FLEXLOG: A Shared Log for Stateful Serverless Computing

Dimitra Giantsidhi
University of Edinburgh

Emmanouil Giortamis
TU Munich

Nathaniel Tornow
TU Munich

Florin Dinu
Huawei Munich Research Center

Pramod Bhatotia
TU Munich
University of Edinburgh

ABSTRACT
Stateful serverless applications need to persist their state and data. The existing approach is to store the data in general purpose storage systems. However, these approaches are not designed to meet the demands of serverless applications in terms of consistency, fault tolerance and performance.

We present FLEXLOG, a storage system, specifically a distributed shared log, distinctively designed to meet the requirements of stateful serverless computing while mitigating the relevant system bottlenecks. FLEXLOG’s data layer leverages the state-of-the-art persistent memory (PM) to offer low latency I/O and improve performance. To match the performance, FLEXLOG’s ordering layer employs a scalable design, namely a tree-structure set of sequencer nodes. Importantly, this design provides serverless applications with the flexibility to implement different consistency guarantees and to seamlessly support multi-tenancy configurations.

We implement FLEXLOG from the ground up on a real hardware testbed and we also prove the correctness of our protocols. In particular, we evaluate FLEXLOG on a cluster of 6 machines with 800 GB Intel Optane DC PM over a 10 Gbps interconnect. Our evaluation shows that FLEXLOG scales to millions of operations per second while maintaining minimal latency. Our comparison with the state-of-the-art shared log for serverless, Boki, shows that we achieve 10× better throughput in the storage layer and 2× – 4× lower latency in the ordering layer, while also providing flexibility to support different consistency properties and multi-tenancy.

CCS CONCEPTS
• Computer systems organization → Cloud computing.

KEYWORDS
Serverless Computing, Shared Distributed Log, Persistent Memory

1 INTRODUCTION
Motivation. Serverless computing is gaining increasing popularity for building scalable cloud applications as it offers the potential to program the cloud in an autoscaling, pay-as-you-go manner. This is evident from the fact that major cloud providers develop serverless computing frameworks, e.g., AWS Lambda [4], Azure Functions [5], Google Cloud Functions [17], and are used in diverse applications, e.g., video processing [45, 66], data analytics [86, 115], machine learning [54, 85, 120] and others [65, 84, 135].

Unfortunately, the stateless nature of serverless functions is opposed to the stateful applications that are built with them [73, 117, 122, 138]. Such applications are often comprised of multiple functions that need to share or persist their state and data [52, 97]. Current industry approaches [4, 69], rely on general purpose storage services [43, 44, 60, 68] for inter-function communication or data/state persistence. However, these approaches fail to achieve strong consistency and fault tolerance while maintaining high performance and scalability [108, 118].

Distributed shared log systems [47, 62, 107, 128] can offer a promising solution for serverless applications. A shared log, an append-only sequence of records, is typically composed of two core components, a data layer that replicates the appended records and an ordering layer that serializes the records in a meaningful order. As such, shared logs offer a fundamental building block for various high-level data structures and systems that are consistent, durable, and scalable [3, 28, 48, 49, 129]. Shared logs’ strong properties can benefit serverless computing; they offer performance while freeing distributed applications from the burden of managing the details of fault-tolerant consensus [49].

Limitation of state-of-art approaches. Existing shared log systems [2, 47, 49, 62, 83] come with limitations that arise especially in the context of serverless computing. These systems build on top of SSDs which incur high I/O latency. For example, the state-of-the-art shared log Boki [83] reports 1–3 ms read latency which can be a problem for short-lived serverless applications that access the storage frequently (§ 3.1). Secondly, their vast majority are designed for total ordering. However, we show that total ordering comes with a performance cost (§ 3.3) while we find it unnecessarily strict for (chained) serverless applications whose updates are applied to disjoint data or in a highly parallel manner (e.g., data analytics [1, 109]). In total, the slow storage layer combined with the strict ordering layer limit the system’s scalability which in turn complicates multi-tenancy, e.g., bursts of serverless functions, as well as high function concurrency.
Key insights, contributions and high-level design. In this work we address these limitations. We introduce FlexLog, a shared log system that is carefully designed for serverless’ requirements. In FlexLog, we overcome the bottlenecks in the storage layer where we embrace the opportunity to leverage modern storage technologies, such as persistent memory (PM) [77], and drastically improve the I/O latencies (§ 5.2). In addition, we tackle the ordering layers’ limitations by implementing a fast ordering layer that offers flexibility in ordering semantics; serverless applications can implement different consistency properties when they need them. For example, data analytics applications [93] do not necessarily require a strict ordering of all events that is traditionally offered by conventional shared log systems. While we offer flexibility, we take great care to preserve data consistency and isolation (for correctness) and, thus, we expose transaction-like operations that allow applications to append multiple records atomically in different parts of the log (§ 6.4). Our design choices for the ordering protocol increase scalability and also enable multi-tenancy (§ 5.1). We implement (§ 6) and provide correctness proofs (§ 7) for all our append/read protocols and auxiliary operations for accessing FlexLog.

At a high-level, FlexLog layers a high-performance ordering layer on top of a data layer, a set of storage nodes that replicate the log. These storage nodes implement a tiered architecture where we use PM for persistence, DRAM for caching and SSD for flushing old parts of the log. On top, our ordering protocol is a tree structure of sequencer nodes that assign sequence numbers to records that denote partial or total ordering. FlexLog overlaps record replication and ordering targeting low latency.

We build FlexLog from the ground up in GoLang [16]. For the networking, we use remote procedure calls (RPCs), specifically gRPCs [18]. For the storage layer, we use Persistent Memory Development Kit (PMDK) [78]. Lastly, we implement a Go-API for accessing PM and creating memory bindings between our Go-API and the C++ implementation of PMDK libraries.

We run FlexLog on a cluster of 6 machines with 800 GB Intel Optane DC PM over a 10 Gbps interconnect. Our evaluation shows that FlexLog seamlessly scales to millions of operations per second. We make an apples-to-apples comparison of FlexLog’s storage and ordering tiers with Boki [83], the state-of-the-art shared log for serverless. We show that our storage layer is an order of magnitude faster than Boki’s while our ordering layer achieves 2–4X lower latency.

To sum up, our paper makes the following contributions:

- Based on our analysis of the bottlenecks and requirements involved in stateful serverless computing (§ 3), we propose a shared log architecture that builds on the state-of-the-art persistent memory and offers a scalable ordering protocol (§ 4 and § 5.2).
- Our system allows serverless applications to implement flexible ordering semantics when they need them and can support multi-tenancy (§ 5.1).
- We provide comprehensive protocols around the shared log abstraction (append/read and auxiliary operations) (§ 6) and we provide proofs of correctness (§ 7).

Limitations of FlexLog. FlexLog is based on Intel Optane DC PM which recently has been discontinued by the vendor [34]. However, we believe that this unfortunate event is not restrictive for FlexLog. Instead, the upcoming CXL [19, 23] technology is quite promising because it provides a Load-Store IO fabric at rack-level in memory and storage pools [20], facilitating the FlexLog’s adoption.

2 BACKGROUND

Distributed shared log. Serverless functions require to persist and communicate their state and data with consistency, fault tolerance and scalability [83]. Shared logs can provide a solution to serverless state management systems as they are a fundamental building block for various systems (e.g., storage systems [28, 48, 49, 129], message queues [95], databases [46]) that meet these properties. Distributed shared logs or simply shared logs offer the view of an append-only sequence of records. The shared log has gained traction both in research [47, 48, 62, 107, 111] and industry [10, 14, 26, 95, 126] because it allows applications to seamlessly replicate and persist state, e.g., by appending updates to the end of the shared log and reading back updates from it.

At a high-level, these systems traditionally consist of three logical components: a data layer, an ordering layer, and clients. The data layer replicates and stores the records persistently while the ordering layer is responsible for ordering records by assigning each record a distinct position in the log. Lastly, clients use the shared log’s API of appending and reading which interacts with the data and the ordering layers.

Persistent memory. At the same time, serverless functions need to persist and update their state/data efficiently to reduce client costs. In this direction, we embrace the opportunity to leverage modern storage technologies, specifically persistent memory (PM) [77], to drastically improve serverless functions’ storage I/O latencies. PM offers durability with close-to-DRAM memory accesses. PM is connected to the CPU via the memory bus, and resides between the main memory and conventional storage such as SSDs or HDDs in the system stack.

Our work builds on PMDK [78], a collection of libraries and tools developed by Intel aiming to support and facilitate application development for persistent memory. Conveniently, our work leverages PMDK’s transactional API (BEGIN, PUT, GET, COMMIT/ROLLBACK) to handle PM’s architectural challenges, e.g., flushing volatile CPU caches, metadata persistency and crash consistency [56, 116, 130].

3 MOTIVATION

3.1 Characteristics of Stateful FaaS

Serverless functions or function-as-a-service (FaaS) [4, 5], allow developers to upload simple functions to the cloud provider which are invoked on demand. While cloud providers offer a variety of execution environments (compute tiers) allowing a pay-as-you-go manner [4, 5, 17, 24], managing serverless functions’ state or persistent data (stateful functions) still remains a challenge [96, 97]. Currently, both research and academia approached this by building or relying on general purpose storage services [11, 43, 44, 60, 68]. These storage services do not usually meet serverless application requirements in terms of performance, cost, fault-tolerance, and consistency [97]. More importantly, serverless functions present the following characteristics that need to be taken into account when designing a storage system for state management.

Low-latency and frequent storage accesses. Serverless functions are short-lived [119]—the Azure study shows that half of the functions complete within 1 s and > 90% of them have runtime below...
FlexLoc: A Shared Log for Stateful Serverless Computing

To understand the bottlenecks, we locally ran and profiled two popular serverless workloads [15, 92, 93], a video processing and a gzip compression workload. Table 1 shows our findings. We found that around the 40% of the CPU time is spent on accessing storage. Note that this reported percentage is based on local storage; we expect even worse results in a real serverless environment where data needs to be synced among the storage nodes and functions might access shared data remotely. Consequently, the short-lived nature of serverless and the frequent storage accesses show that a storage system for serverless needs to optimize for storage latency.

### Flexibility: consistency properties and multi-tenancy

We observe that managing serverless application state requires a variety of consistency guarantees, from strict serializability [74] (e.g., transactions [135]) to weaker consistency guarantees [7] (e.g., data analytics [86, 115], such as map-reduce [93] or graph processing [109]). Both data analytics paradigms follow a similar execution model; a chained set of parallel tasks (phases) accesses, processes and updates the persistent data. That said, we only need to serialize the execution phases but not the parallel tasks within each phase. Consequently, given that strict serializability is expensive and impractical (e.g., network partitions) [107], our observation implies that a well-designed storage system must offer configurable ordering guarantees between updates—now on we refer to this as flexible ordering semantics. Note that even with flexible ordering, the data isolation and correctness properties should not be violated under failures or interruptions.

In addition to that, a carefully-designed storage system for serverless needs to scale well to support multi-tenancy, i.e., to handle bursts and high concurrency of serverless applications and functions.

To sum up, we need to target the following properties in the design of a storage system for serverless state management:

- **High-performance storage accesses** for fast function start-up times, state and data persistence and retrieval.
- **Scalability** for handling bursts of serverless functions as well as high function concurrency.
- **Fault tolerance** for mitigating the impact of failures and decreasing client costs.
- **Flexibility** for ordering guarantees ranging from weaker to stronger, to cater to the diverse consistency requirements of serverless applications, and to support multi-tenancy.

### 3.2 Shared Logs for Serverless Computing

The solutions currently used for serverless state management are not specifically designed to meet the demands of serverless applications. More specifically, current serverless applications primarily rely on cloud storage services (e.g., AWS S3 [44], Google Cloud Storage [68], and high function concurrency.

<table>
<thead>
<tr>
<th>syscall</th>
<th>Video processing</th>
<th>Gzip compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>open()</td>
<td>17%</td>
<td>19%</td>
</tr>
<tr>
<td>read()</td>
<td>15%</td>
<td>3.2%</td>
</tr>
<tr>
<td>write()</td>
<td>N/A</td>
<td>22%</td>
</tr>
<tr>
<td>fstat()</td>
<td>5.1%</td>
<td>2.9%</td>
</tr>
<tr>
<td>close()</td>
<td>6%</td>
<td>1%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>41%</td>
<td>48.1%</td>
</tr>
</tbody>
</table>

Table 1: Profiling of two serverless functions [15, 92, 93]. Percentage of CPU time spent in accessing local storage.

Figure 1: Storage latency for read and write operations.
whereas the advent of PM opens up new opportunities for improvement.

Storage layer bottlenecks. Scalog [62] is built on top of SSDs, incurring high latency for I/O. Due to that, there is a shift towards modern PM technologies in cloud data management systems [27, 75, 132]. Following this trend, we quantify the SSD overheads by measuring the latencies for read and append operations for three competitive baselines: (i) PM via kernel-bypass [91], (ii) PM via OS syscalls and (iii) SSDs. Figure 1 shows the average latency of read and write operations on PM (pmem_read, pmem_write) compared to read and write PM accesses through OS interfaces (read_syscall, write_syscall) and SSDs (fileio_read, fileio_write). Our experiment shows that PM improves I/O latency up to 10× compared to using SSDs. Further, bypassing the kernel to access PM results in up to 100× lower latency.

Ordering layer bottlenecks. Scalog offers a global totally-ordered log by layering a Paxos protocol [98] on top of its storage layer to replicate the tail of the log. Paxos can be costly: classic (leaderless) multi-proposer Paxos [98] runs at least two phases (Propose and Accept) for every single increment of the tail while optimized versions of it, i.e., Multi-Paxos [124], elect a (unique) primary to handle all requests which can become a bottleneck [9]. In case of multiple proposers, Paxos might require to re-execute its phases for an unbounded number of times to reach consensus [6]. Indeed, while experimenting with the protocol [25], we identified livelocks. Specifically, we did not see any progress as concurrent proposers were competing for the tail of the log. Our findings are not orthogonal to what research is suggesting. Practical systems [53] use Paxos out of the critical path, e.g., for leader election, lease management [70], etc. or switch to fallback protocols [124] to improve latency.

4 FLEXLOG’S ABSTRACTION AND SYSTEM MODEL

Definitions. We define a log object as a bounded sequence of Write-Once-Read-Many records, \( \{W_h, \ldots, W_t\} \). If \( h > t \) we say that the sequence is empty, \( \emptyset \). Initially \( h = 1, t = 0 \). A shared log object simply is a concurrent log object.

We define a shard as a set \( S \) of \( r \in \mathbb{N}^+ \) replica nodes (or simply, replicas). The number of replicas within a shard is equivalent to the replication factor of the system.

We define a color \( C \) as a set of \( n \in \mathbb{N}^+ \) shards, \( \{\text{Shard} # 1, \ldots, \text{Shard} # n\} \). The colors in FLEXLOG are an abstraction of the notion of a region within the log. From now on, the terms region and color are used interchangeably.

We define the data layer \( L \) as a set of \( I \in \mathbb{N}^+ \) colors, \( L = \{\text{Color} # 0, \ldots, \text{Color} # I\} \). Lastly, we consider a set \( O \) of nodes, \( O = \{\text{Seq} # 0, \ldots, \text{Seq} # O\} \) that comprise the ordering layer of our system. From now on, we refer to the nodes of the ordering layer as sequencers.

Topology. The data layer and the ordering layer, formally, \( L \cup O \), make up the topology of FLEXLOG. FLEXLOG’s ordering layer is structured as an \( n \)-ary tree, abstracting the color hierarchy on top of the data layer. All replicas of a shard are connected to a leaf-sequencer node from the ordering layer tree that resides in the same color. A color stores an ordered (part) of the log and might consist of multiple other regions. FLEXLOG offers the abstraction of multiple totally ordered shared logs all of which are part of the master-region (root of the tree). Records of different colors are ordered arbitrarily.

Figure 2 shows an example topology that is comprised of three colors (red, green, blue), and thus three sequencers (Seq #0 to Seq #2) that are responsible for ordering the records of each color. The sequencer Seq #0 is the root node (master-region) of the ordering layer tree.

Network model. We assume a partially synchronous message-passing system with unbounded message size [64]. Message delays are initially unbounded for some unknown but finite time, then a delay bound \( \Delta \) starts to hold. We quantize time in rounds of communication. In any given round, a node may broadcast a message, the recipient nodes deliver this message and then possibly perform some negligible local computation.

We assume the system’s network to be reliable (lost, re-ordered or double-sent messages are not allowed). We also assume that there exists a reliable broadcast (from now on simply broadcast) primitive that guarantees (as in [61]), that: if a correct process delivers a message \( m \), then all correct processes will eventually deliver \( m \). Second, a message \( m \) is delivered by each correct process at most once if it was previously broadcast by some process. Finally, correct processes deliver the same messages in the same order.

In practice, FLEXLog builds reliable (FIFO) network connections relying on the TCP protocol, as in [36]. Further, our broadcast properties are realized by combining TCP connections and FLEXLog’s recovery algorithms (§ 6.3) when failures occur. We argue that our requirement for a reliable broadcast is not a limitation. In fact, we are inspired from the modern trends in cloud network infrastructure which have explored the synergies between reliable (or atomic) broadcast algorithms and fast network stacks (RDMA, SmartNICs, programmable switches [59, 82, 99, 113]), showing that they can benefit various distributed systems [50, 71, 101, 102, 114].

Fault model. We assume crash failures; a process can fail completely or omit some computation/communication steps. For liveness, we assume a crash-fail recovery model for the storage nodes: replicas can fail at any point but they will recover eventually and the persistent state (PM and SSDs) is preserved (not corrupted). The shards run a read-one/write-all replication protocol. As in similar protocols [89, 125], upon network partitions (or replicas’ failures), we choose to sacrifice availability to maintain consistency (CAP theorem [51]).

For the ordering layer, each sequencer has 2\( f \) backups where at most \( f \) nodes can crash. Failures (e.g., crashes, network partitions, etc.) are identified by noticing message delays greater than \( \Delta \).
Appends records to the log of color c and returns the last assigned SN

Reads a record with SN from the c-colored log

Receives all records of the c-colored log. The orchestrator can co-locate FaaS applications with FlexLog's component(s) based on resource usage.

5 DESIGN

5.1 Overview

Serverless architecture. Figure 3 shows an example serverless (FaaS) infrastructure which includes FlexLog. Particularly, serverless infrastructures implement a tiered architecture where they comprise of an execution layer (compute tier) and a storage layer (in our case FlexLog). The execution layer (inspired by [4, 33]) is composed of front-end servers that receive and authenticate external requests. A request is then routed to the orchestrator which keeps track of the entire cluster resource utilization. The orchestrator communicates with the workers’ manager which chooses a physical host to launch the new function instance [1]. The workers’ manager retrieves the necessary function state, e.g., a Docker image or others [41] from FlexLog and starts the instance [4]. Finally, each function performs language runtime initialization, after which the user-provided function code retrieves the function invocation’s inputs, e.g., data from FlexLog.

FlexLog components. FlexLog comprises (i) the ordering and (ii) the data layers (Figure 3). The ordering layer is a tree structure of sequencers that achieves fault tolerance through backup nodes. Each sequencer assigns sequence numbers to all Append calls that refer to its owned region, thus, serializing them into a total order in that region. The storage (data) layer, which stores the logs, consists of the storage layer and the replication layer. The storage layer consists of storage servers (replicas) organized into shards sitting below a replication layer that enforces a replication protocol for fault tolerance. Figure 3 also visualizes the storage stack of a replica (§ 5.2). Each shard is connected to a leaf-sequencer of the ordering layer.

Lastly, the serverless functions implement the FlexLog-API (Table 2) and communicate directly with replicas of shards to issue Append or Read requests in the respective regions.

FlexLog-API. Table 2 shows the FlexLog-API. Applications can use existing colors or create new colored regions in FlexLog. As shown in Figure 3, to append a record in color c, a function needs to send the Append request to every replica of a (random) shard in c [5]. Each replica stores the record and requests a sequence number (SN) (that is unique for each record in c) from the ordering layer. Each replica views a persistently stored record that has been assigned a SN as committed and can therefore serve it on read requests. An Append call is considered complete when all replicas have committed the record. To serve a Read call for a specific SN of a given color c, the application contacts a random replica of each shard of c and receives the record from (at least) one of them. (Figure 3, (6)). The auxiliary operations Subscribe and Trim are used to fetch all records of c and erase all records in c up to a sn (given as an argument) respectively.

Table 2: FlexLog’s basic API.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Append(r, c)</td>
<td>Appends records to the log of color c and returns the last assigned SN</td>
</tr>
<tr>
<td>Read(SN, c)</td>
<td>Reads a record with SN from the c-colored log</td>
</tr>
<tr>
<td>Subscribe(c)</td>
<td>Receives all records of the c-colored log</td>
</tr>
<tr>
<td>Trim(SN, c)</td>
<td>Garbage collects the log of color c by deleting all records with sn ≤ SN</td>
</tr>
<tr>
<td>AddColor(c, cp)</td>
<td>Creates a new c-colored log with cp log as its parent</td>
</tr>
</tbody>
</table>

| Table 2: FlexLog’s basic API. |

Example usage of FlexLog-API. Listing 1 shows how to use FlexLog-API to implement a message queue and allow two serverless functions, Func1 and Func2, to communicate with each other. Func1 appends data in a data-log, the yellow log (line 22) and creates a black-colored log, the message queue, where it enqueues the sequence number, sn_y of the previous append (line 24-25). Func2 looks up the black log (lines 28-30) until the expected record (and its SN) is found.

```
Listing 1: Example: A message queue between two functions
```

Consistency models and multi-tenancy. FlexLog’s design allows applications to implement various consistency models while it is designed for scalable multi-tenancy scenarios. We next show how. Applications can support linearizability and sequential consistency by appending to a single color of the log. The leaf-sequencer of that color is the point of serialization. For example, in Figure 3, App2’s functions utilize a single totally-ordered color (yellow) for appends [7]-[8]. Applications can also express causality (happens-before relationship) by implementing synchronization primitives, i.e., locks and barriers, similarly to [76]. In the example of the chained execution, e.g., map-reduce, we suggest the following: each mapper writes to a distinct colored log. Upon its completion, it appends a final record to a specific log, the black log. Reducers wait until all mappers append final records on the black log. This can be done by reading the tail or subscribing.

FlexLog can support weaker, eventual consistency models, that are common in many large distributed systems. Functions can write to arbitrary colors; subsequent reads do not have to reflect the latest append and reads across multiple records might reflect an incoherent mix of not ordered appends, e.g., App1 (Figure 3) writes and reads from different colors.

Multi-tenancy is supported similarly to eventual consistency. Unrelated applications can define distinct colors in FlexLog. Figure 3
shows an example of two applications, App1 and App2 that update distinct colors, red and yellow, respectively. FlexLog does not impose an ordering relation between records of red and yellow color.

**FlexLog beyond serverless.** We adopt the classical API for shared logs allowing FLEXLOG to easily be adopted in various systems, beyond serverless computing. First, FLEXLOG can be used to implement fundamental primitives for systems such as distributed locking [22, 49], message queues and event sourcing [95], data structures (with relaxed consistency) [42, 48], etc. Importantly, it can be used as an external system for the design of large-scale systems; such as an external commit (transactions) log to aid distributed systems (e.g., databases) to re-sync and restore their state and data after failures. It can further improve scalability in messaging and chat applications where a color can represent the history of messages in a chat room. Lastly, FlexLog can help to run pipeline workflows, e.g., learning pipelines, for delivering identical event streams to multiple ML model training services.

### 5.2 FlexLog Architecture

**Ordering layer.** The ordering layer of FlexLog is a scalable and fault-tolerant tree structure of server nodes called sequencers. Sequencers assign distinct 64-bit sequence numbers (SNs) as a response to order requests (OReqs). SNs are realized as the value of an increasing counter inside the sequencer and determine a total order between all OReqs to this sequencer over time. The tree hierarchy of sequencers describes the logical division into regions by viewing each sequencer as the source of total ordering for each region. The sequencer that provides the total order for a region resides at the root of the sequencer (sub-)tree of the respective region. An OReq specifying a color c of a region arrives at one of the leaves of the sequencer tree, which resides in the region or a sub-region of c. The OReq gets propagated through the sequencer tree towards the root, until it reaches the root sequencer of the requested region c. The root sequencer then replies with the distinct SN, which will be sent back to the request’s origin on the same path.

To improve throughput, the sequencers of sub-regions serve as aggregators for incoming OReqs. They merge OReqs that arrive in a specific time interval and that have the same color into a single OReq. A merged OReq then also specifies the number n of single OReqs it consists of. By replying with the SN s, the sequencer assigns all SNs in the range [s, s+n] to the merged OReq, which are distributed to their respective origin.

**Sequencer replication.** To tolerate failures, we use 2f backup nodes that only replicate the epoch number e (incremented on a leader’s failure) of the current sequencer. As such, the backups do not participate in normal execution, do not add up in latency and they are only “activated” when the sequencer fails (detectable through heartbeat messages). In FLEXLOG, the new sequencer is elected as the backup node with the highest known e and the highest node-id (as a tie-breaker). To avoid the split brain problem, e.g., two sequencers which both think they are the new leader, a (old) sequencer shuts down if it does not receive heartbeats from the majority for some time. In addition, every new sequencer sends initialization requests to all replicas and waits to be acknowledged by all before executing the protocol (§ 6.3).

**Safety.** A newly elected sequencer, prior to running the ordering protocol, replicates its epoch in (at least) the majority of backups. That follows that if this sequencer fails, the latest e will survive in at least one replica. The leader issues and increases SNs (64-bit) of the form: the most significant 32-bits consist of the epoch e, and the least significant 32-bits are the result of a counter (incremented on each OReq). That satisfies the criterion for correctness; the SNs are increasing.

**Data layer.** The data layer is organized as the replication layer built on top of the storage layer.

**Replication layer:** For high throughput and fault tolerance, FLEXLOG stores and replicates records across multiple shards. All replicas of a shard connect to the same leaf-sequencer. By definition, a shard is allocated to the region of its leaf-sequencer and all its super-regions. Our replication protocol, realized as an atomic broadcast, allows linearizable reads.

For an append, the record is broadcast to all replicas of the shard, which store the record and request a SN from the ordering layer. When a SN has been determined, the leaf-sequencer broadcasts the SN to all replicas. At this point, replicas can commit the record. All records are committed in a total order that is defined by the ordering layer, and every replica commits the record before the append protocol is completed. That allows linearizable reads on every replica.

**Storage layer:** Each replica implements a storage server that consists of three tiers: (i) an in-memory volatile cache, (ii) the stateful log in PM and (iii) the secondary persistent storage (SSDs). The cache
optimizes the read path by storing some recently accessed records. The stateful log (PM tier) stores records in a crash-consistent manner. Lastly, FlexLog makes use of the SSD in case the log grows indefinitely. By default, records appended to the log are stored in PM (and the cache). If the cache size limit is reached, the oldest record is evicted and replaced by the new record. If the log’s size limit in PM is reached, a (user-configurable) contiguous portion from the start of the log is flushed to SSD and removed from PM. Symmetrically for read operations, the volatile cache is first read, then PM, then the SSD.

6 FLEXLOG SYSTEM PROTOCOLS

We next describe FlexLog’s protocols: we show the Append and Read protocols (§ 6.1), the Subscribe protocol (used to fetch all records of a colored-log) and Trim protocols (used to delete part of the log) (§ 6.2) and the recovery protocols (§ 6.3) that handle failures. Lastly, we present the multi-color Append protocol in § 6.4.

6.1 Read and Append Protocols

Append protocol. Serverless functions append records to the log of color \( c \) by calling \( \text{Append}(r[\cdot], c) \) of FlexLog-API. The Append creates a distinct token \( t \), consisting of the value of a monotonically increasing counter and the id of the caller (FID). The FIDs are distinct across all serverless functions that are appending on \( c \) (Alg.1-6). From then on, the request is broadcast to all replicas of a randomly-chosen shard of \( c \) (Alg.1-7). The first round ends with the replicas receiving this request. On receiving the append request, all replicas store the record (identified by \( t \)) persistently. The second round starts when the replicas send OReq to the ordering layer, requesting a SN in the region \( c \) for the record (Alg1:18-19). In the worst case (total ordering), the OReq needs to traverse the ordering layer tree up to the root. When the SN for the OReq is determined, the leaf-sequence for \( c \) broadcasts (third round) an OResp \((t,sn)\) to all replicas of the shard, where \( sn \) is the SN of the last record issued by the root sequencer of region \( c \), denoting the distinct place of the records in the shared log of color \( c \) (Alg1:33-35). Upon receiving the OResp, replicas commit the record, making it visible to other functions, discoverable with \( sn \). The fourth and final round starts with each replica sending an ACK \((t,sn)\) to the initiating function (Alg1:24). The function returns \( sn \) to the application when every replica has acknowledged the records (Alg1:8-9).

Read protocol. Applications call \( \text{Read}(sn,c) \) of FlexLog-API to read the record stored with \( SN \) on the log of color \( c \). In the first round, the function broadcasts the request \((sn,read)\) to one replica of every shard \( c \). Each replica reads the record with \( SN \) for color \( c \) on its local storage, and, if exists, it sends (second round) it back to the caller. Otherwise, it replies with a \( ⊥ \) value. Since each record is stored only on one shard, only one of the replicas can return the record. If all replicas return \( ⊥ \), then the shared log of color \( c \) has no record with \( SN \) \( sn \) stored.

6.2 Auxiliary System Operation Protocols

Subscribe protocol. Applications invoke \( \text{Subscribe}(c) \) to receive all the records of the log of color \( c \). Similarly to the read protocol, Subscribe broadcasts (first round) a subscribe request \((\cdot, subscribe)\) to one replica of each shard in \( c \). Upon receiving the request, each replica replies (second round) with its local view of the log in \( c \). Finally, the protocol reconstructs the log of color \( c \) by sorting all records received based on their SNs.

Trim protocol. Applications invoke \( \text{Trim}(sn,c) \) to delete all records of color \( c \) with \( SN \leq sn \). The Trim protocol broadcasts (first round) a trim request \((sn,trim)\) to all replicas of all shards in \( c \). On receipt, the replicas delete every record of the log fragment of \( c \) that have been assigned a SN that is smaller or equal to \( sn \). The Trim completes when all replicas acknowledge the operation to all replicas (second round) and send to the caller a \([\text{head},\text{tail}]\) pair message (third round).

6.3 System Recovery Protocols

Replica failures. Failure detection. In line with practical systems [36, 67], FlexLog’s replicas periodically exchange heartbeat messages with the sequencer (or other replicas) to detect failures. If a heartbeat message times out, the replica transits to recovery mode.

Recovery. When a replica recovers, a synchronization phase, sync-phase, takes place to force all replicas to synchronize their state and ensure that no SNs, received only by some replicas, are missed by others. For example, in the extreme scenario where the crashed replica was the only one to receive an SN, e.g., due to sequencer failures (right before its crash), this SN will eventually be received by all replicas. (We elaborate on sequencer failures later in the section.)
The recovered replica sends to all replicas of the shard a sync-request. Replicas receiving a sync-request transition to sync mode and stop processing new append requests or sequencer messages. As a response to this request, the replicas send their latest state, e.g., their known $e$ (current active sequencer) and their latest committed SN. If the $e$ does not match the $e$ known by all replicas, e.g., the recovered replica sees an old sequencer due to network partition, it retries the sync-request. Eventually, the old sequencer shuts down and all replicas find out about the new one. Recall that replica failures block append operations. Similarly, further failures or re-starts during the steps below affect availability, not safety.

Once the recovered replica collects all acks for the sync-request from all replicas, it broadcasts the most up-to-date replica id. The outdated replicas fetch the missing entries from the most up-to-date one, as in [63]. Once they sync their log, they ask this step to every other replica (all-to-all communication). A replica waits for acks from all other replicas and then it transits to operational mode. Essentially, the all-to-all communication step at the end of the sync-phase is a synchronization barrier that guarantees that the execution continues iff all replicas are synchronized and are connected to the same active sequencer. Replicas might still need to re-issue OReq requests for records that have not being assigned an SN after the sync-phase. Safety. Upon network partitions and failures, append block. We encountered two problems for the reads: (1) a function reads a value from a “partitioned-out” replica and (2) a function reads a SN of an ongoing (not yet completed) append from (slow) replicas. The first is not a problem. FlexLog is append-only, a written entry cannot be overwritten. As such, reads of (committed) entries, that have assigned a SN, are correct even if the replica is not part of the membership.

The second problem arises from the local reads that FlexLog allows. For correctness, replicas hold read requests (as in [90]) that refer to a SN that is higher than their currently seen maximum SN for a (configurable) amount of time. After this timeout expires without the replica receiving this SN (or an entry with a bigger SN which implies that the requested SN is a hole), the request times out. That does not violate linearizability; instead, it forces the FaaS-application to re-execute the read and (probably) read from another replica. Permanent failures. If the replica crashes permanently, the PM device is corrupted or the PM cannot be migrated to another node, the log served by the replica’s shard can be read but not written as we cannot recover the latest updates (if any) from the crashed PM.

Sequencer failures. Leaf-sequencer failures during broadcasts might lead to scenarios where only some (but not all) of a shard’s replicas have received the SN. Recall that, the backups are stateless, the new sequencer does not know the actions of the old sequencer.

FlexLog needs to ensure that the new leaf-sequencers will start operating only after all replicas have acknowledged it (guarantees that only a single sequencer a time can be active) and have synchronized their log (up to the previous $e$). That follows, that any interrupted broadcast messages from the previous sequencer that have been received by some replicas will eventually be received by all replicas.

To achieve this, once replicas find out about the new sequencer (prior to initializing the ordering protocol with it), they pass through the sync-phase ensuring that they all acknowledge the new sequencer and have completed the messages of the previous $e$.

If a failure occurs at any point, the execution blocks due to all-to-all communication step in the sync-phase. When the failure is restored, all replicas will complete the sync-phase before any new sequencer starts operating. That way, we ensure correctness even in cases where the partitioned-out replica was the most up-to-date replica, e.g., it was the only one that received an OReq before the old sequencer died. This OReq will be received by all before the new $e$ begins.

Failures of the root and middle sequencers do not face such issues as they establish peer-to-peer TCP connections with their parent and children nodes. If the sequencer fails, the on-going requests will timeout and either some sequencer or a replica will resend the OReq.

Hole management. When a sequencer fails and the new primary issues SNs, it possible to have holes in the log where records have no consecutive SNs. However, this does not violate correctness; by definition, the log sequence is not necessarily consecutive. Specifically, append and trim operations are not affected by any holes in the log. In addition, it is accepted for a pair of read operations $r(1,c)$ and $r(j,c)$ to return values $\bot$ and $\neq \bot$, respectively, even if $i < j$.

### 6.4 Multi-color Append Protocol

We design the multi-color append operation, shown in Algorithm 2, that allows applications to atomically append multiple records to more than a single log region at once while ensuring data consistency and isolation (for correctness).

The protocol assumes a special color known to all functions a priori (e.g., the master-region), that acts as a broker for the operation. The protocol works as follows: the function appends sets of records to the log of the special color first (Alg2:3-4). Replicas handle those sets of records as usual except that now they also persist the target color information and the ID of the function as well. After receiving all the acknowledgements, the function broadcasts a special message, end, marking the end of the atomic append (Alg2:5). At this point, each of the participating replicas broadcasts the sets of records one by one, similarly to the single-color append operation (Alg2:14-18). When all sets are successfully appended, the replicas reply with an acknowledgement to the initiating function.

---

2A timeout of 1ms is safe. It is 2-3 orders of magnitude higher than the common case latency of modern data center fabrics [103] even in cases of network congestion [72].
7 PROOF OF CORRECTNESS

We adopt the definition of linearizability from [74] and show that FlexLog is a linearizable shared log object. We do so by showing that the sequential specification’s properties are maintained in the case of concurrent operations.

For simplicity, and without loss of generality, we fix an arbitrary color. For space efficiency reasons, we omit most of the details and instead provide a sketch of proof. For any arbitrary pair of operations, it suffices to show that the acknowledgment of all participating and correct replicas marks the response of the first operation and that happens before the invocation of the second (in the execution history). This means that the second is guaranteed to see the effects (if any) of the first and not vice versa. To achieve linearizability, the following properties must be satisfied at all times:

Property 1 (Consistency). Any two log sub-sequences $s_1$ and $s_2$ must be comparable; there must exist a sequence $s_i$ that is a common consecutive subsequence (substring) of $s_1$ and $s_2$.

Property 2 (Stability). Consider two successive subscribe operations, $s_{i_1}$ and $s_{i_2}$, such that $s_{i_1}$ responds before the invocation of $s_{i_2}$. Let $s_1$ and $s_2$ be the sequences returned by $s_{i_1}$ and $s_{i_2}$, respectively. Then $s_1$ is a substring of $s_2$ in the absence of conflicting trim operation invocations between them.

Proof. Let $s_1 = (W_i, ... W_j)$ and $s_2 = (W_k, ..., W_l)$ and without loss of generality that the subscribe that returned $s_1$ responded before the invocation of that which returned $s_2$. Protocol-wise, the only operation that can modify the tail of the log, i.e., transform $j$ to $l$, is the append operation. If between $s_{i_1}$ and $s_{i_2}$ zero append operations have completed, then $j = l$. Else, $j < l$. The latter case is true since the completion of an append operation guarantees that at least one shard will contain the new record(s). Then, subscribe will necessarily query that shard and thus, return the updated log. Similarly, the only operation that can transform $i$ to $k$ is the trim operation which increments the head of the log. If at least one trim operation with argument $m$ such that $m > i$ has completed between $s_{i_1}$ and $s_{i_2}$, then it follows that $i < k$. If not, then $i = k$. In total, $i \leq k$ and $j \leq l$ which proves both properties.

Property 3 (Append-Visibility). If the execution of an append operation, $s_{i}$, responds before the invocation of a subscribe operation, $s_{j}$, and no conflicting trim operation invoked between them, then the returned sequence of $s_{j}$ includes the record appended by $s_{i}$. Similarly, a read operation $r$ that succeeds $s_{i}$ and reads its returned sequence number, will return a non-$\perp$ value.

Proof. Let $s_{i}$ return $s_n$ and $s_{j}$ return $(W_{m}, ..., W_{l})$. If $s_{i}$ responds before the invocation of $s_{j}$, then the final round of $s_{i}$ must have happened before the first round of $s_{j}$. That means that all correct replicas that participated in $s_{i}$ contain the (last) record identified by $s_n$. Assuming that no trim$(i)$, with $i \geq s_n$, operations invoke between $s_{i}$ and $s_{j}$, it follows that any correct replica that participated in $s_{i}$ and receives the $s_{j}$ request, will reply with a sequence $(W_{m}, ..., W_{l})$ that contains $W_{s_n}$. Since $s_{j}$ responds (terminates), exactly one replica must have done so. In case a conflicting trim operation completes between $s_{i}$ and $s_{j}$, then $W_{s_n} \notin (W_{m}, ..., W_{l})$. The case for a read operation is simply a special case of subscribe.

Theorem 1 (Linearizable color). Consider any arbitrary execution history containing append, read, subscribe and trim operations, acting on an arbitrary (but the same) color. Their properties w.r.t. the sequential specification are maintained and the operations can be ordered w.r.t. their real-time order in this history. Therefore, the color is linearizable.

Corollary 1 (Linearizable shared log). Since linearizability is compositional, and Theorem 1 states that any color is linearizable, it follows that FlexLog is a linearizable shared log object.

Multi-color append protocol. We next extend the correctness proof for our multi-color append protocol.

Proof. Firstly, appends to the special color behave on the exact same way with any other color, thus their properties still hold. Now assume that a client fails at any point between their first append and the end of the final. Since the replicas never receive the special end message, none of the records are appended to any color. If the client fails just after broadcasting the end message, then: if up to $f$ replicas fail, all records will be eventually appended to the respective colors. This is guaranteed by the fact that all participating replicas initiate the normal append protocol (that is already proven). Recall that append operations are idempotent; the client’s tokens uniquely identify the records and replicas that append requests of already seen tokens. Thus, it follows that in any case either all records are appended to the colors or none is. Finally, each append operation initiated by a replica is linearizable in the same way as any other, since it simply operates on a single color.

8 IMPLEMENTATION

FlexLog’s ordering and replication layer are all written in Golang [16]. For simplicity and time efficiency reasons, we implemented only the single-log Append and Read operations.

Our tiered storage layer is developed in C++ on top of PMDK [81]. We used the libpmemobj [80] and TBB [21] where we model the shared log as a concurrent, thread-safe hashmap. The Go code interacts with the storage layer via CGo [8] that creates C-bindings and allows us to reference Go-allocated memory from C++ and vice versa.

For the network communication we use the gRPC [18] library and Google protobufs [29] for message serialization. Specifically, we implement a gRPC server for each sequencer that receives and sends a stream of order-requests and order-responses respectively. The sequencers operate as aggregators and batch the messages.

9 EVALUATION

We evaluate FlexLog across four dimensions: overall throughput and latency compared to the state-of-the-art (§ 9.1), read-write latency (§ 9.2), scalability (§ 9.3) and recovery latencies (§ 9.4).

Experimental Setup. We build and run FlexLog in a cluster of 6 machines, each with a 12-core Intel(R) Xeon(R) Gold 5317 CPU (3.00GHz) with 800 GB Intel Optane DC PM, connected over a 10 Gbps network. We colocate FlexLog’s processes in the machines to maximize CPU utilization. Unless stated otherwise, each shard contains 3 replicas, the record size equals to 1 KB, and, FlexLog is configured to batch order-requests in an interval of 1 µs.

Metrics. We evaluate FlexLog measuring: (i) latency as the average total execution time of a function call of FlexLog-API, (ii)
We conduct an apples-to-apples evaluation of our implementation compared to Paxos [98]. We report the measured latencies for different workload types (Figure 4, left). Results show that our lightweight protocol can achieve significantly lower latencies compared to Paxos. Our flexible ordering semantics achieve 10% better throughput compared to providing total ordering.

Storage layer. Boki is built on top of RocksDB [30], a highly optimized LSM database engine for fast, low latency flash drives (SSDs), as the backend for its storage layer. We compare FlexLog’s storage layer (based on PM) with Boki’s storage layer (based on RocksDB) under workloads with varying record sizes (Figure 5), number of threads (Figure 6) and read-write (R/W) ratios (Figure 7).

RocksDB is configured with a 64 MB in-memory cache (MemTable) and with Write-Ahead-Log (for durability and consistency) enabled. We use db_bench [12] with uniform index distribution.

Results. Figure 5 shows the throughput, measured as operations per second, of Boki’s and FlexLog’s storage layers with different record sizes. First, we observe that FlexLog’s storage layer is an order of magnitude faster than Boki’s. Boki’s limited performance mainly derives from the sync syscalls to synchronize OS’s write buffer with the SSD. In contrast, FlexLog greatly benefits from the low-latency PM, offering the same properties, i.e., data consistency and durability, with better performance. In addition, throughput is stable compared to the record size in both cases.

Figures 6 and 7 show the throughput of Boki and FlexLog storage layers under different workloads with varying number of threads and R/W ratios. Both engines scale well as the number of threads is increased (Figure 6). However, FlexLog achieves steadily higher (∼10x) throughput than Boki. Lastly, read-heavy workloads achieve higher throughput than write-heavy workloads (Figure 7) which is explained by RocksDB’s MemTable and our FlexLog’s cache.

9.1 FlexLog vs State-of-the-art

RQ1. How does FlexLog perform compared to the state-of-the-art? We conduct an apples-to-apples evaluation of our FlexLog storage and ordering layers with Boki [83], a state-of-the-art shared log for serverless computing.

Ordering layer. Boki builds on top of Scalog [62] leveraging its ordering layer that implements the Paxos consensus protocol [98]. Unfortunately, we were not able to run a multi-client deployment of Boki’s ordering layer. That said, we conduct the following two experiments for a fair comparison between Boki/Scalog’s ordering layers [62, 83] and our system. First, we evaluate Boki’s ordering layer against FlexLog’s ordering layer with a single client where we report the measured latencies for different workload types (Figure 4, left). Secondly, we measure the throughput of Paxos [25, 98], Boki’s and Scalog’s ordering layer abstraction, against FlexLog, all in a multi-user setup (Figure 4, right). In all experiments we isolate the ordering layer overheads by executing the workloads without writing any data to the underlying storage layer.

We configure Boki’s ordering layer with 3 sequencers that run Paxos. FlexLog is comprised of a tree of 3 sequencers (root-middle-leaf). Lastly, we run Paxos [98] (libpaxos [25]), with a single proposer. As stated in § 2, running Paxos with concurrent proposers lead to livelocks and huge latencies.

Results. Figure 4 (left) shows the average latency of FlexLog’s ordering layer compared to Boki’s ordering layer for varying workloads. FlexLog achieves less than 250 µs, that is 2.5×—4× faster than Boki. These results are also verified by our second experiment where we compare Paxos with FlexLog. Specifically, Figure 4 (right) shows the throughput, measured as operations per second, of FlexLog’s ordering layer against an optimized version of Paxos.

In addition to FlexLog, we also run a version of our ordering layer (FlexLog-P) that provides partial ordering using a single leaf sequencer. The leaf sequencer is the point of serialization for this particularly colored-log. However, in FlexLog-P the root sequencer is not called upon to enforce a total global ordering. We observe that our lightweight protocol can achieve 2×—3× better throughput compared to Paxos. Our flexible ordering semantics achieve 10% better throughput compared to providing total ordering.

9.2 Latency

RQ2. What is FlexLog’s latency? We measure FlexLog’s append and read protocols’ latency with varying replication factor.

Replication factor. We evaluate FlexLog’s protocols on a setup of 1 shard (with varying replicas) that are all connected to the same (root) sequencer as the minimal ordering layer for linearizability.

Results. Figure 8 shows FlexLog’s read and append latencies under a 95%W/5%R workload. FlexLog’s latencies are reasonably affected by the varying amount of replicas per shard thanks to its replication protocol. Up to 3 replicas the append latency is stable. As the replication factor increases to 4, 6 and 8 (total) replicas, the append latency doubles; a result of the protocol messages that have to be broadcast to more replicas.
Throughput of FlexLog’s vs Boki’s [83] storage layer under different record sizes, number of threads and workloads.

9.3 Scalability

RQ3. How does FlexLog scale? We evaluate the scalability of FlexLog across two parameters: the number of sequencers in the ordering layer and the number of shards.

Number of sequencers. We evaluate FlexLog’s ordering layer for a varying number of leaf-sequencers. Each sequencer batches the order requests in the batching interval (§8) serving as an aggregator for the incoming requests to the root sequencer.

Results. Figure 9 shows the throughput of the ordering layer as more sequencers are added to the tree as leafs and therefore as a proxy to the root sequencer. We observe that a single leaf sequencer can issue approximately 1.2M sequence numbers per second. If we now add more leaf sequencers to the sequencer tree, we can achieve an additional throughput of 1M sequence numbers per second for each leaf sequencer. It is worth noting that through order-request batching, the throughput of a root sequencer is not dependent on the height of the sequencer tree, but rather just on the branching factor. We find that a sequencer can handle up to 10 direct aggregators, sending order-request in an interval of 1µs before the throughput stagnates.

We further measure the latency of an order request (measured as the time difference between sending the request and receiving a sequence number for that request). We found out that the latency is primarily dependant of the RTT of the network rather than the processing necessary for ordering. We measure a latency of about 110µs with a single sequencer. However, as we add more sequencers increasing the height of the sequencer tree, the latency for an order-request is increased linearly with the height of the ordering layer.

Number of shards. We evaluate the scalability of FlexLog w.r.t. the number of shards by deploying a cluster of 6 shards with an ordering layer consisting of a tree of 3 sequencers, for which the leaf sequencers each have 3 shards allocated to them. We conduct the same experiments as for the 3-shard experiment (§9.2), reading and appending to the global log that is ordered by the root sequencer.

Results. As shown in Figure 11, with an example 95%R workload, FlexLog achieves double the throughput with double the amount of shards, indicating linear scalability. This gain in throughput from scaling both the ordering and the data layer is accompanied only by a slightly higher append latency compared to the 3-shard experiment. This append latency is due to the fact that with the scaled ordering layer, the sequencer tree now has the depth of one, which adds more latency to the ordering protocol (as described in § 9.3). The read latency to the ordering protocol (as described in § 9.3). The read latency...
latency on the other hand is not affected by scaling the data layer, and behaves similar to the read latency of the experiment with 3 shards.

RQ3 takeaway. FlexLog scales linearly with the number of sequencers (ordering layer) and number of shards.

9.4 Recovery

RQ4. How fast can FlexLog recover? We evaluate the recovery latency of FlexLog’s nodes; how fast can a node get up to date on all records that have been committed during its downtime before being able to participate in the protocol and serve requests.

Replica recovery. For this experiment, we use an artificial micro-benchmark that reads the records from the log and, then, applies them to a second file in PM.

Results. Figure 10 shows the latency of the recovery process (of a single replica). We see that recovery time is heavily dependent on the number of committed records during the downtime. As expected, the recovery latency grows almost linearly as a function of the number of records to recover, as a result of reading all records that have to be recovered in a sequential manner.

RQ4 takeaway. FlexLog’s recovery latency grows linearly with the number of records to be recovered.

10 RELATED WORK

Shared logs. The shared log abstraction is well studied in the literature. Corfu [47], a widely-adopted shared log, is one of the first to build on the then-new flash storage units, SSDs. Corfu separates ordering from replication but, unfortunately, Corfu’s single-node sequencer quickly becomes the bottleneck. Scallog [62] aims to improve this bottleneck by replacing Corfu’s centralized sequencer with a replicated counter based on Paxos [98]. It also introduces a tree of aggregators as an optimization that reduces the number of connections in cases where a large number of shards are deployed.

Towards the same direction, Kafka [128] exposes scalability but, in contrast to previous systems, it only provides linearizability within a shard (and not global total ordering). Lastly, FuzzyLog [107] further relaxes the consistency guarantees offering partial ordering semantics, essentially, by capturing happens-before relationships between conflicting operations.

The FlexLog ordering layer exposes similar (strong consistency) semantics to Scallog, that is, linearizable reads. However, Scallog issues new sequence numbers for every order-request using Paxos, while FlexLog decides on an epoch number, only when a sequencer node crashes. In other words, FlexLog combines the simplicity of Corfu’s single sequencer (per color) with Scallog’s tree of aggregators but in a region-tree-structure fashion, enabling locality-aware ordering semantics. In contrast to all those systems, FlexLog extends the conventional append operation with atomic multi-record appends to multiple logs. In FlexLog we leverage the near-DRAM latency of PM by simplifying yet optimizing the ordering and replication of records, making it able to scale for a large number of shards. FlexLog’s protocol enables concurrent appends and local linearizable reads.

Serverless computing frameworks. State management remains a challenge in (stateful) serverless computing [73, 117]. Unfortunately, recent attempts from industry [13, 35] are in very early stages thus recognizing limited adoption.

Prior research efforts on this direction [97, 121, 122] expose a limited Put/Get interface for functions to manage state and have different focus, e.g., heterogeneous storage technology [97], light-weight isolation [121], and auto-scaling [122]. Cloudburst [122] uses Anna [131], an autoscaling KV store for state sharing combined with caches co-located with the functions. Pocket [97] is a distributed data store targeted at the ephemeral data used by serverless functions to share state. Pocket uses multiple tiers (e.g., DRAM, flash, disk) and provides an elastic and cost-effective storage solution.

The state-of-the-art for state management in serverless is Boki [83]; the first attempt to study serverless state management leveraging the shared log API. Boki’s approach is motivated by the fault-tolerance and consistency challenges encountered by stateful serverless applications, which the Put/Get interface might not be able to easily address—indeed recent work [52] argues that 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. Boki partly adopts Scallog’s ordering layer and introduces the metalog, a component that combines ordering, read consistency and fault tolerance.

Similarly to Boki, FlexLog realises the synergy between the shared log abstraction and stateful serverless. However, we go one step further realizing that prominent storage technologies, like PM, in combination with a stateless, scalable ordering layer with flexible semantics can benefit even more serverless environments.

PM systems. PM systems are actively researched in various domains, such as file systems [58, 88, 127, 134, 141], KV-stores [55, 87, 100, 133, 139], crash consistency & reliability [57, 112, 116, 136, 137, 140] and testing tools [104–106]. Well-known data management systems [37–39, 79, 94] have already integrated PM in their system stack. FlexLog leverages PM to offer low-latency storage access which especially benefits short-lived serverless functions [119].

11 CONCLUSION

The shared log abstraction offers a suitable solution for serverless applications that require a fast and fault-tolerant shared data plane. In this paper we present FlexLog, a shared log system carefully designed for serverless applications, that combines a data layer on persistent memory along with a scalable ordering layer that exposes flexible ordering semantics. Our evaluation on a cluster of 6 machines equipped with 800 GB Intel Optane DC PM shows that both FlexLog’s ordering and data layer scale almost linearly while preserving minimal latency thanks to leveraging persistent memory as a storage medium. Compared to the state-of-the-art shared log systems for serverless, FlexLog achieves higher performance in both the ordering and storage layers.

Software artifact. FlexLog’s code is publicly available: https://github.com/TUM-DSE/FlexLog.

Acknowledgements. We thank our shepherd and the reviewers for their insightful comments. We also thank Max Wiesholler and Dimitris Stavrakakis. This work was supported in parts by a Microsoft Research PhD Fellowship, a UK RISE Grant and an ERC Starting Grant.

REFERENCES


FLEXLog: A Shared Log for Stateful Serverless Computing

HPDC ’23, June 16–23, 2023, Orlando, FL, USA