Boki: Stateful Serverless Computing with Shared Logs

Zhipeng Jia  
*University of Texas at Austin*  

Emmett Witchel  
*University of Texas at Austin and Katana Graph*
Today’s Serverless Computing

For stateless functions, FaaS is

- Easy-to-use
- Highly elastic (1,000’s of concurrent functions)
Today’s Stateful Serverless Computing

Stateful functions

AWS Lambda

Amazon S3

Amazon DynamoDB

Amazon Simple Queue Service
State consistency with fault tolerance is difficult with current infrastructure!
Example: Conference Registration App

Serverless application

Function X
(create person)
Invoke with uid

Function Y
(append attendees)

Cloud Database

uid = read_inc(table=id, key="id")

write(table=profile, key="name"+uid, “Zhipeng Jia”)

append_list(table=conference, key=“SOSP21”, uid)

When failure happens, data stored in cloud database can be inconsistent
No easy way to detect and fix the inconsistency
Shared Logs: The Missing Piece in Serverless

The shared log as a write-ahead redo log for fault tolerance

Serverless application

Function X
(create person)

Invoke with $uid$

Function Y
(append attendees)

Cloud Database

$auid = \text{read\_inc}(\text{table}=id, \text{key}="id")$

$\text{write}(\text{table}=\text{profile}, \text{key}="\text{name}+uid\), "Zhipeng Jia")$

$\text{append\_list}(\text{table}=\text{conference}, \text{key}="\text{SOSP21}, uid")$

read\_inc write invoke append\_list a \text{shared log}$
Shared Logs: The Missing Piece in Serverless

The shared log for *state machine replication* (SMR)

How to maintain a shared state between concurrent functions with consistency?
Shared Logs: The Missing Piece in Serverless

The shared log for *state machine replication* (SMR)

Reconstruct the state machine by reading the log

State machine: \( \text{cmd}_Y + \text{cmd}_X + \text{cmd}_Z \)

Total order provided by the shared log is the source of consistency
Boki: FaaS + Shared Logs + Support Libraries

† Boki is the pronunciation of "簿記", meaning bookkeeping in Japanese
Boki: FaaS + Shared Logs + Support Libraries

- Serverless Functions
- Boki Support Libraries
  - BokiFlow
  - BokiStore
  - BokiQueue
- Boki’s LogBook API
- Boki Shared Logs
- Boki Runtime

FaaS Runtime (Nightcore [ASPLOS ‘21])
LogBook: Shared Log API for Serverless Functions

Log records have unique, monotonic sequence numbers (seqnum)

A LogBook: an append-only log
LogBook: Shared Log API for Serverless Functions

A LogBook: an append-only log

Log records can have an optional set of tags

Tag is used for selective reads, so that records with the same tag will form a logical sub-stream
Function Invocations $Share$ a LogBook

A LogBook: an append-only log

<table>
<thead>
<tr>
<th>seqnum</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tag set</td>
<td>{}</td>
<td>{a,b}</td>
<td>{b}</td>
<td>{a}</td>
<td>{b}</td>
<td>{}</td>
</tr>
</tbody>
</table>

Function $f$

Function $g$

Function $h$

Serverless application
Boki: FaaS + Shared Logs + Support Libraries
Challenges for Serverless Shared Logs

Serverless environment requires Boki to efficiently support *diverse* use patterns of shared logs

- **High Throughput**
  State-of-the-art shared log: 1M appends per second

- **Low Latency**
  Serverless environment disaggregates compute and storage
  Low-latency reads can only be achieved by caching, and Boki has to address read consistency

- **High Density**
  Many serverless applications have small resource use
  Boki has to efficiently support a high density of small LogBooks
Boki’s Techniques

- Multiplexing LogBooks on internal physical logs, with log indices for flexible log reads (achieve high density)
- Co-locating log indices and record caches with functions (achieve low latency)
- Metalog design that jointly addresses log ordering, read consistency, and fault tolerance (achieve high throughput, and bridge components together as a distributed system)
Boki’s Techniques

- Multiplexing LogBooks on internal physical logs, with log indices for flexible log reads
- Co-locating log indices and record caches with functions
- Metalog design that jointly addresses log ordering, read consistency, and fault tolerance
LogBooks are Multiplexed onto Internal Physical Logs

Boki’s internal physical logs are distributed shared logs with high-throughput

- Physical log 1
- Physical log 2
LogBooks are Multiplexed onto Internal Physical Logs

Every LogBook maps to a physical log

Appending to a LogBook
\[ \Rightarrow \] Appending to the associated physical log
LogBooks are Multiplexed onto Internal Physical Logs

Every LogBook maps to a physical log

Appending to a LogBook
⇒ Appending to the associated physical log

Boki is configured with a fixed number of physical logs, but can support a high density of LogBooks
Boki Physical Logs are Sharded

Each log shard is stored on 3 storage nodes

A sequencer orders log records across shards to form a totally ordered log

Boki uses Scalog [NSDI ‘20]’s high-throughput ordering protocol
LogBooks and Log Shards?

Initial idea: Assign every LogBook to some fixed log shard?

**Advantage:** Locating records for a LogBook is easy.

**Drawback:** Throughput of a LogBook will be limited to one shard!!
LogBooks and Log Shards?

We want records from a LogBook can go to any shards to enjoy the full throughput provided by the physical log.

Challenge: How to locate records for LogBook reads?
Building index for LogBook Reads

Boki’s log index groups records by \((book_id, tag)\)

\[
\text{logReadNext}(book_id = 3, \min\_seqnum = 8, \tag = 2)
\]

<table>
<thead>
<tr>
<th>Log index</th>
</tr>
</thead>
<tbody>
<tr>
<td>(book_id, tag)</td>
</tr>
<tr>
<td>seqnums</td>
</tr>
<tr>
<td>......</td>
</tr>
<tr>
<td>[......]</td>
</tr>
<tr>
<td>(3, 2)</td>
</tr>
<tr>
<td>[3, 6, 7, 9, 10, ...]</td>
</tr>
</tbody>
</table>
Building index for LogBook Reads

Boki’s log index groups records by \((book_id, tag)\)

Log index only includes metadata of log records (small per-record footprint), so that a single node can index an entire physical log

\[
\text{logReadNext}(book_id = 3, \ min\_seqnum = 8, \ tag = 2)
\]

<table>
<thead>
<tr>
<th>(book_id, tag)</th>
<th>seqnums</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\ldots)</td>
<td>[..]</td>
</tr>
<tr>
<td>((3, 2))</td>
<td>([3, 6, 7, 9, 10, \ldots])</td>
</tr>
</tbody>
</table>
Boki’s Techniques

- Multiplexing LogBooks on internal physical logs, with log indices for flexible log reads
- Co-locating log indices and record caches with functions
- Metalog design that jointly addresses log ordering, read consistency, and fault tolerance
Read Locality: Log Index and Cache on Function Nodes

LogBook engine process on each function node for handling LogBook API requests from functions
Read Locality: Log Index and Cache on Function Nodes

LogBook engine process on each function node for handling LogBook API requests from functions.

Log index is built and maintained by LogBook engines.
Read Locality: Log Index and Cache on Function Nodes

LogBook engine process on each function node for handling LogBook API requests from functions.

Log index is built and maintained by LogBook engines.

LogBook engines also cache log records, using records’ unique seqnums as cache keys.
Read Locality: Log Index and Cache on Function Nodes

LogBook engine process on each function node for handling LogBook API requests from functions.

Log index is built and maintained by LogBook engines.

LogBook engines also cache log records, using records’ unique seqnums as cache keys.

In the best case, LogBook reads can be served without leaving function node!
Read Consistency?

Every function node will maintain log indices for a subset of physical logs, but not all physical logs.

To allow maximum flexibility, we want to serve log reads from any index of the target physical log.

How to ensure read consistency given multiple copies of log indices?

The shared log abstraction requires strong read consistency, i.e., read-your-write and monotonic reads.
Boki’s Techniques

- Multiplexing LogBooks on internal physical logs, with log indices for flexible log reads
- Co-locating log indices and record caches with functions
- Metalog design that jointly addresses log ordering, read consistency, and fault tolerance
Boki’s *Metalog* Framework

**Sequencer** (log ordering)

*metalog*

Order log records across shards

**Storage nodes** (store log shards)

Scalogs’s protocol: periodically issues *cut vectors* to form a total order across log shards.
Boki’s *Metalog* Framework

Scalog’s protocol: periodically issues cut vectors to form a total order across log shards

Sequencer maintains a *metalog* that records issued cut vectors

Metalog is also replicated on 2 other sequencers for fault tolerance
Boki’s *Metalog* Framework

**Sequencer** (log ordering)

```
metalog
v_0  v_1  v_2  v_3
```

**Storage nodes** (store log shards)

```
A    B    C

D    E    F
```

**Function nodes** (maintain log indices)

```
log
v_0  v_1  v_2  v_3
```

Every cut vector in the metalog adds a new batch of records to the physical log.

LogBook engines (on function nodes) subscribe to the metalog to incrementally build log index.
Metalog as the Mechanism for **Read Consistency**

Metalog positions (cuts in the physical log)
Metalog as the Mechanism for *Read Consistency*

Index replicas make progress independently, so that they are actually *inconsistent*
Metalog as the Mechanism for Read Consistency

Boki maintains metalog positions for log readers (i.e., functions), indicating their observed state.
Metalog as the Mechanism for *Read Consistency*

Boki performs consistency check using metalog positions, and retry the read later if the check fails.

- **Fn f**: A 3 wants to read via A
- **Consistency check fails**: Violate *monotonic read*
Metalog as the Mechanism for **Read Consistency**

Metalog position is also updated on appending a new record, to ensure *read-your-write*
Metalog as the Mechanism for **Fault Tolerance**

Control plane
(failure detection and reconfiguration)

Step 1: node failure detected, and initiates reconfiguration protocol

Sequencer (primary)

Sequencer (backup)

Sequencer (backup)

Storage nodes
(store log shards)

Function nodes
(maintain log indices)
Metalog as the Mechanism for **Fault Tolerance**

**Control plane** (failure detection and reconfiguration)

- **Sequencer** (primary)
- **Sequencer** (backup)
- **Sequencer** (backup)

**Storage nodes** (store log shards)

**Function nodes** (maintain log indices)

The metalog controls the global progress of a physical log. Stopping the progress of the metalog will stop the progress of the entire system.
Metalog as the Mechanism for **Fault Tolerance**

**Control plane** (failure detection and reconfiguration)

- **Sequencer** (primary and backup)
- **Storage nodes** (store log shards)
- **Function nodes** (maintain log indices)

---

**Step 2: seal the metalog with all sequencers**

Boki uses Delos [OSDI ‘20]’s fault-tolerant log sealing protocol
Metalog as the Mechanism for **Fault Tolerance**

**Control plane**
(failure detection and reconfiguration)

Step 3: metalog is successfully sealed
Metalog as the Mechanism for *Fault Tolerance*

Control plane
(failure detection and reconfiguration)

Step 3: metalog is successfully sealed

Sealed metalog

Storage nodes
(store log shards)

Function nodes
(maintain log indices)

Sequencer (primary)
Sequencer (backup)
Sequencer (backup)

All frozen as metalog is sealed
Metalog as the Mechanism for **Fault Tolerance**

- **Control plane** (failure detection and reconfiguration)
- **Sealed metalog** $v_0, v_1, v_2, v_3$
- **Sequencer** (primary)
- **Sequencer** (backup)
- **Function nodes** (maintain log indices)
- **Storage nodes** (store log shards)

**Step 4: set up a new configuration**

All frozen as metalog is sealed
Metalog as the Mechanism for *Fault Tolerance*

**Control plane**
(failure detection and reconfiguration)

**Step 4: set up a new configuration**

In the new configuration, a new metalog is used.
Ease the Usage of LogBooks for Serverless Functions
Boki Support Libraries

**BokiFlow**: Fault-Tolerant Serverless Workflows  
- Support workflows composing multiple stateful functions  
- Provide strong end-to-end guarantees (i.e., exactly-once execution semantics)  
- Based on Beldi [OSDI ’20]’s log-based protocol for fault tolerance

**BokiStore**: Transactional Object Store  
- Provide durable JSON objects to serverless functions  
- Strong consistency (sequentially consistent) and transaction support  
- Based on Tango [SOSP ‘13]’s techniques

**BokiQueue**: High-Throughput Message Queues  
- Enable message passing and coordination between serverless functions  
- Use vCorfu [NSDI ‘17]’s CSMR technique for scalability
Evaluation: Experiment Setup

- Test on AWS with EC2 instances
  c5d.2xlarge VMs, each has 8 vCPU, 16GB DRAM, 200GB SSD, 10Gb NIC

- 3 storage nodes for each log shard
  3 sequencer nodes for a metalog
Evaluation: Microbenchmark of LogBook Operations

**Log append**

<table>
<thead>
<tr>
<th>Concurrent writers</th>
<th>320</th>
<th>640</th>
<th>1280</th>
<th>2560</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput (KOp/s)</td>
<td>130.8</td>
<td>279.2</td>
<td>604.4</td>
<td>1,159</td>
</tr>
</tbody>
</table>

Throughput scales to 1.2M appends per second (p50 latency 2.03ms, p99 latency 6.42ms)

**Log read**

<table>
<thead>
<tr>
<th></th>
<th>Local index (cache hit)</th>
<th>Local index (cache miss)</th>
<th>Remote index</th>
</tr>
</thead>
<tbody>
<tr>
<td>50% latency</td>
<td>0.12ms</td>
<td>0.57ms</td>
<td>0.79ms</td>
</tr>
<tr>
<td>99% latency</td>
<td>0.72ms</td>
<td>1.48ms</td>
<td>2.90ms</td>
</tr>
</tbody>
</table>

Read latency is 121μs in the best case
Evaluation: Serverless Workflows
Comparing BokiFlow with Beldi

Unsafe baseline
No mechanism for fault tolerance, so that state can be inconsistent under workflow failures.

Beldi [OSDI ‘20]
Log-based protocols for fault tolerance. Beldi builds logging layer over DynamoDB.

BokiFlow
Adapt Beldi’s techniques to work with Boki’s LogBooks.

Microbenchmarks of primitive operations
(main bars show 50% latencies, error bars show 99% latencies)

Boki provides a more performant logging layer to support Beldi’s fault-tolerance mechanisms
Evaluation: Serverless Workflows
Comparing BokiFlow with Beldi

**Movie review workload**
- 4.7x lower latency
- 54x lower 99% latency

**Travel reservation workload**
- 4.3x lower latency
- 26x lower 99% latency
Evaluation: Object Store
Comparing BokiStore with MongoDB on Retwis (a Twitter clone) workload

Throughput
- Up to 25% higher throughput

Latency by request types
- Executing transactions up to 2.3x faster
Evaluation: Message Queues
Comparing BokiQueue with Amazon SQS and Apache Pulsar

Message throughput

<table>
<thead>
<tr>
<th></th>
<th>Amazon SQS</th>
<th>Apache Pulsar</th>
<th>BokiQueue</th>
</tr>
</thead>
<tbody>
<tr>
<td>64P/256C</td>
<td>6.08ms</td>
<td>7.39ms</td>
<td>3.70ms</td>
</tr>
<tr>
<td>256P/64C</td>
<td>99.8ms</td>
<td>7.81ms</td>
<td>6.61ms</td>
</tr>
<tr>
<td>256P/256C</td>
<td>12.1ms</td>
<td>8.21ms</td>
<td>7.96ms</td>
</tr>
</tbody>
</table>

66% – 114% higher w.r.t Amazon SQS
6% – 23% higher w.r.t Apache Pulsar

Also lower latencies compared to other systems
Conclusion

- Boki justifies the value of shared logs in stateful serverless, where shared logs can provide mechanisms for consistency and fault tolerance.
- Boki proposes novel shared log techniques to address unique challenges introduced by the serverless environment.
- Boki support libraries demonstrate how shared logs can support 3 different serverless use cases, and evaluation of these libraries shows Boki can speed up important workloads by up to 4.7x.

Boki is open source at github.com/ut-osa/boki
Thank you!