has a unique seqnum. The seqnum, from higher to lower bits, is (*term_id*, *log_id*, *pos*), where *log_id* identifies the physical log and *pos* is the record's position in the physical log. Seqnums in this structure determine a total order within a LogBook, which is in accordance with the chronological order of terms and the total order of the underlying physical log. But note that this structure cannot guarantee seqnums within a LogBook to be consecutive, whose records can be physically interspersed with other LogBooks.

4.3 Workflow of Log Appends

When appending a LogBook (shown by the red arrows in Figure 2), the new record is appended to the associated physical log. For simplicity, in this section, the term *log* always refers to physical logs.

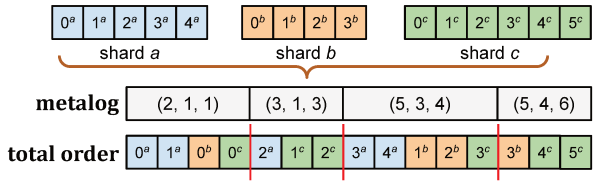


Figure 3. An example showing how the metalog determines the total order of records across shards. Each metalog entry is a vector, whose elements correspond to shards. In the figure, log records between two red lines form a delta set, which is defined by two consecutive vectors in the metalog (§ 4.3).

Records in a Boki log are sharded, and each shard is replicated on n_{data} storage nodes. Within a Boki log, each function node controls a shard. For a function node, its LogBook engine maintains a counter for numbering records from its own shard. On receiving a `logAppend` call, the LogBook engine assigns the counter's current value as the *local_id* of the new record.

The LogBook engine replicates a new record to all storage nodes backing its shard (① in Figure 2). Storage nodes then need to update the sequencers with the information of what records they have stored. The monotonic nature of *local_id* enables a compact progress vector, v. Suppose the log has M shards. We use a vector v of length M to represent a set of log records. The set consists of, for all shards j, records with $local\_id < v^j$. If shard j is not assigned to this node, we set the j-th element of its progress vector as ∞. Every storage node maintains their progress vectors, and periodically communicates them to the primary sequencer (② in Figure 2).

By taking the element-wise minimum of progress vectors from all storage nodes, the primary sequencer computes the global progress vector. Based on the definition of progress vectors, we can see the global progress vector represents the set of log records that are fully replicated. Finally, the primary sequencer periodically appends the latest global progress vector to the metalog (③ in Figure 2), which effectively orders log records across shards.

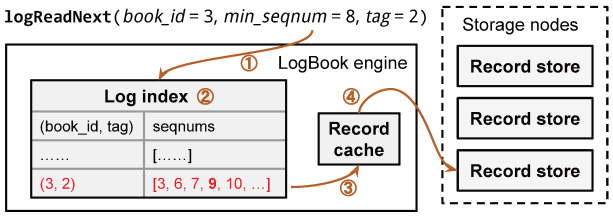


Figure 4. Workflow of LogBook reads (§ 4.4): ① Locate a LogBook engine stores the index for the physical log backing *book_id* = 3; ② Query the index row (*book_id*,*tag*) = (3,2) to find the metadata of the result record (seqnum = 9 in this case); ③ Check if the record is cached; ④ If not cached, read it from storage nodes.

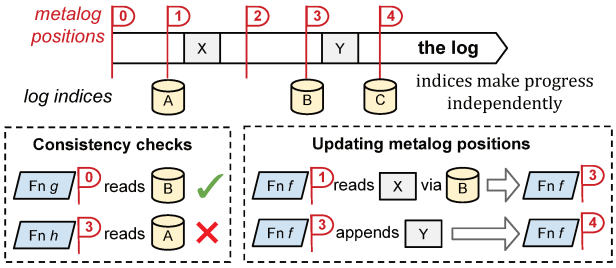


Figure 5. Consistency checks by comparing metalog positions (§ 4.4). For a function, if reading from a log index whose progress is behind its metalog position, it could see staled states. For example, function *h* have already seen record *X*, so that it cannot perform future log reads through index *A*.

We now explain how the total order is determined by the metalog. Consider a newly appended global progress vector, denoted by v_i. By comparing it with the previous vector in the metalog (denoted by v_{i-1}), we can define the delta set of log records between these two vectors: for all shards j, records satisfying $v^j_{i-1} \leq local\_id < v^j_i$. This delta set exactly covers log records that are added to the total order by the new metalog entry v_i. Records within a delta set are ordered by (*shard*, *local_id*). Figure 3 shows an example of metalog and its corresponding total order. In this figure, between two consecutive red lines is a delta set.

The LogBook engine initiating the append operation learns about its completion by its subscription to the metalog (④ in Figure 2). The metalog allows the LogBook engine to compute the final position of the new record in the log, used to construct the sequence number returned by `logAppend`.

4.4 From Physical Logs to LogBooks

Building indices for LogBooks. Boki virtualizes LogBooks by multiplexing them on physical logs, which creates a problem for efficient reads – avoiding consulting every log shard. Previous systems [55] have used fixed sharding, where a LogBook maps to some fixed shard, so that a single storage node has all of its records. But then a single storage node becomes the bottleneck for a LogBook's write throughput. For performance and operational advantages, Boki does not place records from a LogBook using a fixed policy. Boki will store LogBook records in any shard and it builds a log index for locating records when reading LogBooks.

Boki's log index is compact, only including necessary metadata of log records, so that a single machine can store the entire index. Log indices are stored and maintained by LogBook engines, leading to locality benefits because LogBook engines reside on function nodes. Every physical log has multiple copies of the log index maintained by different LogBook engines, for higher read throughput and better read locality.

The structure of the log index is designed to fit the semantic of LogBook read APIs. First, the log index groups records by their *book_id*, because a read can only target a single LogBook. The API semantics for `logReadNext` and `logReadPrev` (see Figure 1) allow selective reads by log tags (tags are specified by users in `logAppend`). Both APIs seek for records sequentially by providing bounds for seqnums, e.g., `logReadNext` finds the first record whose seqnum ≥ *min_seqnum*. Putting them together, Boki's log index groups records by (*book_id*,*tag*). For each (*book_id*,*tag*), it builds an index row as an array of records, sorted by their seqnums. Figure 4 depicts the workflow of LogBook reads using the index.

Read consistency. The consistency of Boki's log reads are determined by the log index. The log index is used to find the seqnum of the result record. The seqnum uniquely identifies a log record, while both data and metadata (i.e., tags) of a log record are immutable after they are appended.

The challenge of enforcing read consistency comes from multiple copies of the log index, which are maintained by different LogBook engines. Keeping these copies consistent makes the system vulnerable to "slowdown cascades" [19, 44], i.e., the slowdown of a single node can prevent the whole system from making progress.

Boki uses *observable consistency* [28, 44], where consistency checks are delayed to the time of data reads. The metalog position defines the version of the log index a function reads. A log index whose version is determined by metalog position l means the log index includes all records ordered by the l-prefix of the metalog.

When a user function reads a LogBook at an index with metalog position l, it can never read an index at $< l$, because that would violate *monotonic reads*. Similarly, if a function appends a log record that is ordered by the l-th metalog entry, subsequent reads from the same function cannot be served by an index whose position $< l$ or *read-your-writes* could be violated.

Therefore, Boki maintains a metalog position for each function and that position provides consistent LogBook reads. LogBook engines subscribe to the metalog to periodically update their indices. Consistency checks are performed by comparing a function's metalog position with the index version. Figure 5 depicts the mechanism. If a consistency check fails, the read is suspended by the engine until its index has

caught up. Successful reads and appends from a function update the function's metalog position, ensuring the consistency of future reads. A child function inherits the metalog position from its parent function, so that consistency guarantees hold across function boundaries.

Trim operations. Because the log index plays an important role in read consistency, trimming records in log indices effectively makes trim operations observable. Storage space for trimmed records can be reclaimed independently in the background by storage nodes. Therefore, Boki implements `logTrim` API calls by simply appending a trim command to the metalog. For a trim command in the metalog, the LogBook engines executes it by trimming related index rows in their log indices, while storage nodes gradually reclaim space for trimmed records.

Auxiliary data. Described in the LogBook API (§ 3), the auxiliary data of log records have relaxed requirements of durability and consistency. This allows a very simple store of auxiliary data that reuses the record cache within LogBook engines. The relaxed consistency of auxiliary data does not even require Boki to exchange them between nodes. Therefore, for `logSetAuxData` calls, Boki simply caches the provided auxiliary data on the same function node. To serve reads from the user function Boki checks if there is auxiliary data in the local cache. If found, it is returned along with the result record.

4.5 Reconfiguration Protocol

Boki's controller can initiate a reconfiguration if node failures (including failures of primary sequencers) are detected or when instructed by a system administrator.

The main part of Boki's reconfiguration protocol is to seal all current metalogs. A sealed metalog cannot have any more entries appended, so the corresponding physical log is sealed as well. Boki employs Delos [22]'s log sealing protocol, that is surprisingly simple but fault-tolerant. To seal a metalog, the controller sends the seal command to all relevant sequencers. On receiving the seal command, the primary sequencer stops issuing new metalog entries, while secondary sequencers commit to reject future metalog entries from the primary sequencer. The sealing is completed when a quorum of sequencers have acknowledged the seal command (see the Delos paper [22] for details).

After all metalogs are successfully sealed, Boki can install a new configuration to start the next term. In the new term, all physical logs start with new, empty metalogs. To ensure read consistency across terms, we include the *term_id* in the consistency check, which is compared before metalog positions. If the number of physical logs changes, the consistent hashing parameters are updated accordingly.

To tolerate failures of the controller, Boki runs a group of controller processes. The reconfiguration protocol is executed by a leader, elected via ZooKeeper.

5 Boki Support Libraries

In this section, we present Boki support libraries, designed for three different stateful FaaS paradigms that benefit from the LogBook API: fault-tolerant workflows (§ 5.1), durable object storage (§ 5.2), and queues for message passing (§ 5.3).

5.1 BokiFlow: Fault-Tolerant Workflows

We build a support library called BokiFlow for fault-tolerant workflows. BokiFlow adapts Beldi [56]'s techniques to ensure exactly-once semantics and support transactions for serverless workflows.

In a Beldi workflow, every operation that has externally visible effects (e.g., a database write) is logged with monotonically increasing *step* numbers. When a workflow fails, Beldi re-executes it using the workflow log. To ensure the exactly-once semantic, Beldi recovers the internal state of the failed workflow step-by-step, while skipping operations with externally visible effects. Beldi builds a logging abstraction on top of DynamoDB, a cloud database from AWS. Beldi applications store user data in the same DynamoDB database with workflow logs.

BokiFlow implements Beldi's techniques by using LogBooks as the logging layer, i.e., logging every workflow step in a LogBook. Similar to Beldi, BokiFlow applications store user data in DynamoDB, so that BokiFlow provides the same user-facing APIs as Beldi. There are three ways BokiFlow distinguishes itself from Beldi.

Atomic "test-and-append". Beldi requires an atomic operation to check if the current step is previously logged and it logs the step only if the check fails. Beldi relies on conditional updates provided by a cloud database for this operation. Unfortunately, the LogBook API does not support conditional log appends. Shown in Figure 6 (a), BokiFlow uses a different mechanism based on log tags provided by LogBooks. The pseudocode shows how BokiFlow uses log tags to distinguish the log records of workflow steps. BokiFlow always reads log records immediately after appends, and only honors the first record of a step. This allows BokiFlow to recognize completed steps during workflow re-execution, by checking if the appended record is the first one.

Idempotent database update. For a workflow step that updates the database, Beldi requires the database update and logging of this step to be a single atomic operation. Because Beldi stores its logs along with user data in the same database, it can use the atomic scope provided by the database (e.g., a row in DynamoDB) for this requirement. However, BokiFlow's LogBook is not in the same atomic scope as user data, so no mechanism exists to update both in a single atomic operation. Instead, BokiFlow makes data updates *idempotent*. Pseudocode in Figure 6 (a) demonstrates the approach, where the `rawDBWrite` statement uses the sequence number of the

```
def write(table, key, val):
    tag = hashLogTag([ID, STEP])
    logAppend(tags: [tag], data: [table, key, val])
    rec = logReadNext(tag: tag, minSeqnum: 0)
    rawDBWrite(table, key,
        cond: "Version < {rec.seqnum}",
        update: "Value={val}; Version={rec.seqnum}")
    STEP = STEP + 1

def invoke(callee, input):
    tagPre = hashLogTag([ID, STEP, "pre"])
    logAppend(tags: [tagPre],
              data: { "calleeId": UUID() })
    rec = logReadNext(tag: tagPre, minSeqnum: 0)
    calleeId = rec.data["calleeId"]
    retVal = rawInvoke(callee, [calleeId, input])
    tagPost = hashLogTag([ID, STEP, "post"])
    logAppend(tags: [tagPost],
              data: { "retVal": retVal })
    rec = logReadNext(tag: tagPost, minSeqnum: 0)
    STEP = STEP + 1
    return rec.data["retVal"]
```

(a) BokiFlow's write and invoke functions

```
def checkLockState(key):
    tail = None
    for rec in logIterRecords(tag: key):
        if tail == None ||
                rec.data["prev"] == tail.seqnum:
            tail = rec
    return tail

def tryLock(key, holderId):
    rec = checkLockState(key)
    if rec.data["holder"] == EMPTY:
        logAppend(tags: [key], data: {
          "holder": holderId, "prev": rec.seqnum})
        rec = checkLockState(key)
    if rec.data["holder"] == holderId:
        # Lock succeeded
        return rec   # rec is used for unlock
    return None  # Lock failed

def unlock(key, lockRec):
    logAppend(tags: [key], data: {
       "holder": EMPTY, "prev": lockRec.seqnum})
```

(b) BokiFlow's lock

```
# Get the object with name "x"
x = getObject("x")
# Suppose object x is
# {"a":{}, "b":"foo"}
print(x.Get("b")) # ⇒ "foo"
x.Set("a.c", "bar")
# x ⇒ {"a":{"c":"bar"},"b":"foo"}
x.MakeArray("a.d")
x.PushArray("a.d", 1)
# x ⇒ {"a":{"c":"bar", "d":[1]},
#       "b":"foo"}

txn = createTransaction(readonly: False)
alice = txn.GetObject("alice")
bob = txn.GetObject("bob")
if alice.Get("balance") ⩾ 10:
    alice.Inc("balance", -10)
    bob.Inc("balance", 10)
txn.Commit()
```

(c) Demonstration of BokiStore API

Figure 6. Pseudocode demonstrating Boki support libraries (§ 5). In BokiFlow pseudocode, `hashLogTag` computes a hashing-based log tag for the provided tuple. `logIterRecords` iterates over log records having the provided tag in the increasing order of their sequence numbers.

step log as the "version" of the database update. During workflow re-execution, re-executing this database update will fail the update condition.

Locks. Beldi provides locks for mutual exclusion; locks also serve as building blocks for Beldi's transactions. Implementing locks requires an atomic "test-and-set" operation, where Beldi uses conditional updates provided by the database. BokiFlow implements locks as registers backed by replicated state machines using the LogBook API. For a BokiFlow lock, its register stores the lock holder (unique identifiers such as UUID), or a special `EMPTY` value. The most natural way to "test" a lock is to execute a predicate on the current state machine. The most natural "set" is to append an update. When we try to combine these operations into a "test-and-set", the LogBook API cannot linearize the result because other BokiFlow clients may also append updates to the same state machine. BokiFlow's solution is to include the log position of the current state machine in the log record of the proposed update. On log replay, only choose the first of any updates that were concurrently proposed. In this way, the total order provided by the LogBook API becomes a mechanism for linearizability.

Pseudocode in Figure 6 (b) demonstrates BokiFlow locks. The lock uses the *prev* field to store the log position, as shown in Figure 7. The "prev" pointers form a linearizable chain of state machine updates. This technique provides a general approach for building linearizable replicated state machines with the LogBook API.

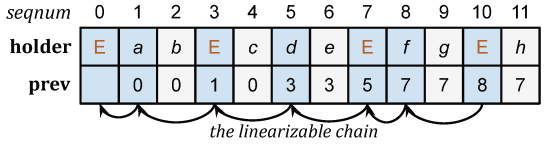


Figure 7. An example log behind a BokiFlow lock. Holders $\{a,d,f\}$ acquire the lock. *prev* pointers form an implicit linearizable chain, which alternates successful acquire and release attempts.

5.2 BokiStore: Durable Object Storage

The second support library we built is BokiStore, providing durable object storage for stateful functions. BokiStore employs Tango's [24] techniques for building replicated data structures over a shared log. BokiStore's objects are represented as JSON objects. Objects are identified by unique string names. Figure 6 (c) shows the BokiStore APIs for reading and modifying fields of JSON objects. BokiStore stores all object updates within a LogBook. Reading object fields requires replaying the log to re-constructs the object's state. Log records containing object updates are tagged with object names, so that objects can be re-constructed by only reading relevant records.

Transactions. BokiStore supports transactions for reading and modifying multiple objects. BokiStore's log-based transaction protocol largely follows Tango. To start a transaction, BokiStore first appends a *txn_start* record with its *txn_id*. For all subsequent object reads within the transaction, BokiStore only replays the log up to the position of its *txn_start* record. This essentially takes a snapshot of the entire object storage at the *txn_start* position, which achieves *snapshot isolation*.

When committing the transaction, BokiStore appends a *txn_commit* record, including its *txn_id* and all object writes made within the transaction. The *txn_commit* record is *speculative* – by itself, it does not indicate the success of this transaction. The commit outcome of a transaction is determined by replaying the log up to its *txn_commit* record. A transaction succeeds in committing if and only if there is no conflicting

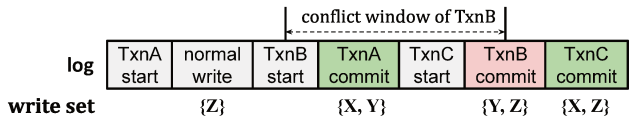


Figure 8. Transactions in BokiStore (§ 5.2). *TxnB* fails due to conflict with *TxnA*. For *TxnC*, despite its write set overlaps with *TxnB*'s, *TxnC* still succeeds due to the failure of *TxnB*.

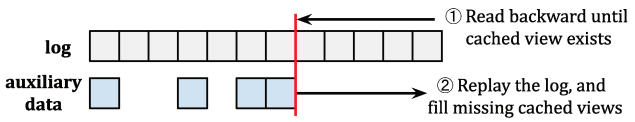


Figure 9. Use auxiliary data to cache object views in BokiStore, which can avoid a full log replay (§ 5.4).

write made between its *txn_start* and *txn_commit* records (i.e., within the conflict window). Figure 8 depicts a transaction log. In this example, *TxnB* is a failed transaction and it is ignored when determining the commit outcome of *TxnC*.

Read-only transactions in BokiStore are simpler. They do not need *txn_start* and *txn_commit* records, because there is no need for conflict detection. To achieve isolation, BokiStore simply caches the tail position of the current log when starting the transaction, and only replays records until the cached position to serve following object reads.

5.3 BokiQueue: Message Queues

Queues are the most common data structure for message passing. The final support library we build is BokiQueue which provides serverless queues. BokiQueue provides a push and pop API for sending and receiving messages. Like BokiStore, BokiQueue uses the log to store all writes, i.e., push and pop operations. The outcome of a pop is determined by replaying the log. To improve the scalability of BokiQueue, we uses vCorfu [55]'s composable state machine replication (CSMR) technique, that divides a single queue into multiple SMR-backed queue shards. Each queue shard is consumed by a single consumer, which reduces contention. A queue producer can choose an arbitrary queue shard to push. In our implementation, we simply use round-robin.

5.4 Optimizing Log Replay with Auxiliary Data

Reads in BokiStore are served by replaying the log to reconstructs object state. This naive approach makes read latency proportional to the number of relevant log records, i.e., the number of object writes. Tango optimizes log replay by caching local object views, such that only new records from the shared log are replayed. However, in the FaaS setting, in-memory state is not guaranteed to be preserved between invocations, so a simple memory cache for objects is not a viable solution.

Boki's auxiliary data (§ 3) is motivated by the need to provide per-log-record cache storage. In BokiStore, for every object write that generates a log record, the auxiliary data of the record stores a snapshot view of the object. When reading an object, BokiStore seeks from the log tail to find the first relevant record having a cached object view in its auxiliary data. Then BokiStore replays the log from this position to re-construct the target object state. During replay, for records missing cached object views, their auxiliary data are filled with object views. Figure 9 demonstrates this accelerated replay process.

One important special case for accelerating log replay is commit records. For *txn_commit* records, their auxiliary data stores the decided commit outcome and if the commit succeeds, the auxiliary data also caches a view of modified objects.

In BokiFlow's log-based locks (shown in Figure 6 (b)), auxiliary data of a record is used for caching the current tail of the linearizable chain. This allows the `checkLockState` function to optimize its log replay as illustrated in Figure 9.

5.5 Garbage Collector Functions

The FaaS paradigm simplifies garbage collection in shared-log-based storage systems. Boki support libraries use garbage collector functions to trim useless log records, in order to prevent unlimited growth of LogBooks. These functions are periodically invoked and they reclaim space via LogBook's `logTrim` API (Figure 1).

In BokiFlow, garbage collector functions trim log records produced by completed workflows. In BokiStore, log records from deleted objects are trimmed. In BokiQueue, log records related to popped queue elements are trimmed.

6 Implementation

The Boki prototype is based on Nightcore [16], where we add 9,726 lines of code, mostly in C++. Boki's support libraries are implemented in Go, consisting of 2,776 lines of code. One of the support libraries, BokiFlow, derives from the Beldi codebase [11]. The LogBook API makes Beldi's techniques easier to implement, so that BokiFlow shrinks the Beldi library from 1,823 lines to 1,137 lines, or a 38% reduction.

Boki's storage nodes use RocksDB [12] for storing log records. LogBook engines use Tkrzw [14]'s LRU cache DBM for caching records. LogBook engines store and maintain log indices in DRAM, while future implementation can choose to use on-disk data structures (e.g., B-trees) if DRAM is scarce. In the current prototype, metalogs are replicated in the DRAM of sequencer nodes. This is viable in our failure assumption that requires a quorum of sequencers for a metalog to always be alive.

Boki uses 64-bit integers as the tag type for LogBook records. In Boki support libraries, when we need other types (e.g., strings) as the log tag, we use their hash value for the log tags and store the string in record data. Boki employs Dynamo [29]'s variant of consistent hashing (strategy 3 in their paper) for mappings between LogBooks and physical logs.

7 Evaluation

In this section, we first evaluate Boki with microbenchmarks to explore its performance characteristics (§ 7.1). We then evaluate Boki's support libraries using realistic workloads (§ 7.2, § 7.3, and § 7.4). Finally, we analyze how Boki's techniques benefit its use cases (§ 7.5).

Experimental setup. We conduct all our experiments on Amazon EC2 instances in the us-east-2 region. Boki's function, storage, and sequencer nodes use c5d.2xlarge instances, each of which has 8 vCPUs, 16GiB of DRAM, and 1×200GiB NVMe SSD. Boki's gateway and control plane use c5d.4xlarge instances. Experimental VMs run Ubuntu 20.04 with Linux kernel 5.10.17, with hyper-threading enabled. We measure that the round trip time between VMs is 107μs ±15μs, and the network bandwidth is 9,681 Mbps.

Unless otherwise noted, the following Boki settings are fixed in our experiments: (1) the ZooKeeper cluster in the control plane has 3 nodes; (2) the replication factors of both physical logs (n_{data}) and metalogs (n_{meta}) equal 3; (3) one single physical log configured for all LogBooks; (4) for each physical log, there are 4 LogBook engines that store its index (though functions can read their LogBooks via remote engines); (5) the record cache per LogBook engine is 1GB (for both record data and auxiliary data §3).

7.1 Microbenchmarks

We start the evaluation of Boki using microbenchmarks, where we answer the following questions.

- *What is the append throughput of a single LogBook?* We use an append-only workload to measure the throughput, and how the throughput scales with more resources. In this workload, each function is a loop of appending 1KB log records. Results are shown in Table 2a. From the table, we see that when Boki is configured with 64 nodes, the append throughput scales to 1.2M Ops/s under 2,560 concurrent appending functions. At this point, the median latency is 2.03ms, and the p99 tail latency is 6.42ms. We also increase the replication factor of metalogs (n_{meta}) to 5, that provides higher durability for a metalog but potentially affects the metalog's append latency. However, it demonstrates similar LogBook throughput and scalability as $n_{\text{meta}}=3$.
- *Can Boki efficiently virtualize LogBooks?* We use the same append-only workload, but log appends are uniformly distributed over many LogBooks. We use 1, 2, and 4 physical logs to virtualize 100 and 100K LogBooks. Boki is configured with 4 function and 4 storage nodes when using one physical log, and resources are added linearly with more physical logs. Table 2b shows the results. From the table, we can see Boki is capable of virtualizing LogBooks with high density.
- *How fast can Boki functions read LogBook records?* We use an append-and-read workload to measure read latencies, where each function loops a procedure that first appends a log record, then reads the appended record 4 times. We configure Boki with 8 function and 8 storage nodes. Table 3 shows the results. For *remote engine* case, we enforce Boki to use remote LogBook engines for log reads. Cache hits take 121μs and never leave the local LogBook engine, retrieving the result from the record cache (§ 4.4).
- *What is the impact of reconfiguration?* We use the append-only workload to evaluate the impact of reconfiguration. In the experiment, Boki is reconfigured to a new set of sequencer nodes. New sequencer nodes are provisioned before the reconfiguration, to factor out provisioning delays from the experiment. Figure 10 shows the results. We see that Boki recovers to normal append latency after reconfiguration within 100ms. The actual reconfiguration protocol, executed by the controller, takes 15.7ms and 18.1ms, in experiments of $n_{\text{meta}}=3$ and $n_{\text{meta}}=5$, respectively.

Table 2. Boki's throughput in append-only microbenchmark (§ 7.1).

	Concurrent functions / Storage (S) nodes			
	320/4S	640/8S	1280/16S	2560/32S
$n_{\text{meta}}=3$	130.8	279.2	604.4	1157.8
$n_{\text{meta}}=5$	142.3	282.8	583.2	1147.9

(a) Append throughput (in KOp/s) of a single LogBook, where n_{meta} denotes the replication factor of Boki's metalog. Boki can scale append throughput of a totally ordered log to 1.2M Ops/s.

	1 PhyLog	2 PhyLogs	4 PhyLogs
100 LogBooks	120.5	226.3	458.2
100K LogBooks	122.3	225.6	446.9

(b) Aggregate throughput (in KOp/s) when using multiple physical logs (PhyLogs) to virtualize LogBooks. Boki scales with more physical logs, and can efficiently virtualize 100K LogBooks.

Table 3. Boki's read latencies under different scenarios (§ 7.1).

	Local LogBook (LB) engine		Remote LB engine
	cache hit	*cache miss*	
median	0.12ms	0.57ms	0.79ms
99% tail	0.72ms	1.48ms	2.90ms

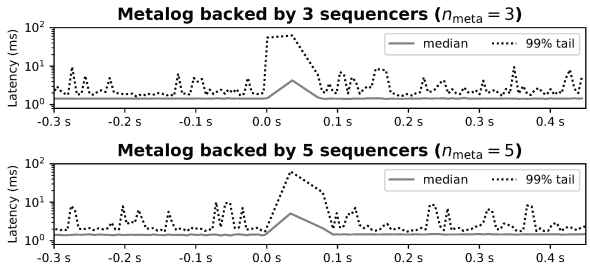
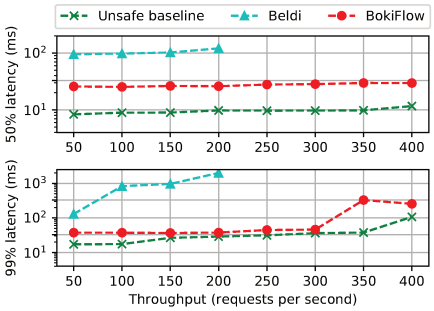
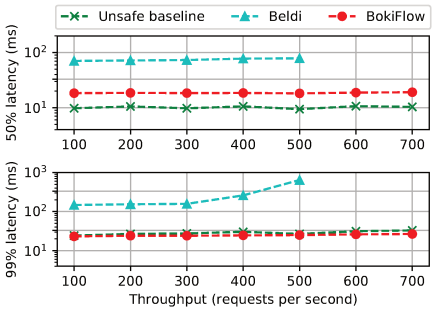


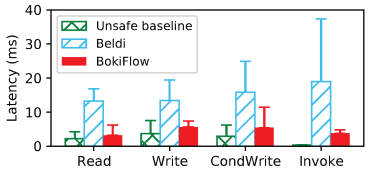
Figure 10. Log append latencies during reconfiguration. The x-axis shows the timeline (in seconds). The reconfiguration starts at $t=0$.



(a) Movie review workload.



(b) Travel reservation workload.



(c) Microbenchmarks of Beldi primitive operations. Main bars show median latencies, while error bars show 99% latencies.

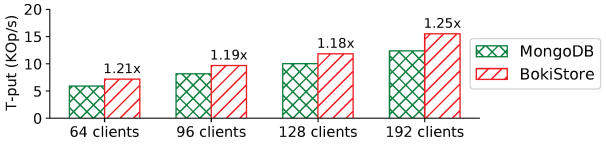
Figure 11. Comparison of BokiFlow with Beldi [56]. BokiFlow takes advantage of the LogBook API. "Unsafe baseline" refers to running workflows without Beldi's techniques, where it cannot guarantee exactly-once semantics or support transactions (§ 7.2).

7.2 BokiFlow: Fault-Tolerant Workflows

We evaluate BokiFlow by comparing it with Beldi [56]. We use Beldi's workflow workloads, which model movie reviews and travel reservations. Both of them are adapted from DeathStarBench [8, 33] microservices. For a fair comparison, we port Beldi and its workloads to Nightcore, the underlying FaaS runtime of Boki. Both BokiFlow and Beldi store user data in DynamoDB [1]. BokiFlow stores workflow logs in a LogBook, while Beldi uses its linked DAAL technique to store logs in DynamoDB. For both systems, they are configured with 8 function nodes and Boki is configured with 3 storage nodes.

Figure 11 shows the results. In both workloads, BokiFlow achieves much lower latencies than Beldi for all throughput values. In the movie workload, when running at 200 requests per second (RPS), BokiFlow's median latency is 26ms, 4.7× lower than Beldi (121ms). In the travel workload, BokiFlow's median latency is 18ms at 500 RPS, 4.3× lower than Beldi (78ms). In this experiment, we also run a baseline without Beldi's techniques, where it cannot guarantee exactly-once semantics or support transactions for workflows. When comparing BokiFlow with this baseline, we see that exactly-once semantics and transactions increase median latency by 3.0× in the movie workload, and by 1.8× in the travel workload.

We then run the microbenchmark that evaluates Beldi's primitive operations (Figure 13 in the Beldi paper [56]). Results are shown in Figure 11c. The *Invoke* operation shows the largest differences among the three implementations and *Invoke* operations are very frequent in microservice-based workflows. In the baseline without workflow logs, the *Invoke* operation is very fast (well below 1ms). The underlying FaaS runtime, Nightcore, is heavily optimized to reduce invocation latencies. In BokiFlow, the *Invoke* operation needs needs 5 LogBook appends, thus it has a median latency of 3.8ms. 2 of the 5 log appends are demonstrated in Figure 6 (a) and the other 3 appends are made within the child function. For comparison, *Invoke* operation in Beldi also need 5 log appends, but has a median latency of 19ms, because of multiple DynamoDB updates for each log append. These results justify the value of shared logs for the serverless environment, where building logging layers using a cloud database is difficult to make performant.



(a) Throughput of BokiStore compared with MongoDB.

Request types	50% latency		99% latency	
	Mongo	Boki	Mongo	Boki
`UserLogin` (non-txn read)	**0.86**	1.47	**3.32**	6.32
`UserProfile` (non-txn read)	**0.86**	1.05	**3.57**	5.29
`GetTimeline` (read-only txn)	7.57	**3.35**	25.01	**11.38**
`NewTweet` (read-write txn)	7.72	**5.30**	21.39	**15.33**

(b) Latencies (in ms) under 192 clients. Although non-transactional reads in BokiStore are slower than MongoDB, transactions in BokiStore are up to 2.3x faster. Best performing result is in bold.

Figure 12. Evaluating BokiStore on Retwis workload (§ 7.3).

7.3 BokiStore: Durable Object Storage

Retwis workload. To evaluate BokiStore, we build a transaction workload inspired by Retwis, a simplified Twitter clone [15]. The Retwis workload has been used as a transaction benchmark in previous work [57, 58]. We re-implement the Retwis workload in Go, requiring 1,458 lines of code. Our implementation uses BokiStore objects to store users, tweets, and timelines. For comparison, we also implement a version that uses MongoDB [13] to store objects, because MongoDB also employs a JSON-derived data model.

The evaluation workload first initializes 10,000 users, and then runs a mixture of four functions: UserLogin (15%), UserProfile (30%), GetTimeline (50%), and NewTweet (5%). UserLogin are UserProfile are normal single object reads. GetTimeline is a read-only transaction that reads the timeline and multiple tweets. NewTweet is a transaction that writes multiple user, tweet, and timeline objects.

In the experiment, we configure Boki with 8 function nodes and 3 storage nodes. MongoDB is configured with 3 replicas. For BokiStore, we configure LogBook engines on all 8 function nodes to have log index for the target LogBook, which achieves best data locality. We analyze the performance impact of using remote LogBook engines in § 7.5.

Figure 12 shows the results. From the figure, we see BokiStore achieves 1.18–1.25× higher throughput than MongoDB. When breaking down latency details by request types, we see BokiStore has considerable advantages over MongoDB in transactions (up to 2.3× faster). On the other hand, BokiStore is slower than MongoDB for non-transactional reads. This is caused by the log-structure nature of BokiStore, where log replay incurs overheads for data reads.

Comparison with Cloudburst. Cloudburst [52] is a recently proposed stateful FaaS runtime, which exports a put/get interface (i.e., key-value store) for functions to store state. BokiStore can also be used as a key-value store, by using keys as object names and storing values in the corresponding BokiStore object. However, BokiStore provides stronger consistency guarantees (sequential) than Cloudburst (causal). BokiStore also supports transactions reading and modifying multiple keys, which are not supported by Cloudburst.

We use a microbenchmark to compare Cloudburst's performance with BokiStore. Both systems use 8 storage nodes and 8 function nodes in the experiment. Figure 13 shows the result. BokiStore can achieve up to 2.01× higher throughput than Cloudburst on get operations. For put operations, BokiStore achieves 1.23× higher throughput when the concurrency is high. BokiStore provides higher throughput and lower median latency at 192 clients than Cloudburst, but it does have higher tail latency.

7.4 BokiQueue: Message Queues

We evaluate BokiQueue by comparing it with Amazon Simple Queue Service (SQS) [2] and Apache Pulsar [3]. Amazon SQS is a fully managed message queue service from AWS, while Pulsar is a popular open source distributed message queue. Similar to BokiQueue, both SQS and Pulsar use sharding to improve the data throughput of their message queues. In the experiment, we configure Boki with 8 function nodes and 3 storage nodes. For Pulsar, we run its broker services on function nodes for better locality, and use the 3 storage nodes for queue data.

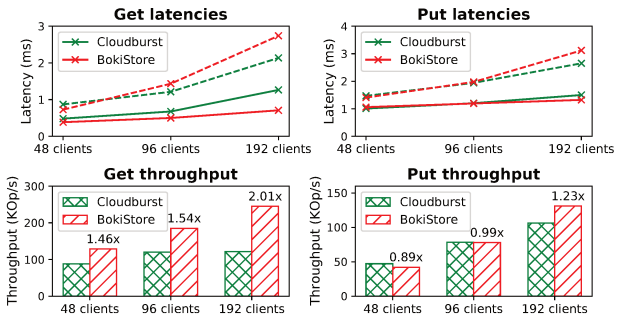


Figure 13. Comparison of BokiStore with Cloudburst [52]. We measure the latencies and throughput for put and get operations, using different numbers of concurrent clients. In the latency charts, solid lines show median latencies, and dashed lines show 99% tail latencies. BokiStore not only provides stronger consistency guarantees, but also achieves higher performance than Cloudburst (§ 7.3).

Table 4. Comparison of BokiQueue with Amazon SQS [2] and Pulsar [3]. Throughput is measured in 10^3 message/s. Delivery latency is the duration that a message stays in the queue. Latencies are shown in the form of "median (99% tail)". Best performing result is in bold.

Producer/ Consumer	Throughput			Delivery latency (ms)		
	SQS	Pulsar	Boki	SQS	Pulsar	Boki
16P/64C	2.25	5.05	**5.06**	6.27 (52.5)	4.01 (12.3)	**2.98 (9.17)**
32P/128C	4.03	9.67	**9.92**	6.01 (51.3)	6.70 (12.8)	**3.28 (12.7)**
64P/256C	7.62	14.1	**15.0**	6.08 (56.5)	7.39 (13.7)	**3.70 (14.7)**
64P/16C	2.34	8.71	**8.90**	33.9 (228)	**6.20 (12.7)**	6.40 (27.3)
128P/32C	5.35	14.6	**15.6**	53.9 (370)	**7.38 (14.0)**	7.45 (32.8)
256P/64C	9.77	19.1	**20.9**	99.8 (764)	7.81 (33.7)	**6.61 (37.9)**
64P/64C	6.37	10.0	**10.3**	7.22 (76.0)	6.77 (12.9)	**3.37 (14.2)**
128P/128C	10.1	17.8	**20.5**	7.24 (79.6)	7.74 (21.4)	**4.37 (17.7)**
256P/256C	18.5	25.0	**30.8**	12.1 (84.5)	8.21 (39.5)	**7.96 (35.5)**

We use a fixed number of producer and consumer functions for the evaluation, where each producer keeps pushing 1KB messages to the queue. We experiment with three ratios of producers to consumers (P:C ratio), which are 1:4, 4:1, and 1:1. In the evaluation, we measure the message throughput of the queue, and the median and p99 latency of message deliveries.

Table 4 shows the results. When the P:C ratio is 1:4, the queue is lightly loaded. We see both BokiQueue and Pulsar achieve double the throughput of Amazon SQS. BokiQueue achieves up to 2.0× lower latencies than Pulsar. When the P:C ratio is 4:1, the queue is saturated. Amazon SQS suffers significant queueing delays, limiting its throughput. BokiQueue and Pulsar have very similar throughput, while BokiQueue achieves 1.18× lower latency than Pulsar in the case of 256 producers. Finally, when the P:C ratio is 1:1, the queue is balanced. BokiQueue consistently achieves higher throughput and lower latency than both Amazon SQS and Pulsar.

Combining these three cases, BokiQueue achieves 1.66–2.14× higher throughput than Amazon SQS, and up to 15× lower latency. Compared with Pulsar, BokiQueue achieves 1.06–1.23× higher throughput, and up to 2.0× lower latency.

7.5 Analysis

The importance of auxiliary data. We describe in § 5.4 the log replay optimization using LogBook's auxiliary data. We use Retwis workload to demonstrate its importance for BokiStore. We run an experiment that disables this optimization. Furthermore, to demonstrate the efficiency of Boki's storage mechanism for auxiliary data, we modify Boki to store auxiliary data in a dedicated Redis instance.

Table 5. The importance of log replay optimization using auxiliary data (§ 7.5). The table shows Retwis throughput (in Op/s).

Workload duration	1min	3min	10min	30min
Optimization disabled	1,565	939	–	–
AuxData w/ Redis	11,014	10,046	9,548	9,344
AuxData w/ Boki	11,388	11,078	10,923	10,891

Table 6. Locality impact from LogBook engines (§ 7.5). The table shows Retwis throughput (in Op/s), when adjusting the percentage of reads processed by local LogBook engines.

Local reads	25%	50%	75%	100%
Throughput	8,548	9,319	10,262	11,078
Normalized tput	0.77x	0.84x	0.93x	1.00x

Table 5 shows the results. From the table, we see that the log replay optimization is crucial for BokiStore to achieve an acceptable performance. The results also show the optimization is robust even for long executions, where more object writes are logged. Compared to the Redis-backed implementation, Boki achieves 1.17× higher throughput. Boki's approach is more efficient because it maintains data locality by reusing the record cache within LogBook engines.

Locality impact from LogBook engines. In the previous evaluation of BokiStore, we configure Boki so all LogBook reads are served by local LogBook engines. In a large-scale deployment, having all LogBook engines maintain an index for a particular physical log is not viable. Boki relies on the function scheduler to optimize for the locality of LogBook engines.

To experiment with the impact from using remote LogBook engines we limit the ratio of log reads that are locally processed, with the remainder processed remotely. Table 6 shows the results. We see even under a poor locality of LogBook engines, the performance drop is moderate (e.g., 77% of maximum throughput at 25% local reads).

Read locality also comes from the record cache included in LogBook engines. The cache stores both record data and auxiliary data for LogBook records. We experiment with different cache sizes to analyze its impact on BokiStore performance. Results are shown in Table 7. We observe a sharp dorp in throughput when the cache size is decreased to 16MB. The cause of this drop is insufficient cache storage for auxiliary data. Auxiliary data is important for BokiStore performance, and a small record cache decreases the effectiveness of the log replay optimization. We modify Boki to backup auxiliary data on storage nodes, so that under a cache miss, storage nodes can also return auxiliary data. With this mechanism, small cache sizes no longer cause a sharp dorp in performance.

Log index versus fixed sharding. In § 4.4, we motivate the log index design because it allows records from a LogBook to be placed in arbitrary log shards. An alternative approach is fixed sharding used in previous systems such as vCorfu [55]. We use the append-only microbenchmark to demonstrate the advantage of Boki's approach. For comparison, we modify Boki to use a fixed sharding approach, where a hashing function maps each LogBook to a log shard. Results are shown in Table 8. When log appends are uniformly distributed over LogBooks, the two approaches show no difference. However, when the distribution is skewed, fixed sharding suffers from uneven loads between log shards, while Boki's log index approach is unaffected.

Table 7. LogBook engines maintain local cache for log records, and the cache size has performance impact for Boki's applications (§ 7.5). The table shows Retwis throughput (in Op/s).

LRU cache size	16MB	32MB	64MB	1GB
Auxiliary data only stored on function nodes				
Throughput	3,561	10,476	11,263	11,245
Auxiliary data also backed up on storage nodes				
Throughput	11,358	11,852	12,032	12,075

Table 8. Append throughput (in KOp/s) when log appends are distributed over 128 LogBooks under a uniform or Zipf distribution.

	Uniform	**Zipf** ($s=3$)	**Zipf** ($s=5$)
Fixed sharding	242.7	164.0	129.6
Log index (Boki)	250.6	253.4	278.6

Table 9. Scaling read-only transactions with LogBook engines (§ 7.5). The experiment runs Retwis workload under a fixed write rate.

	Concurrent functions / LogBook engines				
	100/8E	200/16E	300/24E	400/32E	600/48E
T-put (txn/s)	6,548	12,749	18,618	23,662	30,286
Normalized	1.00x	1.95x	2.84x	3.61x	4.63x

Scaling LogBook engines. We then demonstrate the scalability of LogBook engines, by running read-only transactions in the Retwis workload. The workload is a mixture of read-only transactions (GetTimeline) and read-write transactions (NewTweet). In the experiment, we add more function nodes to scale LogBook engines, while always using 3 storage nodes. Every LogBook engine maintains a log index for the target LogBook. We fix the rate of NewTweet to 700 requests per second. Results are shown in Table 9. The results demonstrate Boki can scale from 8 LogBook engines to 48, thereby providing 4.63× higher read throughput.

Sensitivity study of reconfigurations. We finally study how reconfiguration frequency affects Boki's performance. In the experiment, Boki is configured with a single physical log using $n_{\text{meta}} = 3$. To allow reconfigurations without frequently allocating new nodes, we provision redundant nodes for Boki. In the experiment, 8 sequencer nodes are provisioned, while

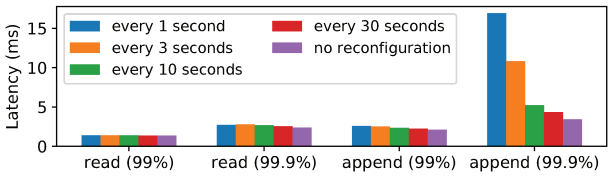


Figure 14. Sensitivity study of LogBook latencies to reconfiguration frequency (§ 7.5). Reconfigurations have little impact on log read latencies, but can significantly affect tail latencies of log appends when they are frequent. In all tested frequencies, throughput of log reads and appends is not affected (same as "no reconfiguration"). Data are collected over a 5-minute period.

only 3 of them are active at one time because $n_{\text{meta}} = 3$. Reconfigurations are manually triggered periodically with a fixed frequency, from every 1 second to every 30 seconds. For each reconfiguration, 3 sequencer nodes are randomly chosen to store the metalog in the new term. We run a workload of log appends and reads (check tail), where the ratio between appends and reads is 1:4. 320 concurrent functions are executed over 8 function nodes. Results are shown in Figure 14. For read operations, we see that even frequent reconfigurations have little impact on their latencies. But for append operations, when reconfigurations become very frequent, their tail latencies increase significantly.

8 Related Work

Stateful serverless computing. State management remains a key challenge in the current serverless environment [36, 48]. To meet the increasing demand for stateful serverless, there are recent attempts from industry, e.g., Cloudflare's Durable Objects [17] and Azure's Entity Functions [9]. These systems are still in their early stages and have seen limited adoption.

There are also proposals from academia, e.g., Pocket [42], Cloudburst [52], and Faasm [51]. These projects have different focus, e.g., heterogeneous storage technology [42], lightweight isolation [51], and auto-scaling [52]. These systems all export put-get interfaces (i.e., a key-value store) for functions to manage state. Boki is the first to study a different interface for serverless state management, the shared log API. Boki's shared log approach is motivated by the fault-tolerance and consistency challenges encountered by stateful serverless applications, which the put-get interface cannot easily address.

A recent article [48] argues future serverless abstractions will be general-purpose, where cloud providers expose a few basic building blocks, e.g., cloud functions (FaaS) for computation and serverless storage for state management. The shared log and key-value store are both promising storage building blocks, which can work together to enable new serverless applications.

Distributed shared logs. Recent studies on distributed shared logs [22–25, 30, 43, 55] heavily inspire the design of Boki. A shared log is a powerful primitive for achieving strong data consistency in the presence of failures, because it can be used for state machine replication (SMR) [49], the canonical approach for building fault-tolerant services.

Boki leverages Scalog [30]'s high-throughput ordering protocol. Virtual consensus in Delos [22] inspires Boki's design of metalogs. Materialized streams in vCorfu [55] inspire the design of log tags in the LogBook API, and LogBook's virtualization. However, Boki's metalog design distinguishes it from these prior works. The logical decoupling provided by the metalog allows existing techniques to be adopted smoothly, while enabling new techniques, e.g., the log index for read efficiency. For applications, Tango [24]'s techniques enable serverless durable objects [17] backed by shared logs.

Fault-tolerant workflows. Orchestrating serverless functions as workflows is an important serverless paradigm, provided by all major cloud providers [5, 10, 18]. Workflows aim at providing *exactly-once* execution semantics, but stateful serverless functions (SSF) complicate this goal.

Beldi [56] proposes solutions for current serverless platforms. Beldi's mechanism is inspired by Olive [50]'s log-based fault tolerance protocol. In a Beldi workflow, during execution of SSF operations, the actions are logged. Beldi periodically re-executes SSFs that encounter failures. The operation log is used to prevent duplicated execution of operation, so that *at-most-once* execution semantics are guaranteed. On the other hand, re-execution for failed SSFs ensures *at-least-once* execution semantics.

Beldi's log-based fault-tolerant mechanism motivates Boki's shared log approach for stateful serverless computing. However, their techniques would need to be adapted for use with shared logs (§ 5.1), mostly because the workflow log is not co-located with user data in the same database.

9 Conclusion

State management has become a major challenge in serverless computing. Boki is the first system that allows stateful serverless functions to manage state using distributed shared logs. Boki's shared log abstraction (i.e., LogBooks) can support diverse serverless use cases, including fault-tolerant workflows, durable object storage, and message queues. Boki's shared logs achieve elasticity, data locality, and resource efficiency, enabled by a novel metalog design. The metalog is a unified solution to the problems of log ordering, consistency, and fault tolerance in Boki. Evaluations of Boki and its support libraries demonstrate the performance advantages (up to 4.7×) of the shared-log-based approach for serverless state management.

Acknowledgements. We thank our shepherd Jason Flinn and the anonymous reviewers for their insightful feedback. We also thank Cong Ding, Youer Pu, Zhiting Zhu, Yige Hu, Cheng Tan, Vijay Chidambaram, and Mark Silberstein for their valuable comments on the early draft of this work. This work is supported in part by NSF grants CNS-2008321 and NSF CNS-1900457, and the Texas Systems Research Consortium.

References

[1] [n.d.]. Amazon DynamoDB | NoSQL Key-Value Database | Amazon Web Services. https://aws.amazon.com/dynamodb/ [Accessed Jan, 2021].

[2] [n.d.]. Amazon SQS | Message Queuing Service | AWS. https://aws.amazon.com/sqs/ [Accessed Apr, 2021].

[3] [n.d.]. Apache Pulsar. https://pulsar.apache.org/ [Accessed Apr, 2021].

[4] [n.d.]. AWS Lambda – Serverless Compute - Amazon Web Servicesy. https://aws.amazon.com/lambda/ [Accessed Jan, 2021].

[5] [n.d.]. AWS Step Functions. https://aws.amazon.com/step-functions/ [Accessed Jan, 2021].

[6] [n.d.]. Azure Functions Serverless Compute | Microsoft Azure. https://azure.microsoft.com/en-us/services/functions/ [Accessed Jan, 2021].

[7] [n.d.]. CorfuDB. https://github.com/corfudb [Accessed Apr, 2021].

[8] [n.d.]. delimitrou/DeathStarBench: Open-source benchmark suite for cloud microservices. https://github.com/delimitrou/DeathStarBench [Accessed Jan, 2021].

[9] [n.d.]. Durable entities - Azure Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities [Accessed Jan, 2021].

[10] [n.d.]. Durable Functions Overview - Azure | Microsoft Docs. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp [Accessed Apr, 2021].

[11] [n.d.]. eniac/Beldi. https://github.com/eniac/Beldi [Accessed Apr, 2021].

[12] [n.d.]. RocksDB | A persistent key-value store | RocksDB. https://rocksdb.org/ [Accessed Apr, 2021].

[13] [n.d.]. The most popular database for modern apps | MongoDB. https://www.mongodb.com/ [Accessed Apr, 2021].

[14] [n.d.]. Tkrzw: a set of implementations of DBM. https://dbmx.net/tkrzw/ [Accessed Apr, 2021].

[15] [n.d.]. Tutorial: Design and implementation of a simple Twitter clone using PHP and the Redis key-value store. https://redis.io/topics/twitter-clone [Accessed Apr, 2021].

[16] [n.d.]. ut-osa/nightcore: Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. https://github.com/ut-osa/nightcore [Accessed Apr, 2021].

[17] [n.d.]. Workers Durable Objects Beta: A New Approach to Stateful Serverless. https://blog.cloudflare.com/introducing-workers-durable-objects/ [Accessed Jan, 2021].

[18] [n.d.]. Workflows | Google Cloud. https://cloud.google.com/workflows [Accessed Apr, 2021].

[19] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. 2015. Challenges to Adopting Stronger Consistency at Scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/ajoux

[20] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. 2020. Disaggregation and the Application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association. https://www.usenix.org/conference/hotcloud20/presentation/angel

[21] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/3267809.3267815

[22] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2020. Virtual Consensus in Delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 617–632. https://www.usenix.org/conference/osdi20/presentation/balakrishnan

[23] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 1–14. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/balakrishnan

[24] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 325–340. https://doi.org/10.1145/2517349.2522732

[25] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, Jingming Liu, Filip Gruszczynski, Jun Li, Rounak Tibrewal, Ali Zaveri, Rajeev Nagar, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2021. Log-structured Protocols in Delos. In *Proceedings of the 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA.

[26] Philip A. Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. 2015. Optimizing Optimistic Concurrency Control for Tree-Structured, Log-Structured Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1295–1309. https://doi.org/10.1145/2723372.2737788

[27] 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

[28] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Washington, DC, USA) *(PODC '17)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/3087801.3087802

[29] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[30] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. 2020. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 325–338. https://www.usenix.org/conference/nsdi20/presentation/ding

[31] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi

[32] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston,

MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[33] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[34] Pedro García-López, Aleksander Slominski, Simon Shillaker, Michael Behrendt, and Barnard Metzler. 2020. Serverless End Game: Disaggregation enabling Transparency. *arXiv preprint arXiv:2006.01251* (2020).

[35] S. Guo, R. Dhamankar, and L. Stewart. 2017. DistributedLog: A High Performance Replicated Log Service. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 1183–1194. https://doi.org/10.1109/ICDE.2017.163

[36] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

[37] 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.

[38] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701

[39] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[40] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing* (El Paso, Texas, USA) *(STOC '97)*. Association for Computing Machinery, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660

[41] Martin Kleppmann and Jay Kreps. 2015. Kafka, Samza and the Unix Philosophy of Distributed Data. *IEEE Data Eng. Bull.* 38, 4 (2015), 4–14. http://sites.computer.org/debull/A15dec/p4.pdf

[42] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[43] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. 2018. The FuzzyLog: A Partially Ordered Shared Log. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 357–372. https://www.usenix.org/conference/osdi18/presentation/lockerman

[44] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 453–468. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/mehdi

[45] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372. https://doi.org/10.1145/2517349.2517350

[46] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[47] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[48] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Commun. ACM* 64, 5 (April 2021), 76–84. https://doi.org/10.1145/3406011

[49] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319. https://doi.org/10.1145/98163.98167

[50] Srinath Setty, Chunzhi Su, Jacob R. Lorch, Lidong Zhou, Hao Chen, Parveen Patel, and Jinglei Ren. 2016. Realizing the Fault-Tolerance Promise of Cloud Storage Using Locks with Intent. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 501–516. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/setty

[51] 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

[52] 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

[53] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *ACM Comput. Surv.* 47, 3, Article 42 (Feb. 2015), 36 pages. https://doi.org/10.1145/2673577

[54] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

[55] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. 2017. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 35–49. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wei-michael

[56] 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

[57] Irene Zhang, Niel Lebeck, Pedro Fonseca, Brandon Holt, Raymond Cheng, Ariadna Norberg, Arvind Krishnamurthy, and Henry M. Levy. 2016. Diamond: Automating Data Management and Storage for Wide-Area, Reactive Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 723–738. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhang-irene

[58] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 263–278. https://doi.org/10.1145/2815400.2815404

[59] 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