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

Invoke cloud functions

```
def my_funcion(a, b):
    return a * b
```

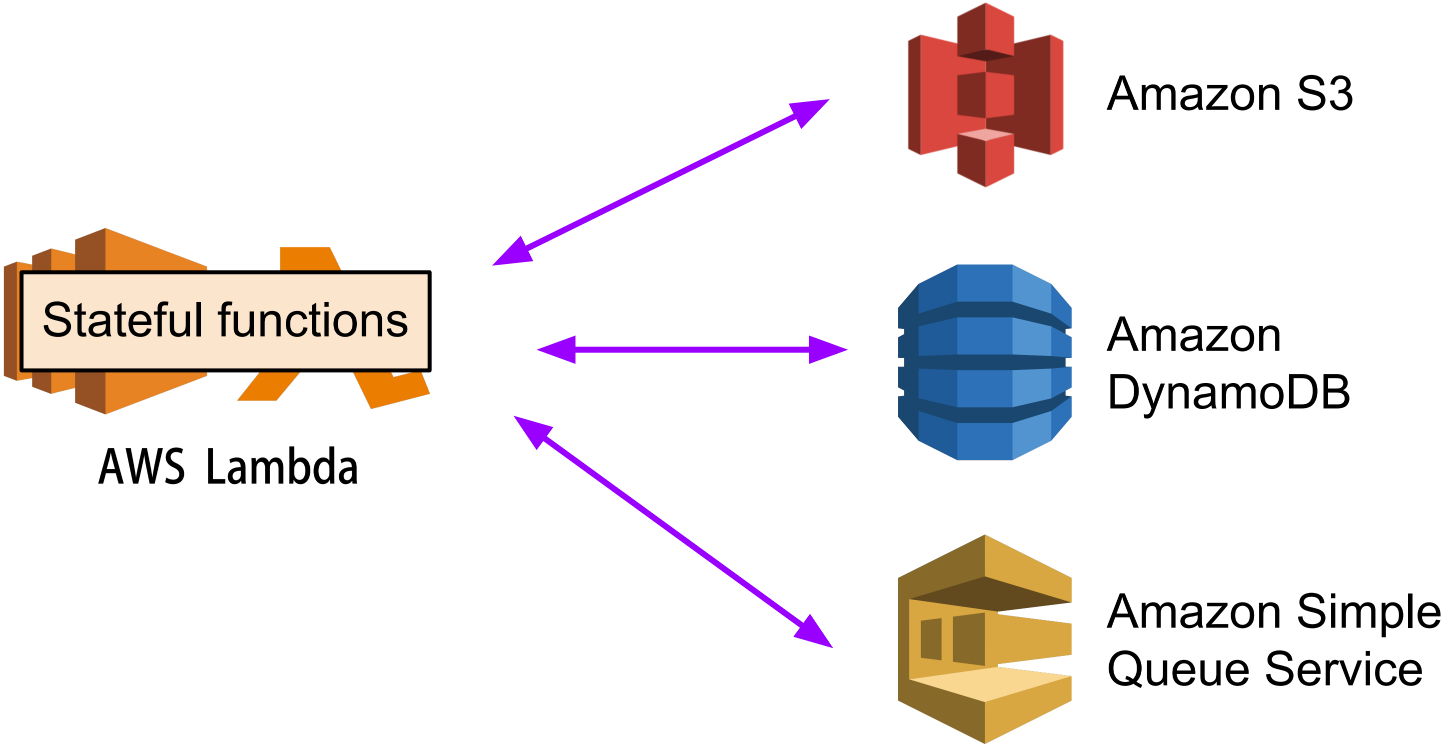
Function-as-a-service (FaaS)

AWS Lambda

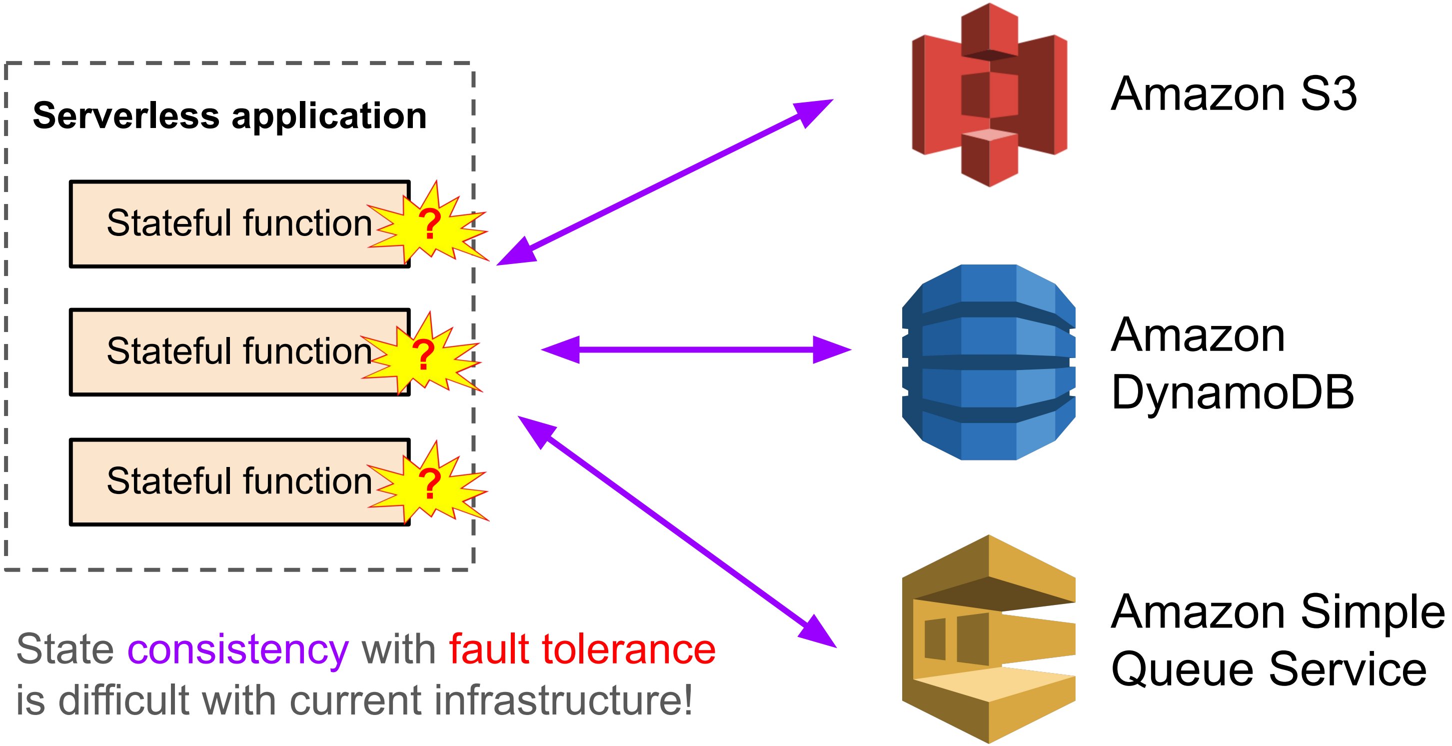
For stateless functions, FaaS is

- Easy-to-use
- Highly elastic (1,000's of concurrent functions)

Today’s Stateful Serverless Computing

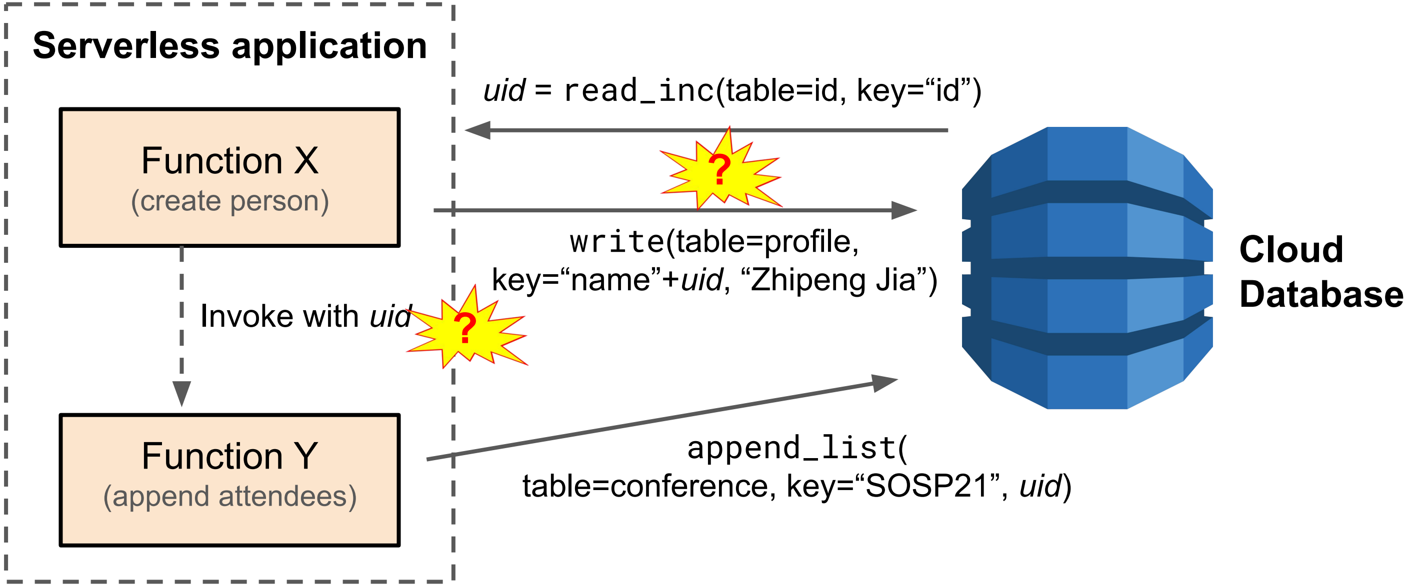


Today’s Stateful Serverless Computing



State consistency with fault tolerance is difficult with current infrastructure!

Example: Conference Registration App

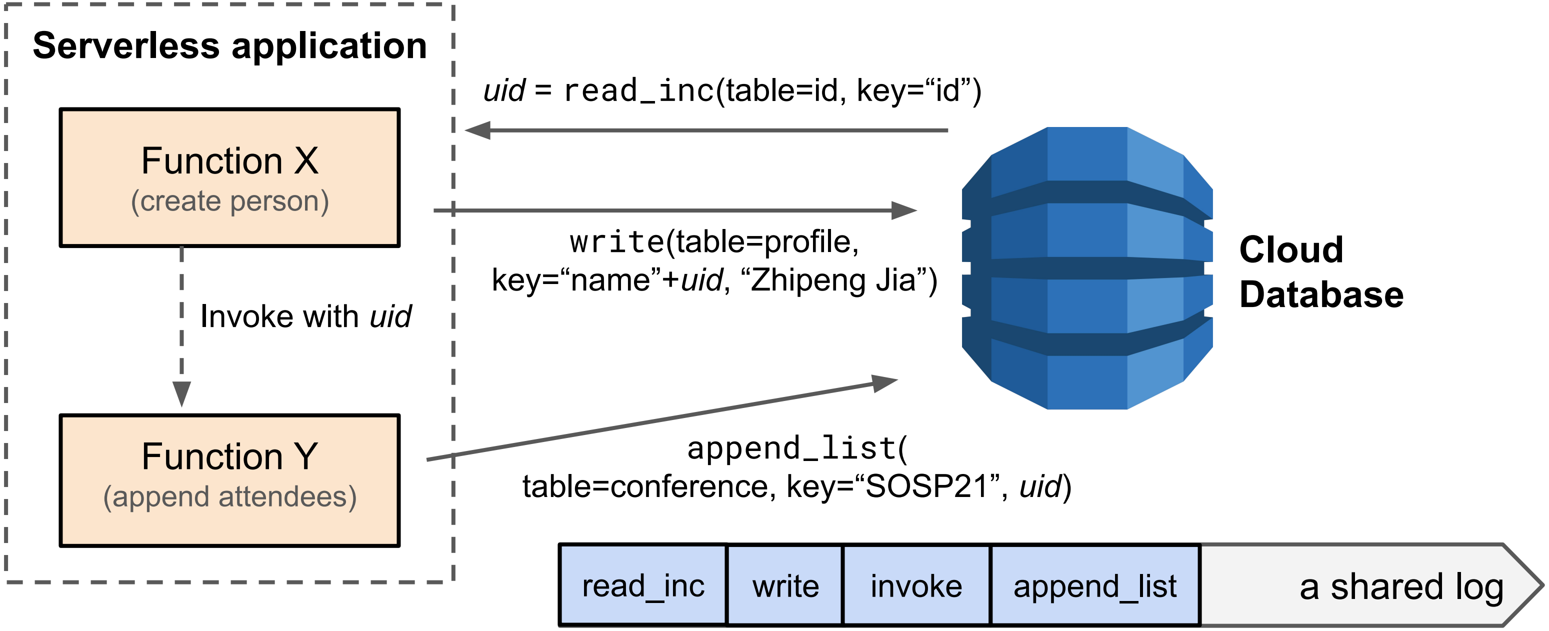


When failure happens, data stored in cloud database can be *inconsistent*

No easy way to detect and fix the inconsistency

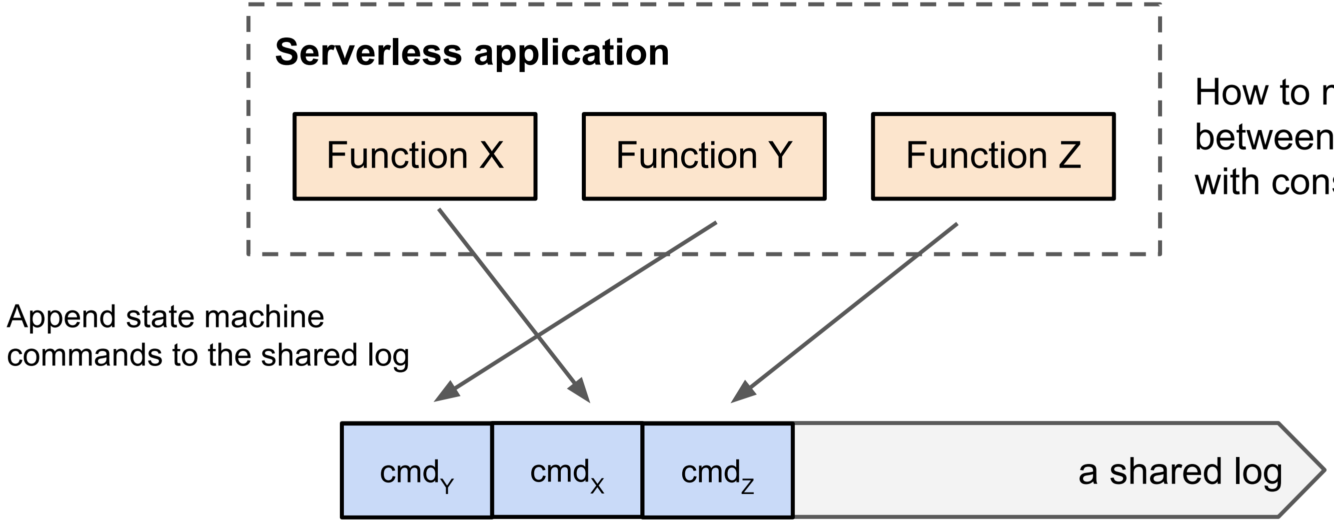
Shared Logs: The Missing Piece in Serverless

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



Shared Logs: The Missing Piece in Serverless

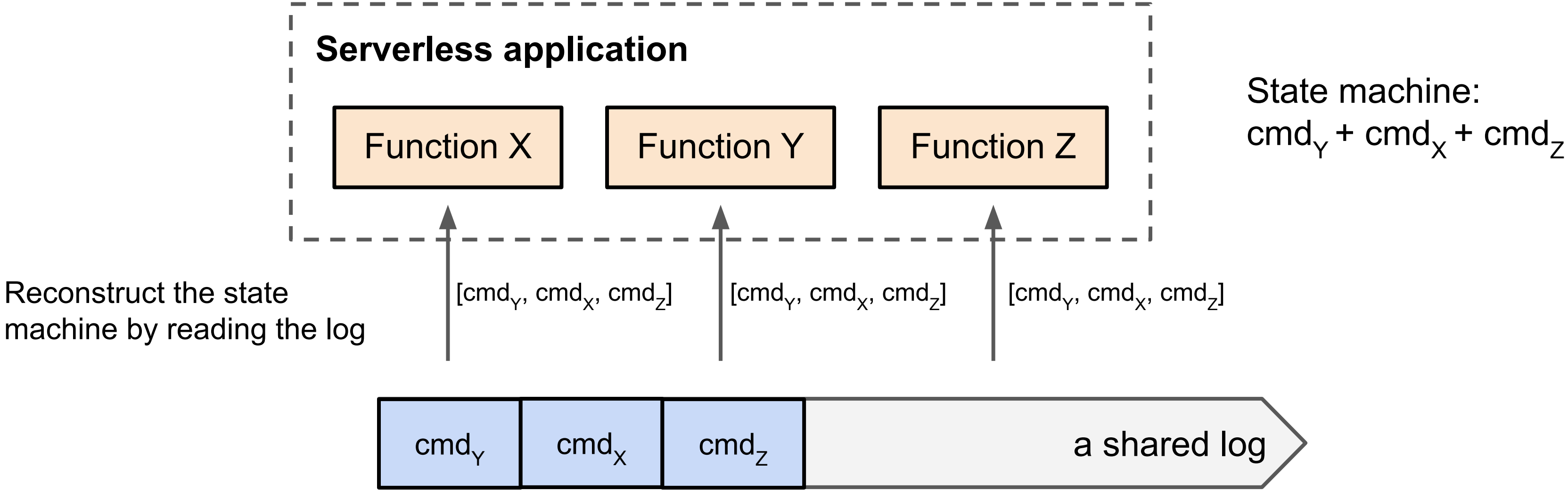
The shared log for *state machine replication* (SMR)



How to maintain a shared state between concurrent functions with consistency?

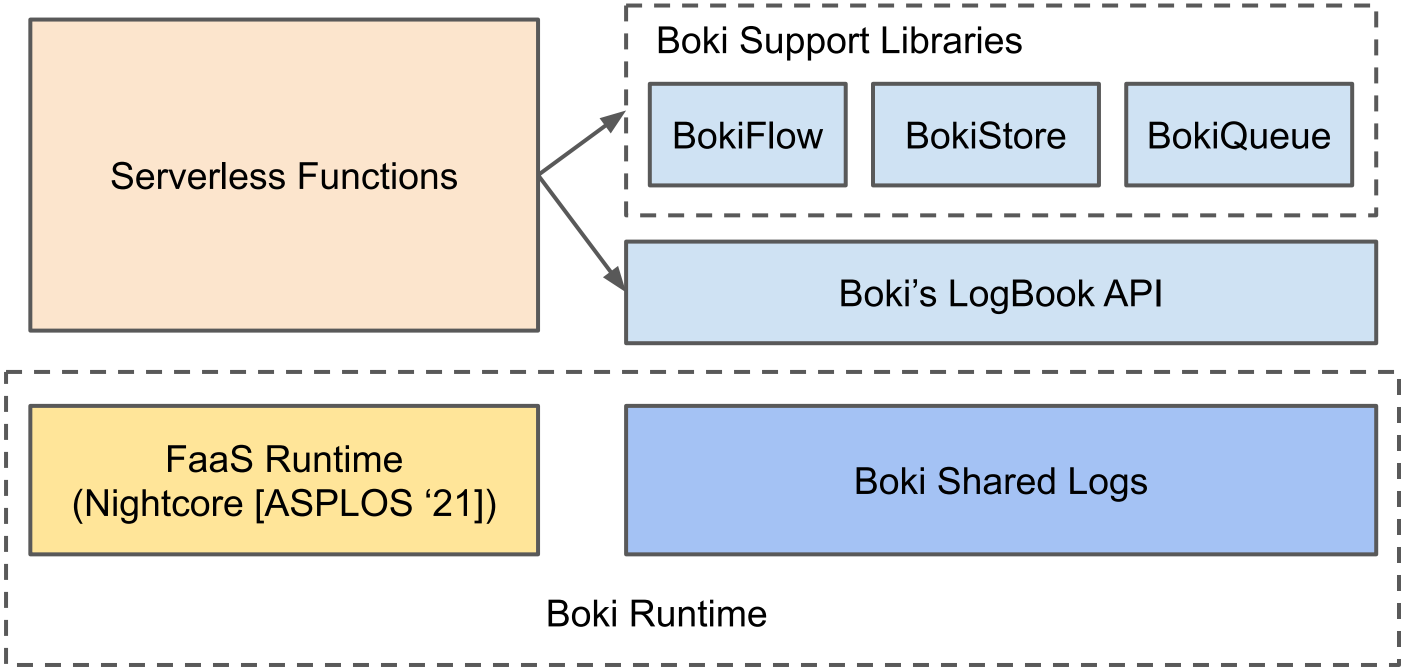
Shared Logs: The Missing Piece in Serverless

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



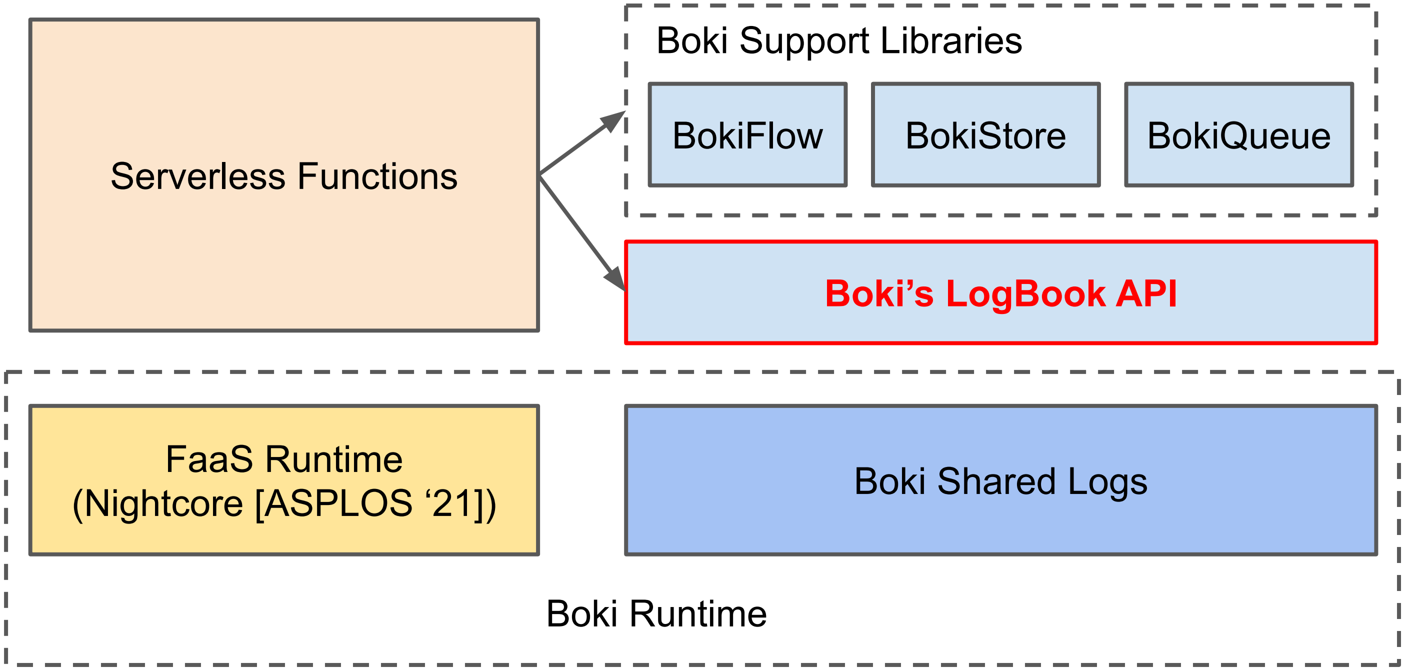
Total order provided by the shared log is the source of *consistency*

Boki: FaaS + Shared Logs + Support Libraries

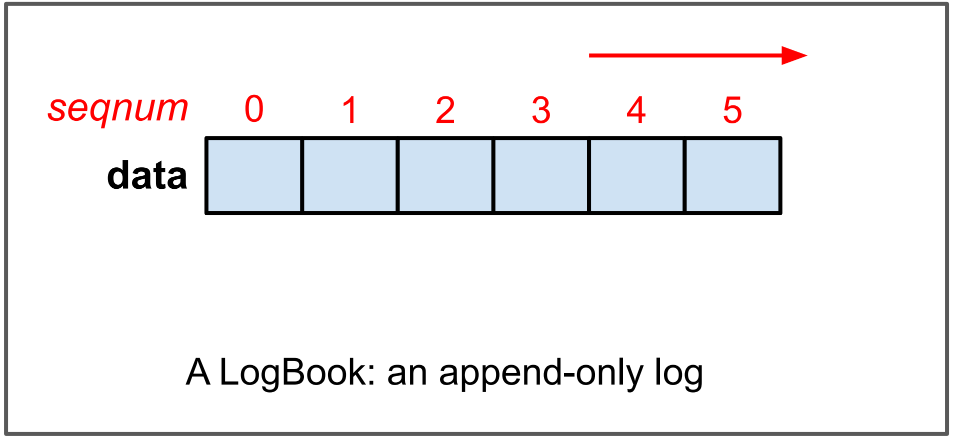


† Boki is the pronunciation of "簿記", meaning *bookkeeping* in Japanese

Boki: FaaS + Shared Logs + Support Libraries

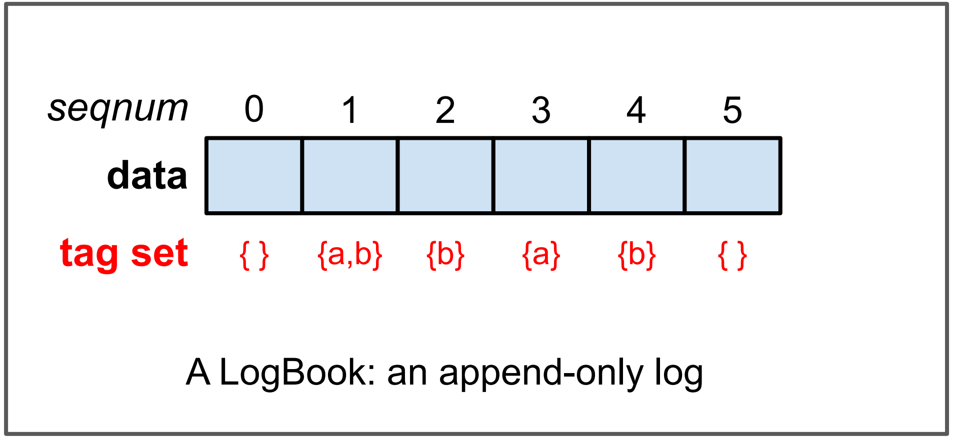


LogBook: Shared Log API for Serverless Functions



Log records have *unique*, *monotonic* sequence numbers (seqnum)

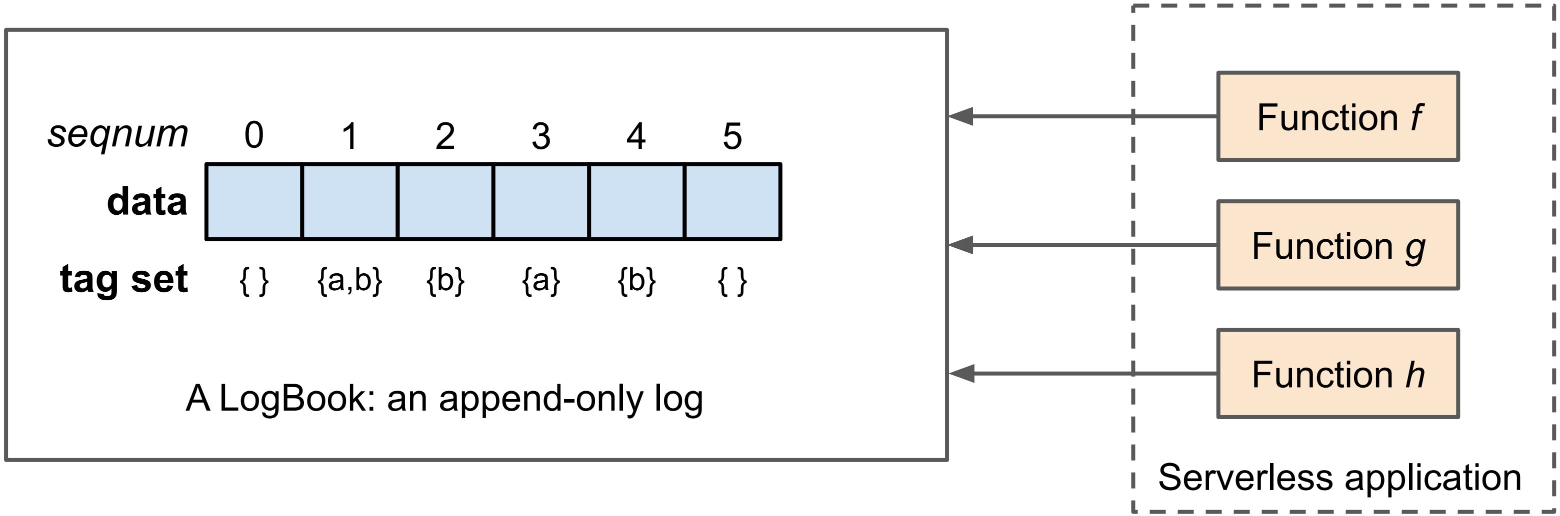
LogBook: Shared Log API for Serverless Functions



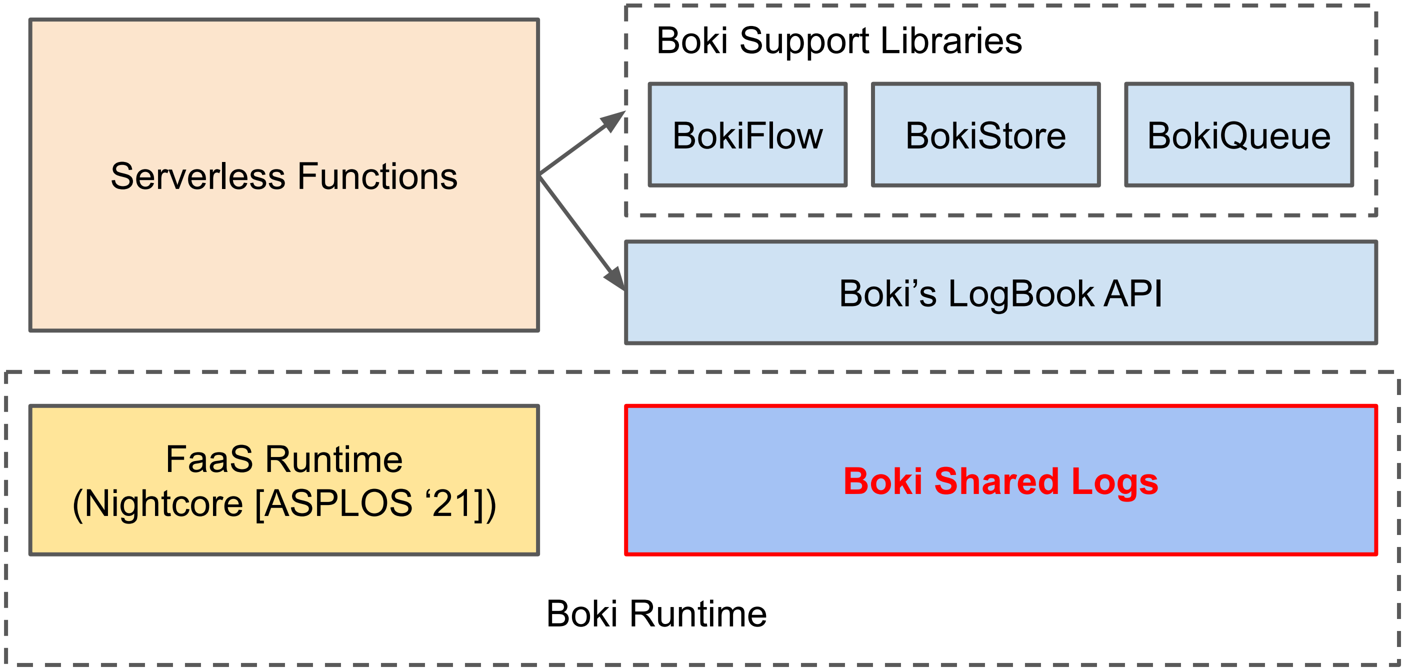
Log records can have an optional set of *tags*

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

Function Invocations *Share* a LogBook



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

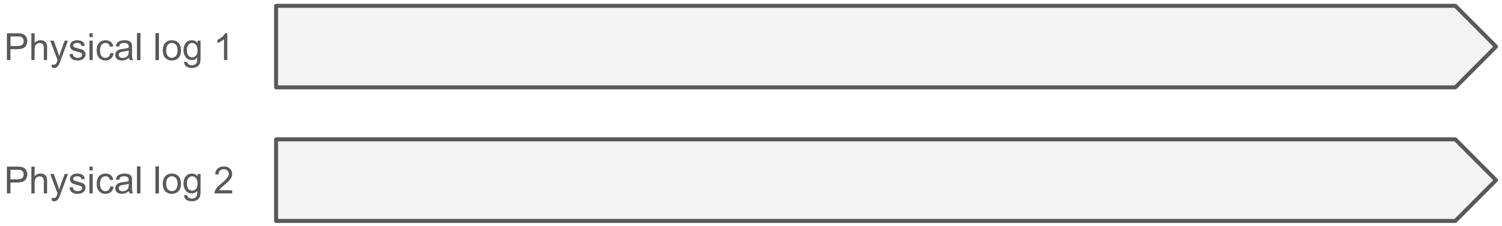
- Multiplexing LogBooks on internal physical logs, with log indices for flexible log reads (achieve high density)
- Co-locating log indices and record caches with functions (achieve low latency)
- Metalog design that jointly addresses *log ordering*, *read consistency*, and *fault tolerance* (achieve 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



LogBooks are Multiplexed onto Internal Physical Logs

LogBook A

LogBook B

LogBook C

LogBook D

LogBook E

Every LogBook maps to a physical log

Appending to a LogBook
⇒ Appending to the associated physical log

Physical log 1

Physical log 2

LogBooks are Multiplexed onto Internal Physical Logs

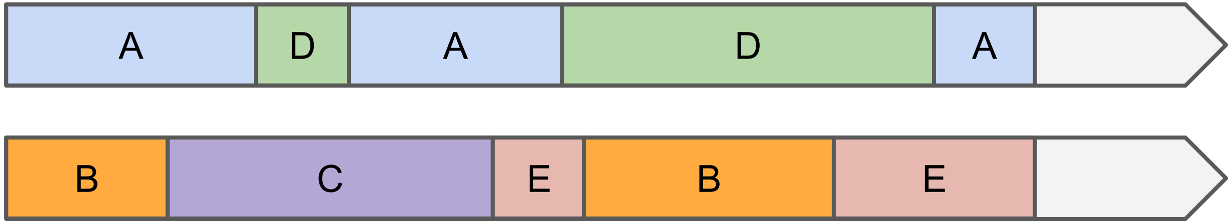
LogBook A | LogBook B | LogBook C | LogBook D | LogBook E

Every LogBook maps to a physical log

Appending to a LogBook
⇒ Appending to the associated physical log

Physical log 1
(LogBooks A, D)

Physical log 2
(LogBooks B, C, E)



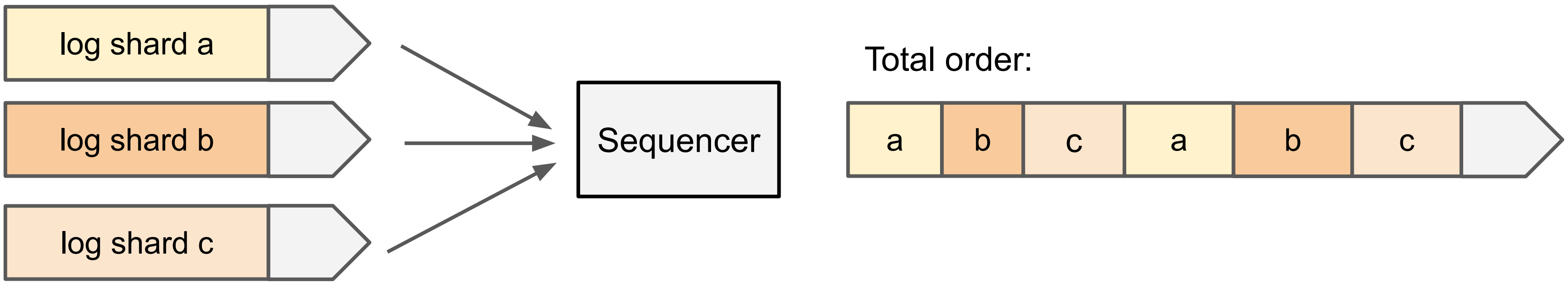
Boki is configured with a **fixed** number of physical logs,
but can support a high density of LogBooks

Boki Physical Logs are Sharded

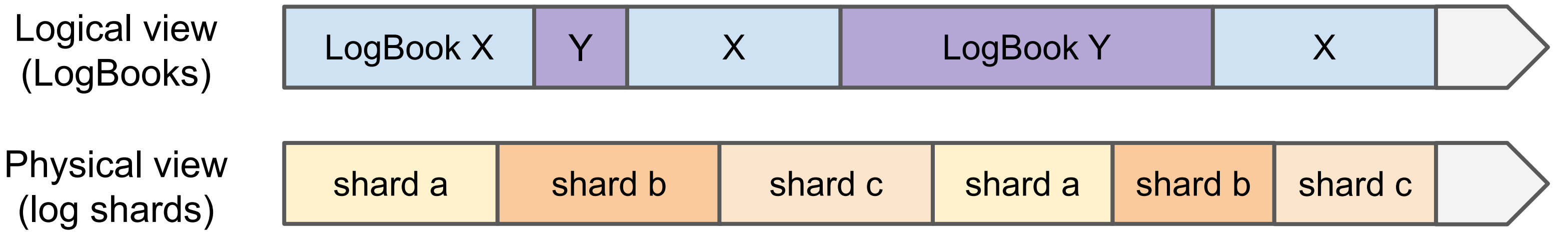
Each log shard is stored on 3 storage nodes

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?

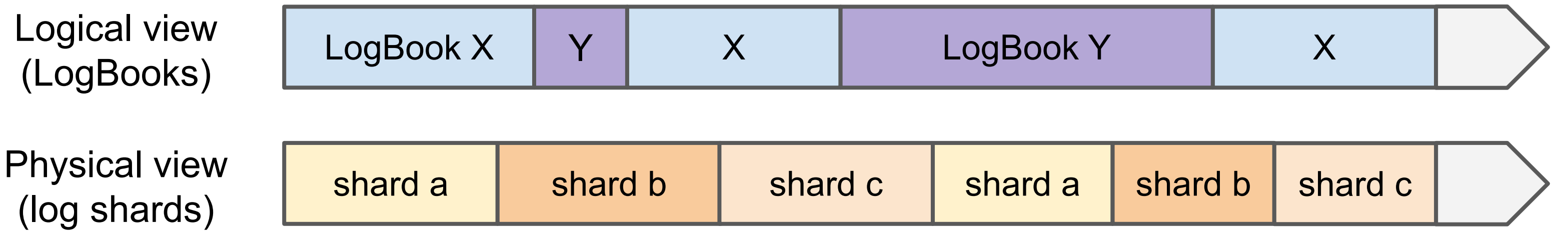


Initial idea: Assign every LogBook to some fixed log shard?

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*)

`logReadNext`(*book_id* = 3, *min_seqnum* = 8, *tag* = 2)

Log index	
(book_id, tag)	seqnums
……	[……]
(3, 2)	[3, 6, 7, 9, 10, …]

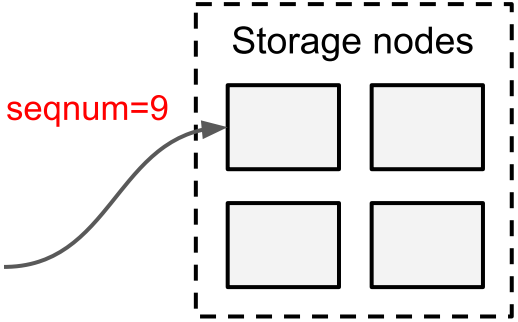
Building index for LogBook Reads

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

logReadNext(*book_id* = 3, *min_seqnum* = 8, *tag* = 2)

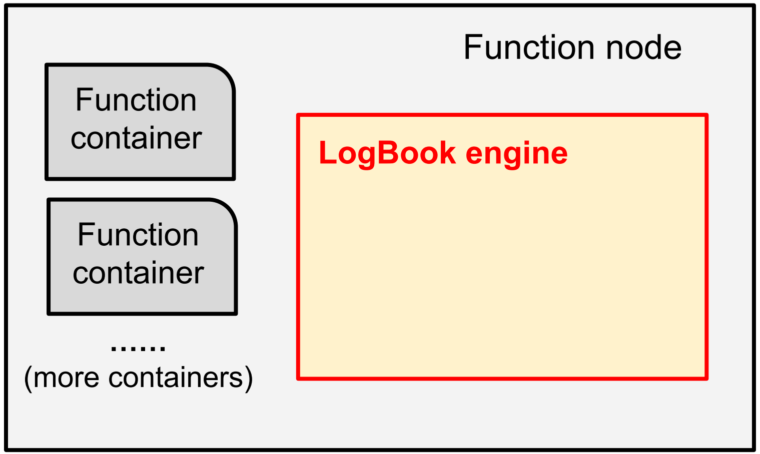
Log index	
(book_id, tag)	seqnums
……	[……]
(3, 2)	[3, 6, 7, **9**, 10, …]



Boki's Techniques

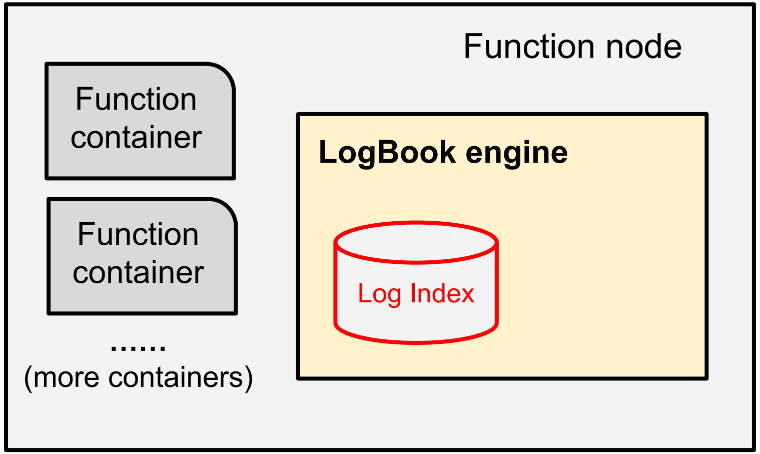
- Multiplexing LogBooks on internal physical logs, with log indices for flexible log reads
- Co-locating log indices and record caches with functions
- Metalog design that jointly addresses *log ordering*, *read consistency*, and *fault tolerance*

Read Locality: Log Index and Cache on Function Nodes



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

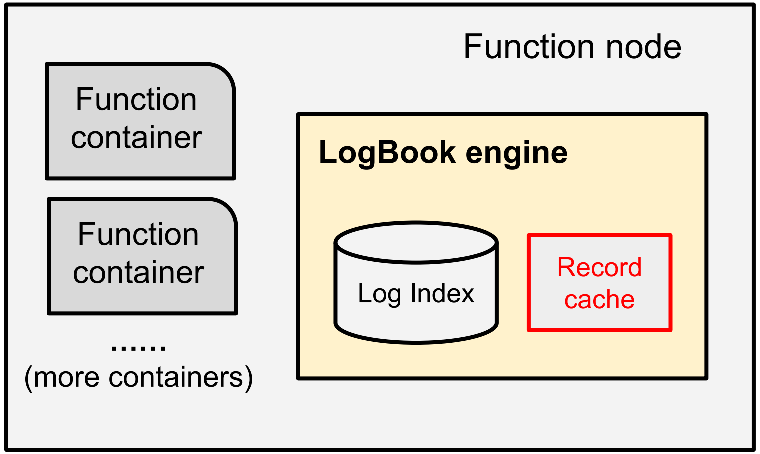
Read Locality: Log Index and Cache on Function Nodes



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

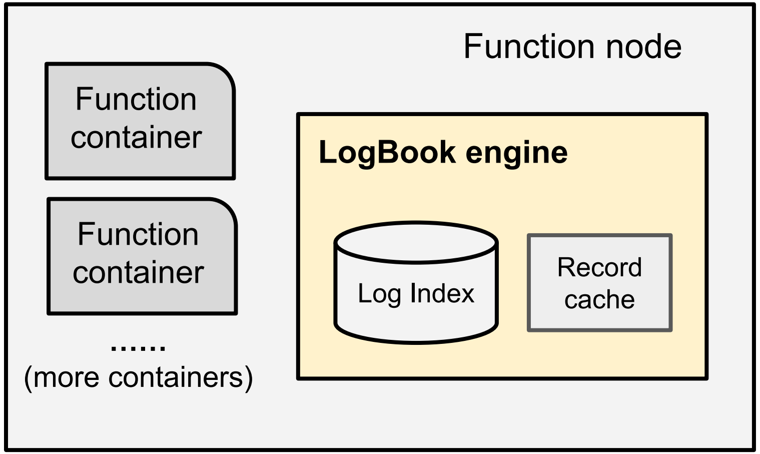


LogBook engine process on each function node for handling LogBook API requests from functions

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

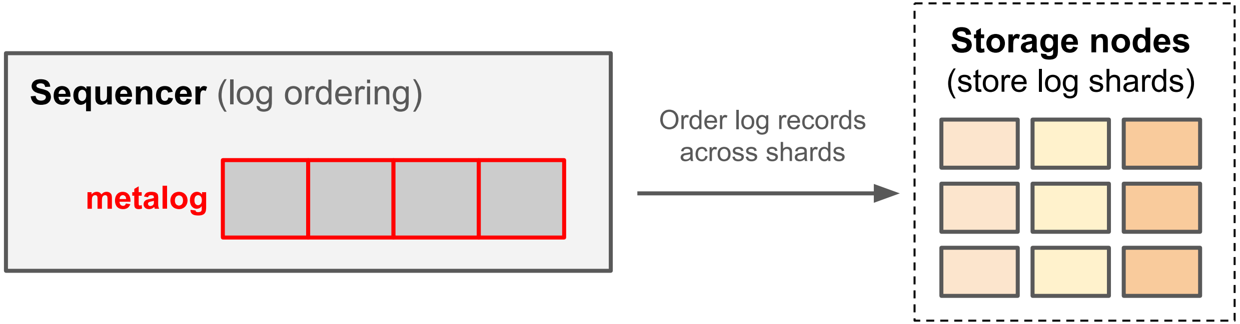
How to ensure read consistency given multiple copies of log indices?

The shared log abstraction requires strong read consistency, i.e., *read-your-write* and *monotonic reads*

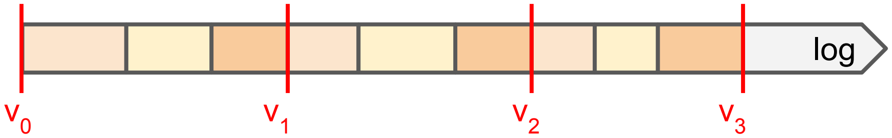
Boki’s Techniques

- 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*

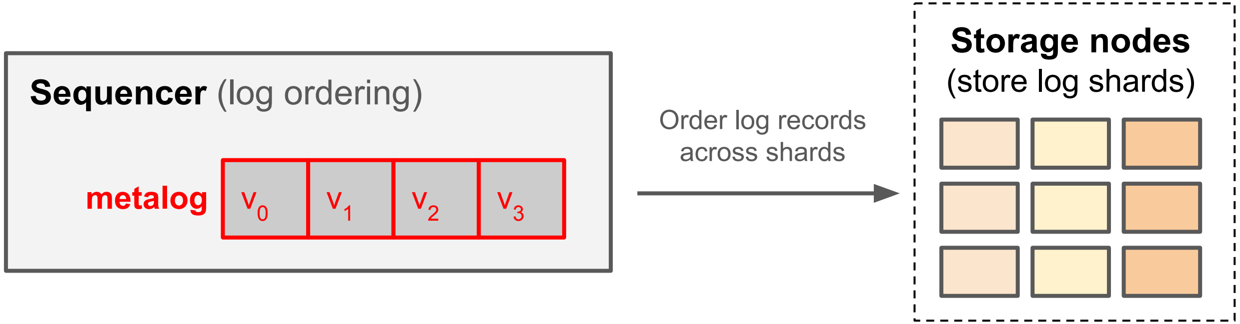
Boki's *Metalog* Framework



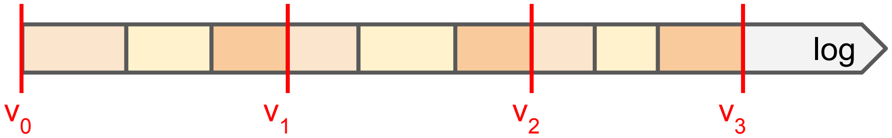
Scalog's protocol: periodically issues cut vectors to form a total order across log shards



Boki's *Metalog* Framework



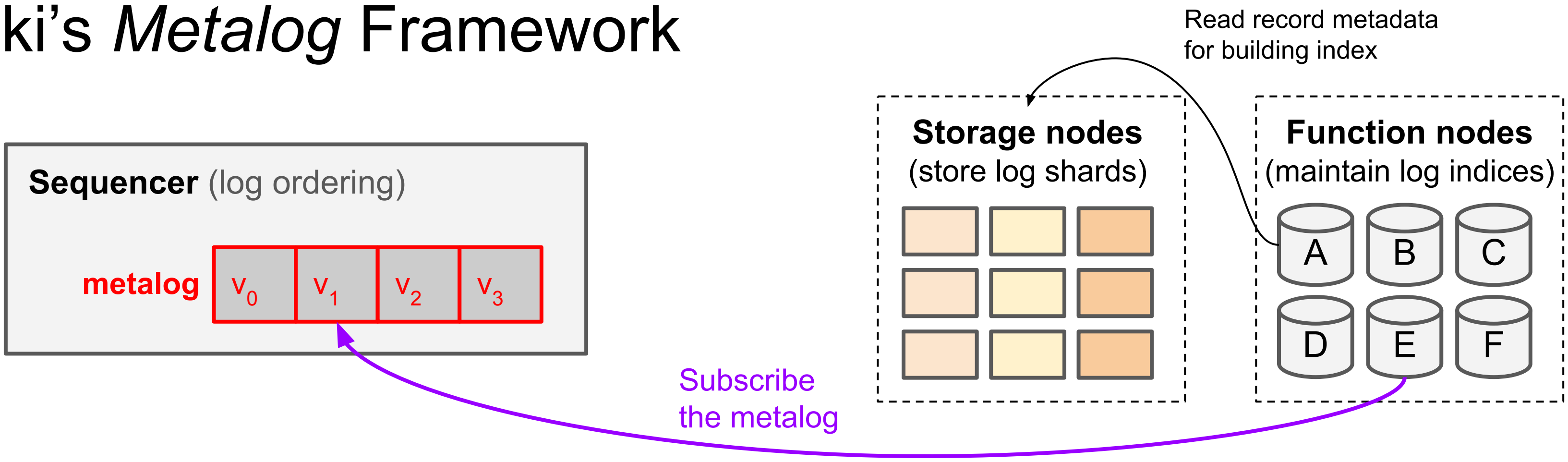
Scalog's protocol: periodically issues cut vectors to form a total order across log shards



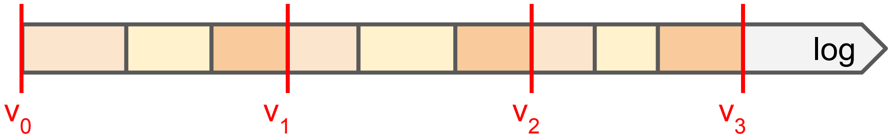
Sequencer maintains a *metalog* that records issued cut vectors

Metalog is also replicated on 2 other sequencers for fault tolerance

Boki's *Metalog* Framework

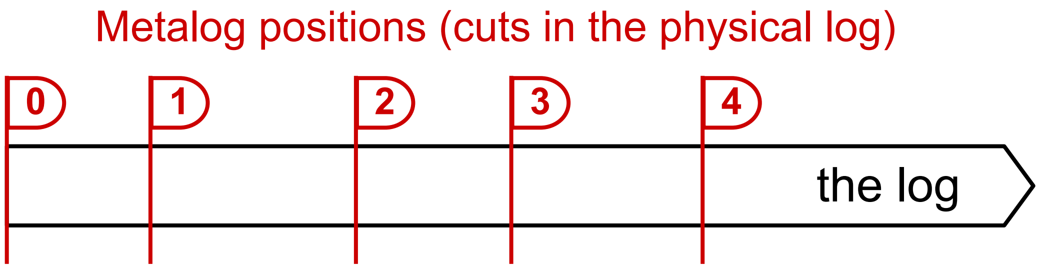


Every cut vector in the metalog adds a new batch of records to the physical log

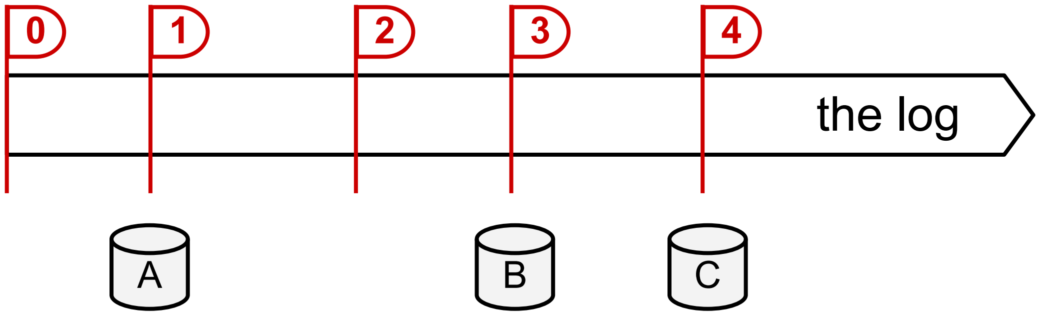


LogBook engines (on function nodes) subscribe to the metalog to incrementally build log index

Metalog as the Mechanism for ***Read Consistency***

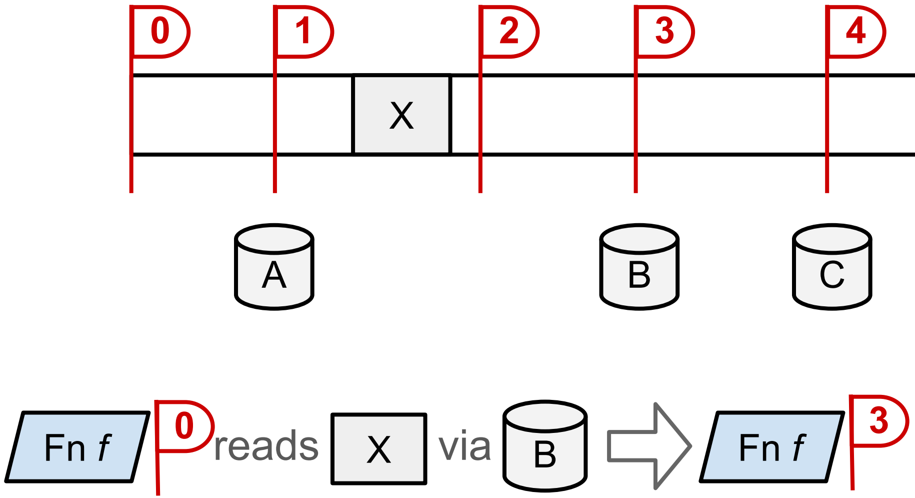


Metalog as the Mechanism for ***Read Consistency***



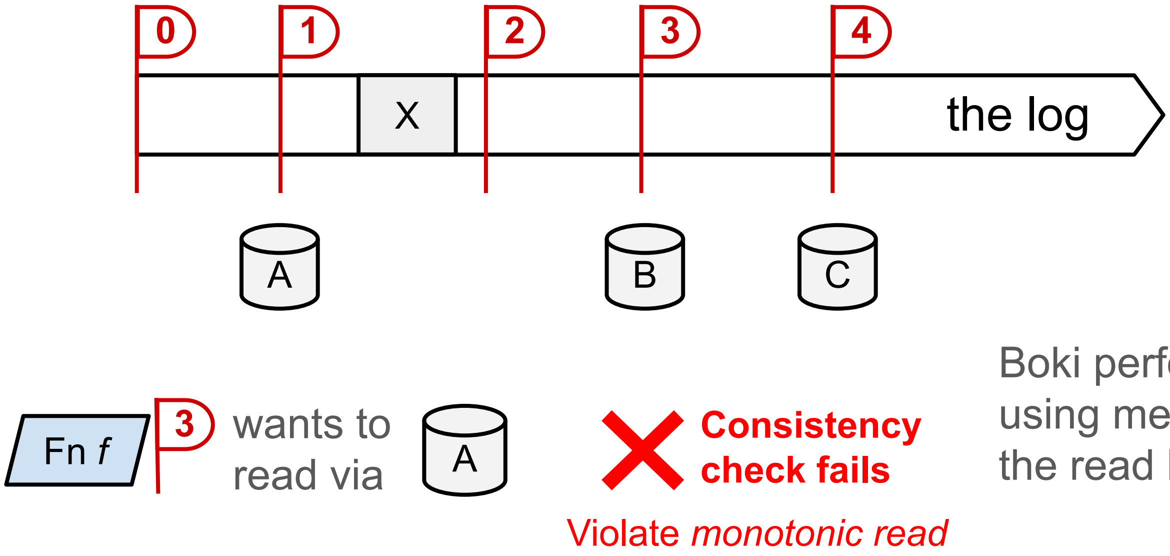
Index replicas make progress independently,
so that they are actually ***inconsistent***

Metalog as the Mechanism for ***Read Consistency***



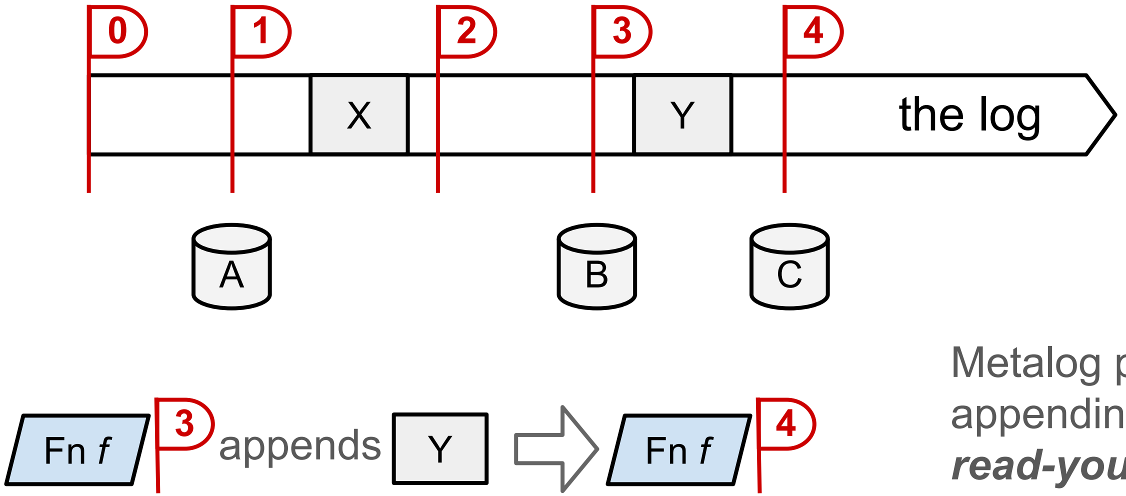
Boki maintains metalog positions for log readers (i.e., functions), indicating their observed state

Metalog as the Mechanism for ***Read Consistency***



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

Metalog as the Mechanism for ***Read Consistency***

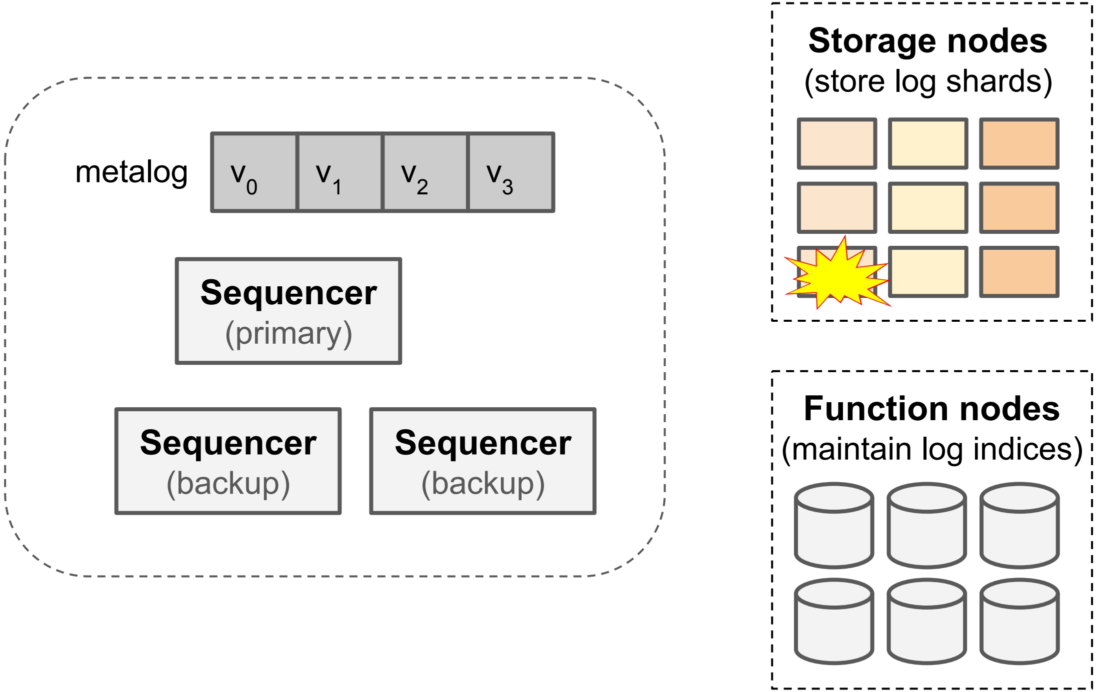


Metalog position is also updated on appending a new record, to ensure ***read-your-write***

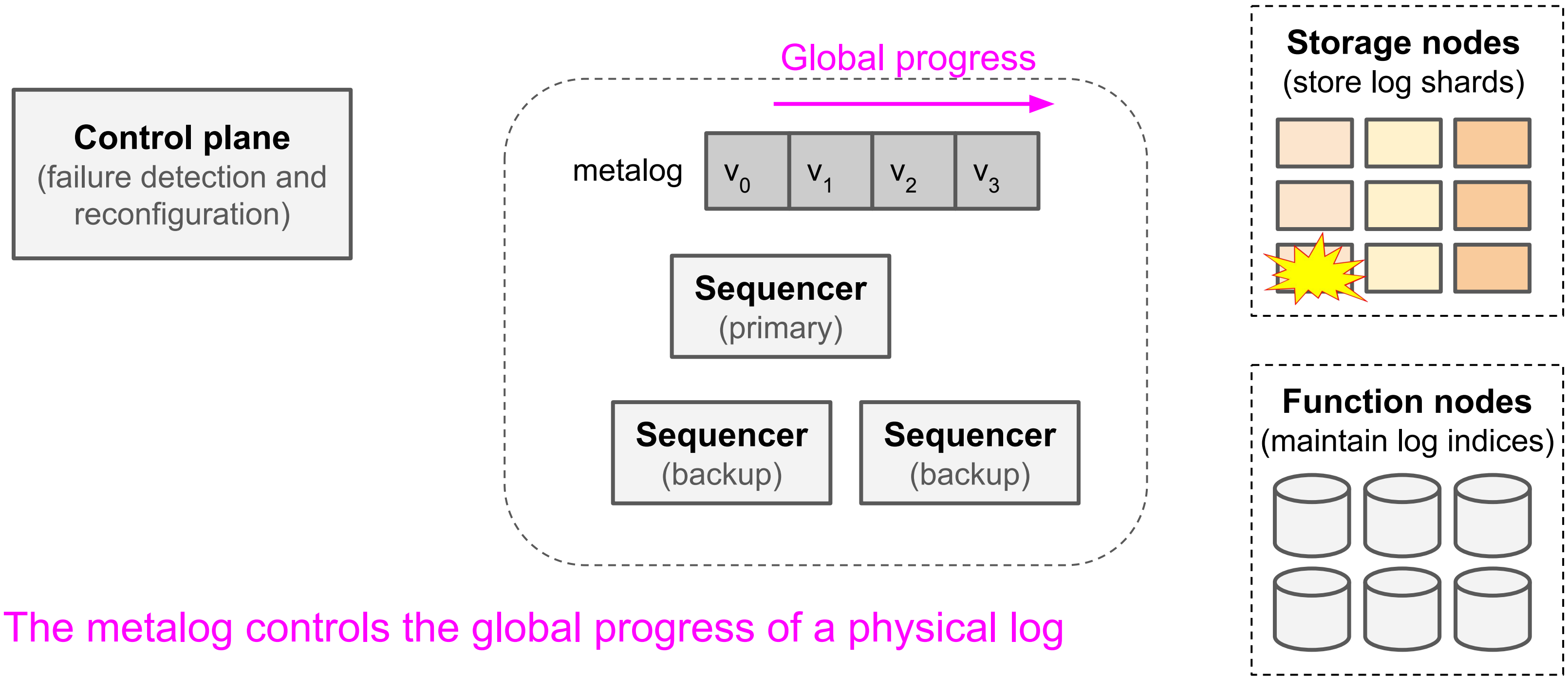
Metalog as the Mechanism for ***Fault Tolerance***

Control plane
(failure detection and reconfiguration)

Step 1: node failure detected, and initiates reconfiguration protocol



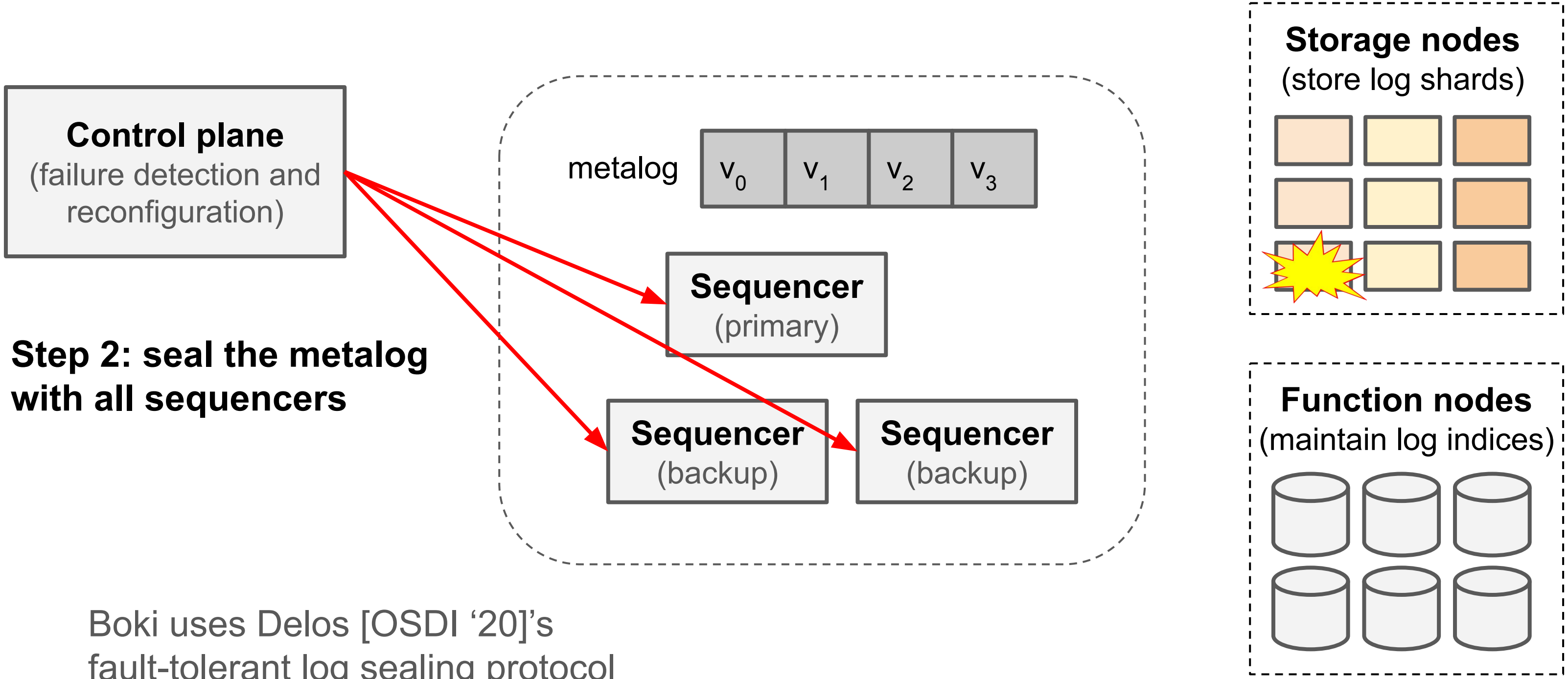
Metalog as the Mechanism for ***Fault Tolerance***



The metalog controls the global progress of a physical log

Stopping the progress of the metalog will stop the progress of the entire system

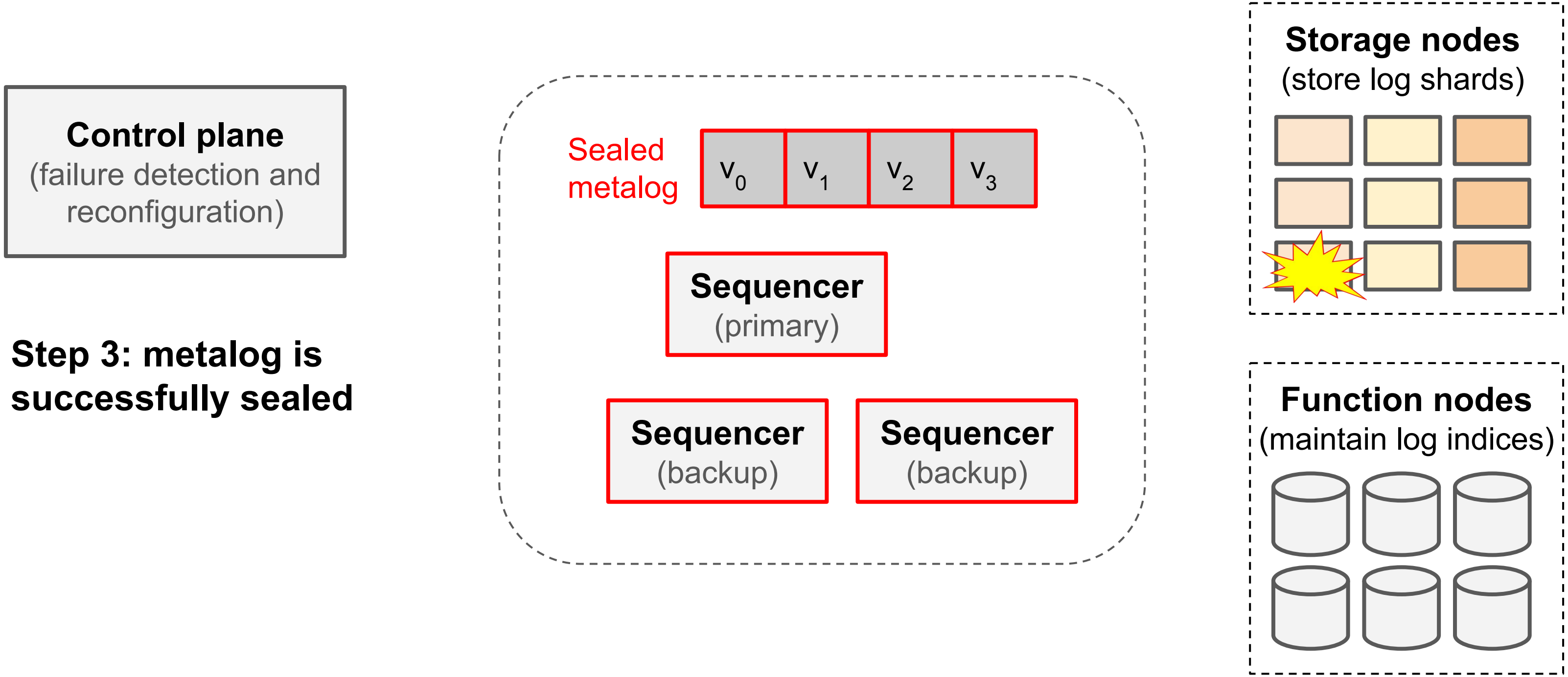
Metalog as the Mechanism for ***Fault Tolerance***



Step 2: seal the metalog with all sequencers

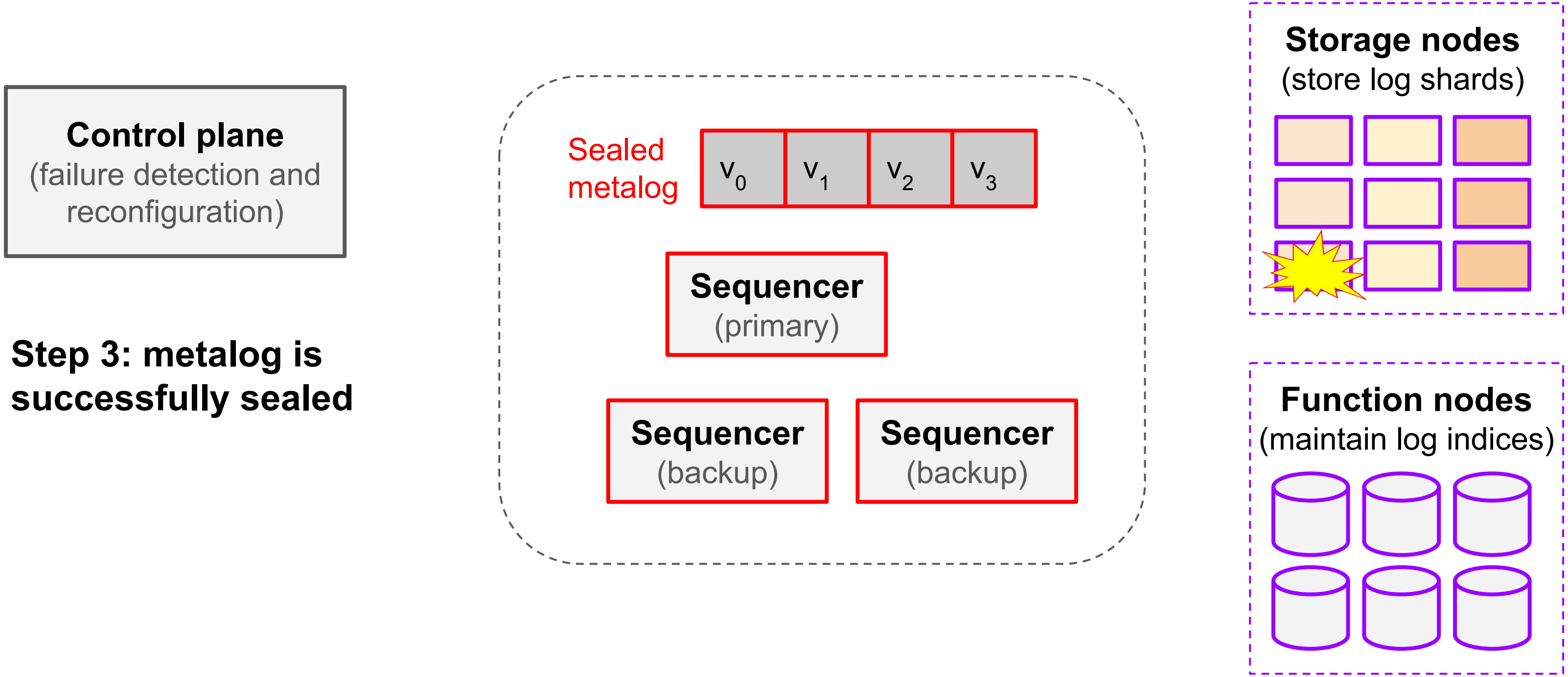
Boki uses Delos [OSDI ‘20]’s fault-tolerant log sealing protocol

Metalog as the Mechanism for ***Fault Tolerance***



Step 3: metalog is successfully sealed

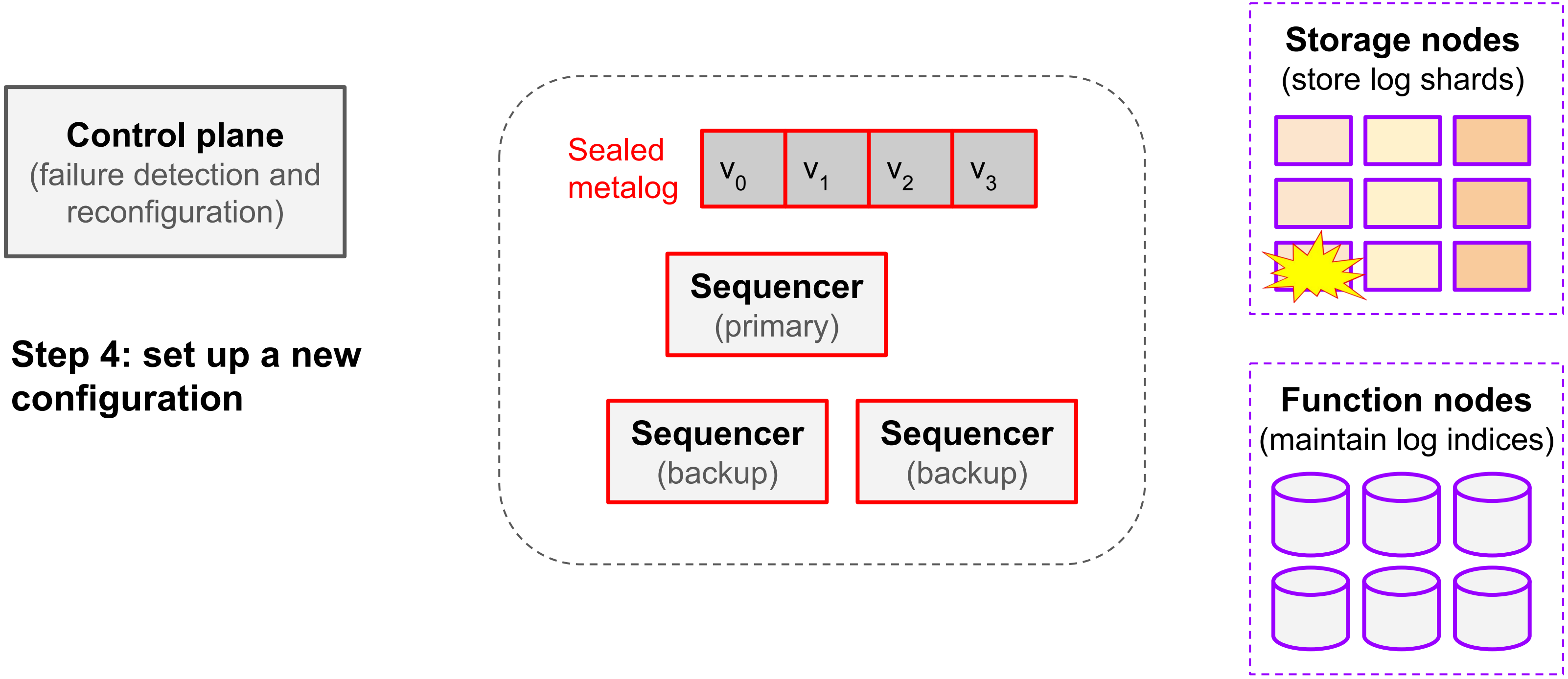
Metalog as the Mechanism for ***Fault Tolerance***



Step 3: metalog is successfully sealed

All frozen as metalog is sealed

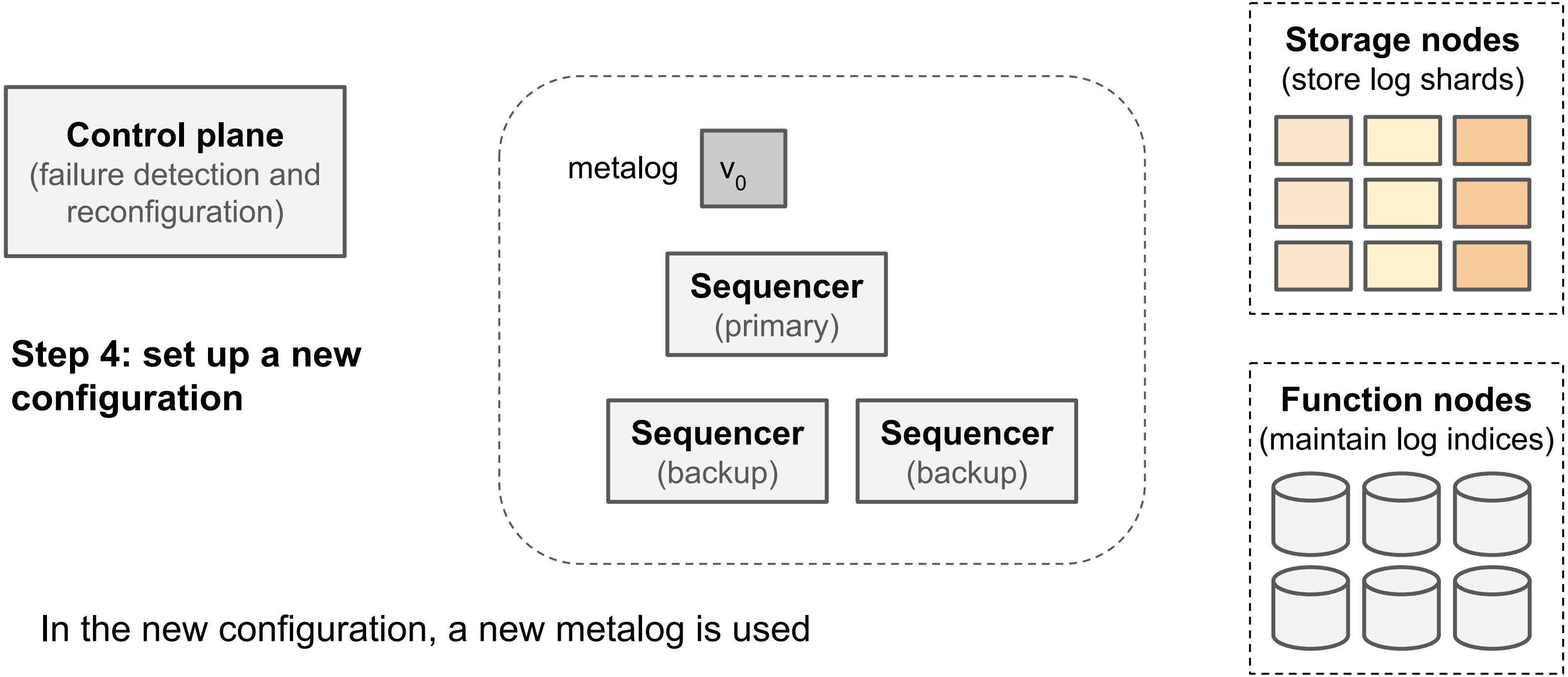
Metalog as the Mechanism for ***Fault Tolerance***



Step 4: set up a new configuration

All frozen as metalog is sealed

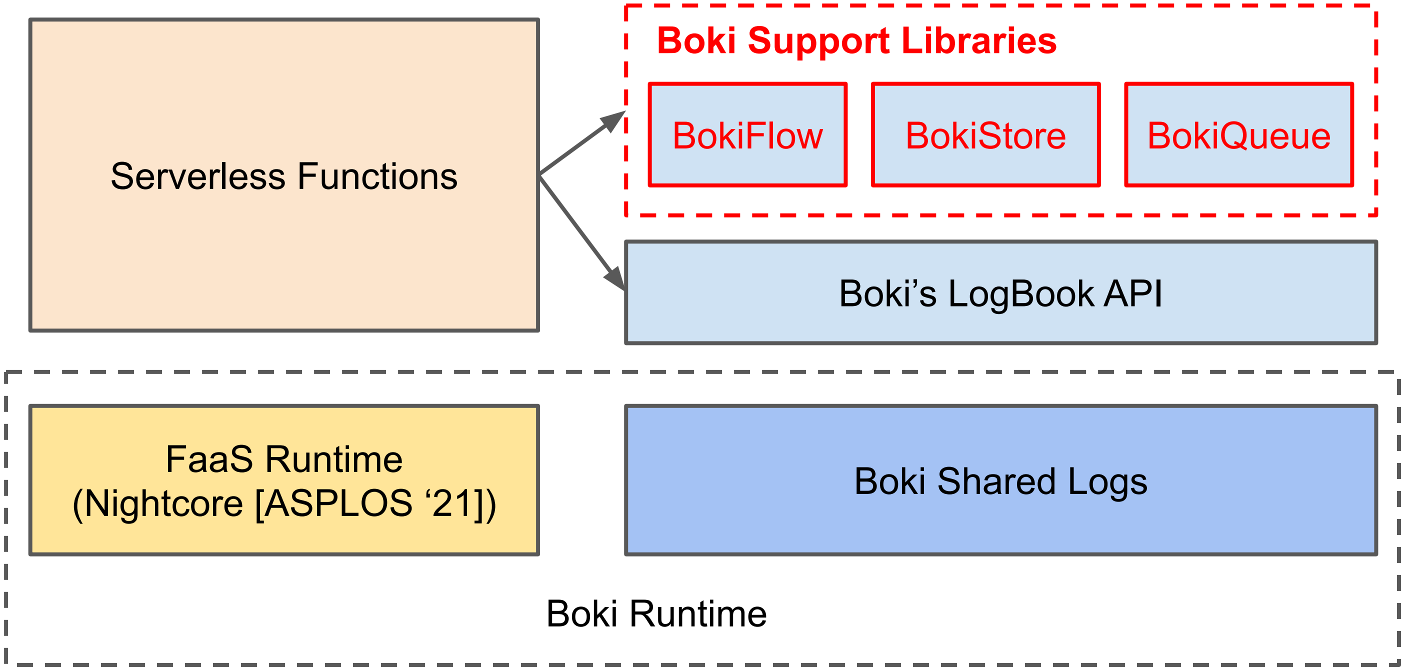
Metalog as the Mechanism for ***Fault Tolerance***



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 — **1,137 LOC**

- 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 — **1,207 LOC**

- 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 — **369 LOC**

- 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

Concurrent writers	320	640	1280	2560
Throughput (KOp/s)	130.8	279.2	604.4	**1,159**

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

Log read

	Local index (cache hit)	Local index (cache miss)	Remote index
50% latency	**0.12ms**	0.57ms	0.79ms
99% latency	0.72ms	1.48ms	2.90ms

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]

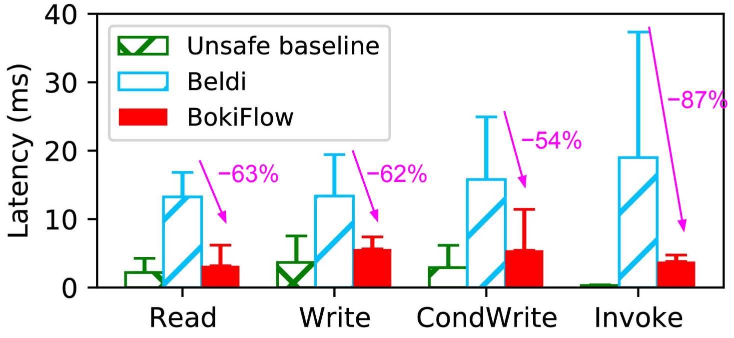
Log-based protocols for fault tolerance. Beldi builds logging layer over DynamoDB.

BokiFlow

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

Microbenchmarks of primitive operations

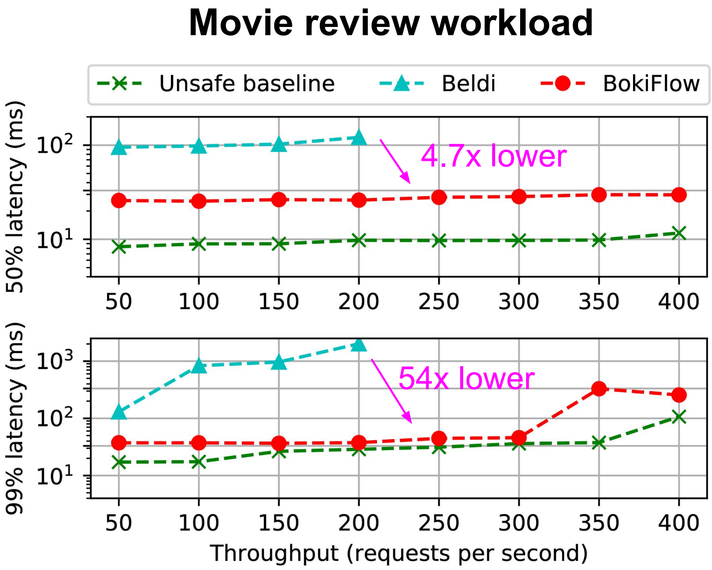
(main bars show 50% latencies, error bars show 99% latencies)

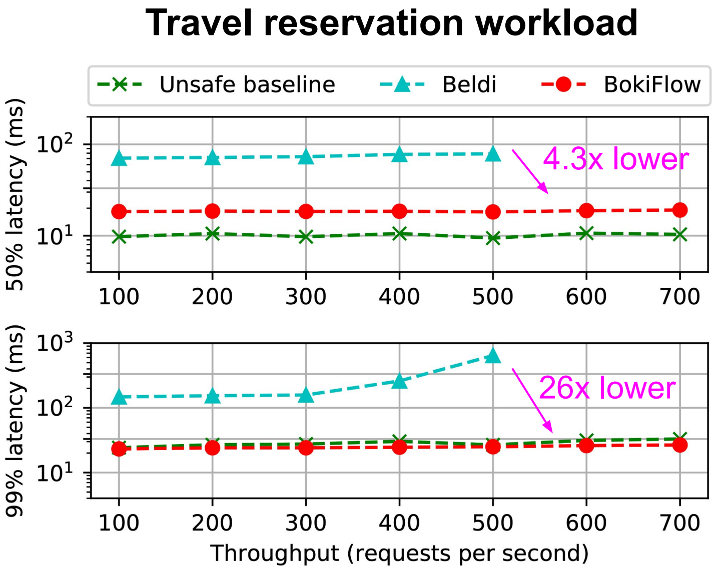


Boki provides a more performant logging layer to support Beldi’s fault-tolerance mechanisms

Evaluation: Serverless Workflows

Comparing BokiFlow with Beldi

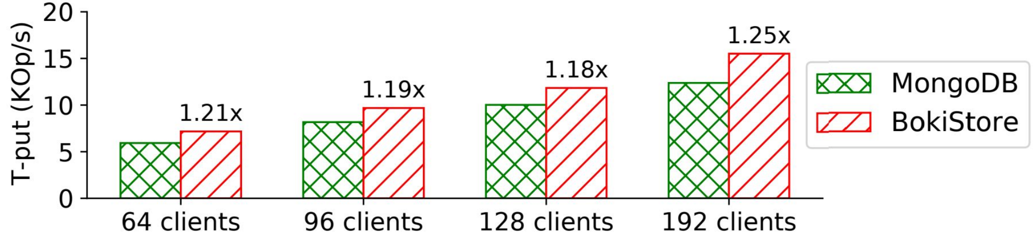




Evaluation: Object Store

Comparing BokiStore with MongoDB on Retwis (a Twitter clone) workload

Throughput



Up to 25% higher throughput

Latency by request types

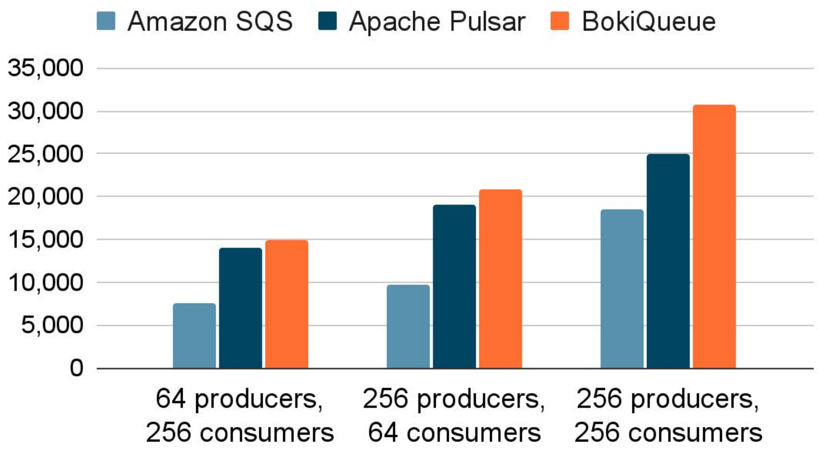
Request types	50% latency Mongo	50% latency Boki	99% latency Mongo	99% latency Boki
`UserLogin` (non-txn read)	**0.86**	1.47	**3.32**	6.32
`UserProfile` (non-txn read)	**0.86**	1.05	**3.57**	5.29
`GetTimeline` (read-only txn)	7.57	**3.35**	25.01	**11.38**
`NewTweet` (read-write txn)	7.72	**5.30**	21.39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

Executing transactions up to 2.3x faster

Evaluation: Message Queues

Comparing BokiQueue with Amazon SQS and Apache Pulsar

Message throughput



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

Delivery latency (median)

	Amazon SQS	Apache Pulsar	BokiQueue
64P/256C	6.08ms	7.39ms	**3.70ms**
256P/64C	99.8ms	7.81ms	**6.61ms**
256P/256C	12.1ms	8.21ms	**7.96ms**

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!