FlexLog: A Shared Log for Stateful Serverless Computing

Dimitra Giantsidi
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

ACM Reference Format:
Dimitra Giantsidi, Emmanouil Giortamis, Nathaniel Tornow, Florin Dinu, and Pramod Bhatotia. 2023. FlexLog: A Shared Log for Stateful Serverless Computing. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23), June 16–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3588195.3592993

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

ACM ISBN 979-8-4007-0155-9/23/06…$15.00
https://doi.org/10.1145/3588195.3592993

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—4× 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

syscall	Video processing	Gzip compression
open()	17%	19%
read()	15%	3.2%
write()	N/A	22%
fstat()	5.1%	2.9%
close()	6%	1%
Total	41%	48.1%

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

10 s [123]—while they frequently need to access the storage persisting data and communicating with each other [52, 73, 117, 122, 138].

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—from 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.
- **Flexiblity** 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], DynamoDB [60], etc.) to manage their state, exchange data between the executing functions and persist data. Unfortunately, such approaches cannot simultaneously provide low latency, low cost, and high throughput [96]. At the same time, these systems only offer Put/Get interfaces which make quite challenging maintaining data consistency and isolation (under failures) in stateful transactional serverless workloads [135].

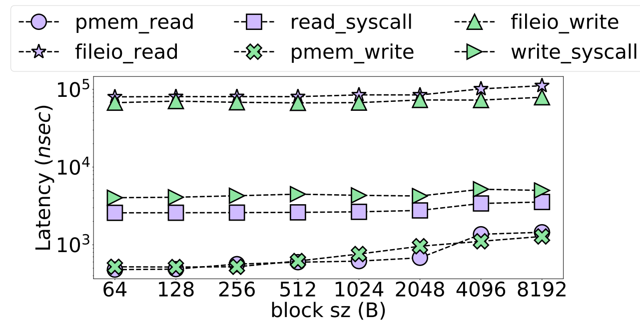

Figure 1: Storage latency for `read` and `write` operations.

In line with prior work [83], we advocate that shared logs can benefit serverless functions because, by design, they provide fault tolerance and can address serverless functions' performance, scalability and consistency requirements [83]. Indeed, considering the trends in serverless computing [15, 92, 93], we found several use cases where a shared log can provide an efficient solution. A shared log can be used for inter-process communication (building serverless message queues [31, 32]). In addition, shared logs can serve as the building block for designing high-level data structures, e.g., Durable Objects [35], that are durable, scalable and consistent [48] because they hide a consensus protocol behind their API. Lastly, the database community has shown how to use logs [40] to implement transactions which are fundamental for fault-tolerant workflows [135].

Unfortunately, existing shared log systems [47, 48, 62, 107] are ill-suited for modern serverless applications because they all build on high-latency storage technologies (SSDs) and primarily design protocols for providing the rather expensive total ordering. For example, Corfu [47] implements a replicated data layer in a chained topology and a single-node sequencer as its ordering layer that, unfortunately, has been proven to become a bottleneck [62, 107]. On the other hand, Fuzzylog [107] targets performance in geo-distributed deployments by relaxing the ordering consistency guarantees. However, FuzzyLog replicates each partition via chain replication which increases latency. Lastly, the state-of-the-art Scalog [62] implements a Paxos [98]-based counter service (as its ordering layer) and implements a totally-ordered log. However, researchers [110] have shown that even well-studied Paxos implementations can have large performance variations (even with mild workloads) that cascade and create insufficient application-level performance. To this end, there is a need for a system that does not sacrifice performance while still providing serverless functions with the consistency guarantees they need.

3.3 Bottleneck Analysis in Shared Log Systems

We conduct an empirical evaluation of the state-of-the-art shared log Scalog [62], whose architecture has recently been adopted by Boki [83], a shared log for serverless computing. Our analysis shows that both its storage and ordering layers can limit performance, 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, $(W_h, ..., W_t)$. 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, $\{Shard\#1, \ldots, Shard\#n\}$. 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 $l \in \mathbb{N}^+$ colors, $L = \{Color\#0, \ldots, Color\#l\}$. Lastly, we consider a set O of $o \in \mathbb{N}^+$ nodes, $O = \{Seq\#0, \ldots, Seq\#o\}$ 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.

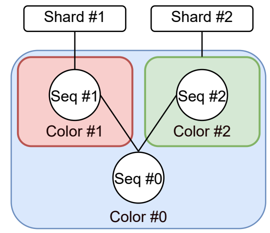

Figure 2: FlexLog's abstraction; the data layer (*Shard* #1, *Shard* #2) is organized into regions of totally-ordered logs (blue, red, green) each of which is managed by a single sequencer node (*Seq* #0, *Seq* #1, *Seq* #2).

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 Δ 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 iff 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 $2f$ backups where at most f nodes can crash. Failures (e.g., crashes, network partitions, etc.) are identified by noticing message delays greater than Δ.

Append(r[ ], c)	Appends records to the log of color c and returns the last assigned SN
Read(SN, c)	Reads a record with SN from the c-colored log
Subscribe(c)	Receives all records of the c-colored log
Trim(SN, c)	Garbage collects the log of color c by deleting all records with sn ≤ SN
AddColor(c, c_p)	Creates a new c-colored log with c_p log as its parent

Table 2: FlexLog's basic API.

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 1. A request is then routed to the orchestrator which keeps track of the entire cluster resource utilization 2. The orchestrator communicates with the workers' manager which chooses a physical host to launch the new function instance 3[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 Appends 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 r 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.

[1]We make no assumption about FlexLog's (physical) location. The provider's orchestrator can co-locate FaaS applications with FlexLog's component(s) based on resource usage.

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.

```
1 type MessageQueue struct { // A message queue defined by a color
2     color   Color,  handle   FlexLog
3 }
4 // Enqueue method to distribute record to serverless applications
5 func (mq *MessageQueue) Enqueue(idx int, record String) {
6     mq.handle.Append({idx, record}, mq.color)
7 }
8 // Get method that returns the index and the record of an index (idx)
9 func (mq *MessageQueue) Get(idx int) {int, Record} {
10     return mq.handle.Read(idx, mq.color);
11 }
12 // Lookup of the expected record (exp_record) in the message queue
13 func (mq *MessageQueue) getIdx(exp_record String) int {
14     log := mq.handle.Subscribe(mq.color)
15     for (idx, record := <- log) { // Iterate the next entry
16         if (record == exp_record) { return idx; }
17     }
18     return -1;
19 }
20 // Append to yellow color the data to be read from Func2
21 func (mq *MessageQueue) Func1() {
22     sn_y := handle.Append(record[], yellow)
23     // Create the black log
24     mq.color := mq.handle.AddColor(black)
25     mq.Enqueue(sn_y, String{YELLOW_READ_IDX})
26 }
27 func (mq *MessageQueue) Func2() {
28     for (;;) { // Lookup until the entry (s_idx) is found
29         s_idx := mq.getIdx(String{YELLOW_READ_IDX})
30         if (s_idx != -1) {break;}
31     }
32 }
```

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

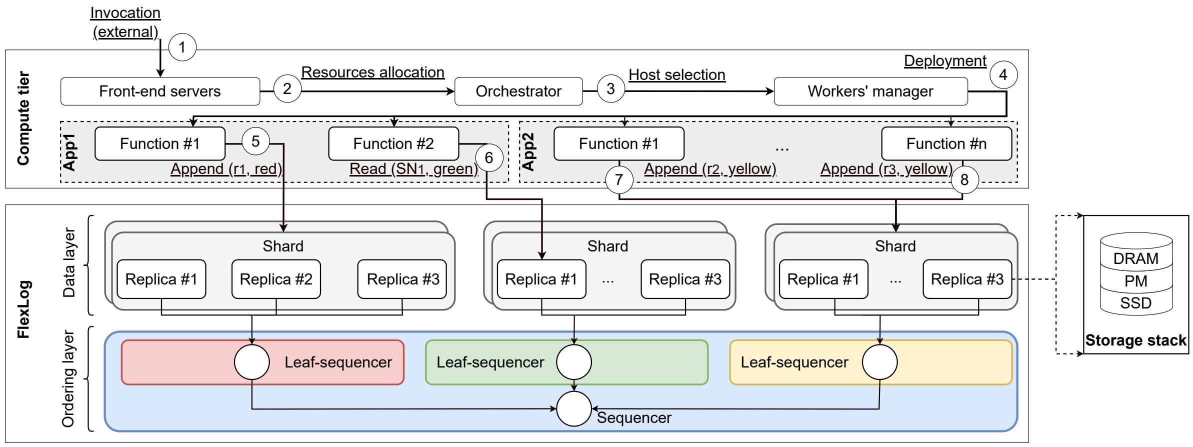

Figure 3: FlexLog system architecture.

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 (w/ 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 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 local 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` (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 `Append`$(r[\,], c)$ 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 (Alg1:6). Then, the request is broadcast to all replicas of a randomly-chosen shard of c (Alg1:7). 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 an 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-sequencer 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 `Read`(*sn*,c) of FlexLog-API to read the record stored with SN *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 $\perp$ value. Since each record is stored only on one shard, only one of the replicas can return the record. If all replicas return $\perp$, then the shared log of color c has no record with SN *sn* stored.

6.2 Auxiliary System Operation Protocols

Subscribe protocol. Applications invoke `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 (0,*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.

Algorithm 1: Append protocols

```
1  client_append(record, color)
2  // Append protocol for a caller with id = FID
3  Initially, counter := 0
4  function Append (records[], c)
5  begin
6      counter ⟵ counter + 1; t ⟵ (FID « 32) + counter
7      broadcast(records[], c, t) to all replicas in shard
8      wait(t, sn) from all replicas in shard
9      return sn
10 end
11
12 // Append protocol for replica nodes
13 Initially, tokens := ∅
14 upon receiving (records[], c, t) do //from client
15 begin
16     if t ∈ tokens then return
17     persist(records[], t); tokens ⟵ tokens + t
18     shard ⟵ ids_of_replicas_in_shard()
19     send(c, t, |records[]|, shard) to sequencer_node
20 end
21
22 upon receiving (t, sn) do //from sequencer
23 begin
24     commit_all(t, sn); send(t, sn) to FID (ACK)
25 end
26
27 // Append protocol for sequencer node with id = SID
28 Initially, sn := 0, counter := 0, tokens := ∅
29 upon receiving (c, t, nrecords, shard) do
30 begin
31     if t ∈ tokens then return
32     if is_root(SID, c) then
33         tokens ⟵ tokens + t; counter ⟵ counter + nrecords
34         sn ⟵ (epoch « 32) + counter
35         broadcast(t, sn) to shard
36     else
37         send(c, t, nrecords, shard) to parent in sequencer tree
38     end
39 end
```

Finally, the protocol reconstructs the log of color c by sorting all records received based on their SNs.

Trim protocol. Applications invoke `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 [*head*,*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 ack 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, appends 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 on-going (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[2]. 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-sequencer 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*.

[2]A 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].

Algorithm 2: Atomic Multi-color append protocol

```
1  function Multi_Append (records[][], colors[])
2  begin
3      shard ← get_random_shard(special_color)
4      for i ← 0 to |records[]| do append(records[i]:colors[i]:ID, special_color)
5      broadcast(end) to all replicas in shard
6      wait(ack) from any replica in shard
7  end
8
9  // Multi-color append protocol for replica nodes
10 upon receiving (end) do
11 begin
12     records[][], colors[], tokens[] ← read_records(FID)
13     for i ← 0 to |records[]| do
14         shard ← get_random_shard(colors[i])
15         broadcast(records[i][], colors[i], tokens[i]) to all replicas in shard
16         wait(token, sn) from all replicas in shard
17     end
18     send(ack) to CID
19 end
```

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 $r(i,c)$ and $r(j,c)$ to return values $= \perp$ and $\neq \perp$, 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.

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 by 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_3 that is a common consecutive subsequence (substring) of s_1 and s_2.*

Property 2 (Stability). *Consider two successive $subscribe$ operations, sub_1 and sub_2, such that sub_1 responds before the invocation of sub_2. Let s_1 and s_2 be the sequences returned by sub_1 and sub_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 sub_1 and sub_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 sub_1 and sub_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, app, responds before the invocation of a subscribe operation, sub, and no conflicting $trim$ operation invoked between them, then the returned sequence of sub includes the record appended by app. Similarly, a read operation r that succeeds app and reads its returned sequence number, will return a non-⊥ value.*

Proof. Let *app* return *sn* and *sub* return $(W_h,...,W_t)$. If *app* responds before the invocation of *sub*, then the final round of *app* must have happened before the first round of *sub*. That means that all correct replicas that participated in *app* contain the (last) record identified by *sn*. Assuming that no $trim(i)$, with $i \geq sn$, operations invoke between *app* and *sub*, it follows that any correct replica that participated in *app* and receives the *sub* request, will reply with a sequence $(W_h,...,W_t)$ that contains W_{sn}. Since *sub* responds (terminates), exactly one replica must have done so. In case a conflicting $trim$ operation completes between *app* and *sub*, then $W_{sn} \notin (W_h,...,W_t)$. 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 drop append requests for 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 collocate 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)*

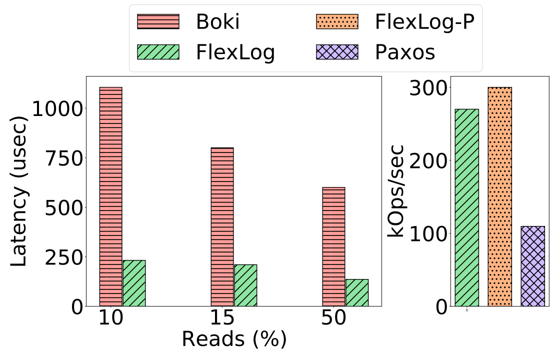

Figure 4: Latencies and throughput of FlexLog' ordering layer compared to Boki [83]'s and compared to Paxos [98].

throughput as the number of completed function calls (operations) per second, and *(iii)* scalability, as FlexLog's ability to handle increasing throughput with *reasonable* latency.

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.

Storage layer. Boki is build 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 (> 10×) 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.

RQ1.1 takeaway. FlexLog achieves up to 10× and 3× higher throughput (operations per second) for its storage and ordering layers, respectively, w.r.t. the state-of-the-art Boki [83]. In addition weaker ordering improves performance by 10% w.r.t. total ordering.

While we minimize both the (mean) latencies for storage and ordering, the ordering layer latency dominates the operations' latency. We found that in two workloads (10%R and 50%R) when the function access the local storage (co-located with the storage node), the storage latency is 1 us whereas the ordering latency is 250 us and 100 us respectively. As a result, the use of the fastest storage shifts the bottleneck even more to the ordering layer (in appends). Reads only do storage accesses and do not incur overheads due to ordering.

RQ1.2 takeaway. The performance gains come from the fast ordering protocol (in appends) rather than the use of PM.

9.2 Latency

RQ2. *What is FlexLog's latencies?* 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.

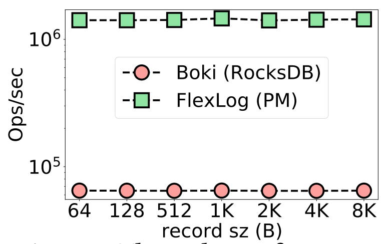

Figure 5: Throughput of storage layers with various record sizes.

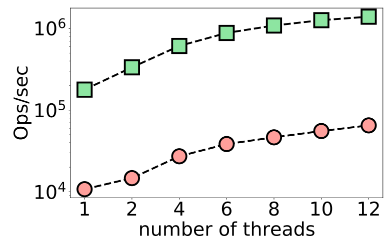

Figure 6: Throughput of storage layers with various number of threads.

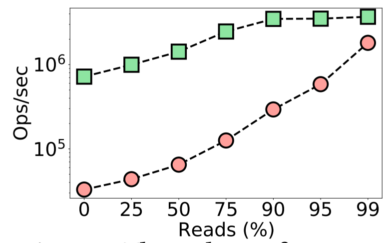

Figure 7: Throughput of storage layers for various workloads (R/W ratios).

Throughput of FlexLog's vs Boki's [83] storage layer under different record sizes, number of threads and workloads.

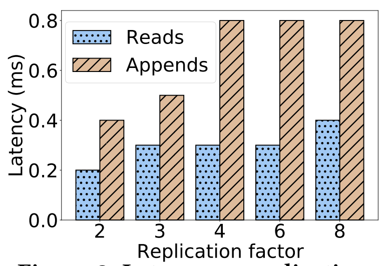

Figure 8: Latency vs replication factor on a shard.

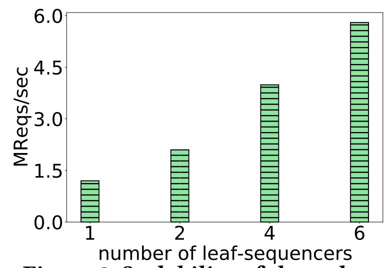

Figure 9: Scalability of the ordering layer.

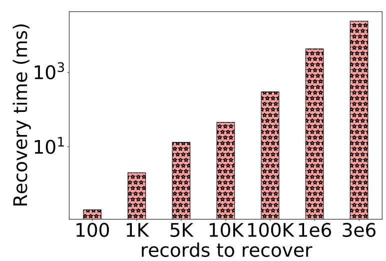

Figure 10: Recovery time vs records to recover.

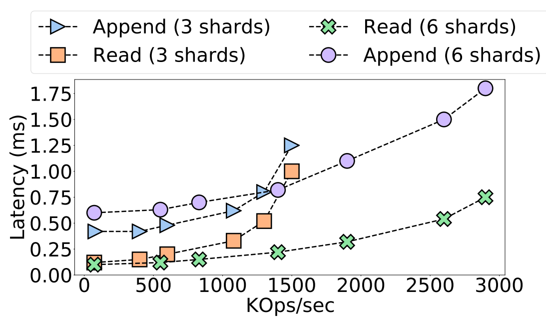

Figure 11: Latency vs throughput for an increasing number of shards [95% read, 5% append workload].

Our findings are reasonable; first the read latency remains stable or is increased marginally with the number of replicas. This result is justified by FlexLog's read protocol which allows for local reads on a shard's replicas, keeping latency to a minimum. Secondly, FlexLog achieves low append-latency when executing an-almost-write-only workload (5%R). This is achieved by FlexLog's applications-centric replication protocol where applications broadcast the records directly to all replica of each shard.

RQ2 takeaway. FlexLog achieves low append latencies while it also offers stable read latency w.r.t. the replication factor.

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 $1\mu s$ 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 $110\mu s$ 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 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. Scalog [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 Scalog, that is, linearizable reads. However, Scalog 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 Scalog'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 *append*s and local linearizable *read*s.

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 Scalog'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 filesytems [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

[1] Ad hoc big data processing made simple with serverless mapreduce. https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-

simple-with-serverless-mapreduce/. Last accessed: May 24, 2023.
[2] Apache kafka streams. https://kafka.apache.org/documentation/streams/. Accessed: 2021-02-04.
[3] Apache kafka website. https://kafka.apache.org/. Accessed: 2021-10-12.
[4] Aws lambda. https://aws.amazon.com/lambda/. Accessed: May 24, 2023.
[5] Azure functions. https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview. Accessed: May 24, 2023.
[6] A brief analysis of consensus protocol: From logical clock to raft. https://www.alibabacloud.com/blog/a-brief-analysis-of-consensus-protocol-from-logical-clock-to-raft_594675. Accessed: May 24, 2023.
[7] Causal consistency. https://mariadb.org/causal-consistency/. Accessed: May 24, 2023.
[8] Cgo command. https://pkg.go.dev/cmd/cgo. Accessed: 2021-10-12.
[9] Consistency and replication model. https://docs.hazelcast.com/imdg/4.2/consistency-and-replication/consistency. Accessed: May 24, 2023.
[10] CorfuDB. https://github.com/corfudb. Last accessed: May 24, 2023.
[11] Couchbase. https://www.couchbase.com/. Last accessed: May 2021.
[12] db_bench. https://github.com/EighteenZi/rocksdb_wiki/blob/master/Benchmarking-tools.md. Accessed: May 24, 2023.
[13] Entity functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities?tabs=csharp. Accessed: May 24, 2023.
[14] etcd. https://etcd.io/. Last accessed: May 24, 2023.
[15] Functionbench. https://github.com/ddps-lab/serverless-faas-workbench. Accessed: May 24, 2023.
[16] Golang. https://golang.org/. Accessed: 2021-07-10.
[17] Google functions. https://cloud.google.com/functions. Accessed: May 24, 2023.
[18] grpc. https://grpc.io/. Accessed: 2021-07-10.
[19] How cxl may change the datacenter as we know it. https://www.theregister.com/2022/05/16/cxl_datacenter_memory/. Accessed: May 24, 2023.
[20] Intel sees cxl as rack-level disaggregator with optane connectivity. https://blocksandfiles.com/2021/08/18/intel-sees-cxl-as-rack-level-disaggregator/. Accessed: May 24, 2023.
[21] Intel threading building blocks. https://github.com/oneapi-src/oneTBB. Accessed: 2021-10-12.
[22] Introducing zelos: A zookeeper api leveraging delos. https://engineering.fb.com/2022/06/08/developer-tools/zelos/. Accessed: May 24, 2023.
[23] Is intel's optane technology really dead? https://gestaltit.com/checksum/stephen/is-intels-optane-technology-really-dead-checksum-episode-21/. Accessed: May 24, 2023.
[24] knative. https://knative.dev/docs/. Accessed: May 24, 2023.
[25] libpaxos. https://libpaxos.sourceforge.net/. Accessed: May 24, 2023.
[26] Logdevice: Distributed storage for sequential data. https://logdevice.io/. Accessed: 2021-10-12.
[27] pmem-rocksdb. https://github.com/pmem/pmem-rocksdb. Last accessed: May 24, 2023.
[28] Pravega – a reliable stream storage system. https://pravega.io/#. Accessed: 2021-10-12.
[29] Protocol buffers. https://developers.google.com/protocol-buffers. Last accessed: May 24, 2023.
[30] Rocksdb. https://github.com/facebook/rocksdb. Accessed: May 24, 2023.
[31] Serverless message queues and how to leverage the best. https://dzone.com/articles/serverless-message-queues-and-how-to-leverage-the. Accessed: May 24, 2023.
[32] Sqs queues. https://www.serverless.com/framework/docs/providers/aws/events/sqs. Accessed: May 24, 2023.
[33] vhive. https://vhive-serverless.github.io/. Accessed: May 24, 2023.
[34] Why intel killed its optane memory business. https://www.theregister.com/2022/07/29/intel_optane_memory_dead/. Accessed: May 24, 2023.
[35] Workers durable objects beta: A new approach to stateful serverless. https://blog.cloudflare.com/introducing-workers-durable-objects/. Accessed: May 24, 2023.
[36] Zookeeper internals. https://zookeeper.apache.org/doc/r3.4.13/zookeeperInternals.html.
[37] Memhive: Scale applications with persistent memory! https://www.memhive.io/, 2021. Accessed 27-09-2021.
[38] Pmem-Redis. https://github.com/pmem/pmem-redis, 2021. Accessed 27-09-2021.
[39] pmem-rocksdb. https://github.com/pmem/pmem-rocksdb, 2021. Accessed 27-09-2021.
[40] The transaction log. https://learn.microsoft.com/en-us/sql/relational-databases/logs/the-transaction-log-sql-server?view=sql-server-ver16, 2021. Accessed: May 24, 2023.
[41] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.
[42] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. pages 85–98, 04 2013.
[43] Amazon. Amazon MemoryDB for Redis. https://aws.amazon.com/memorydb/?p=ft&c=db&z=3. Last accessed: May 24, 2023.
[44] Amazon. Amazon S3 Cloud Object Storage. https://aws.amazon.com/s3. Last accessed: Dec, 2018.
[45] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 263–274, New York, NY, USA, 2018. Association for Computing Machinery.
[46] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual consensus in delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 617–632. USENIX Association, November 2020.
[47] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. CORFU: A shared log design for flash clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 1–14, San Jose, CA, April 2012. USENIX Association.
[48] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 325–340, New York, NY, USA, 2013. Association for Computing Machinery.
[49] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, Jingming Liu, Filip Gruszczynski, Jun Li, Rounak Tibrewal, Ali Zaveri, Rajeev Nagar, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Log-structured protocols in delos. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 538–552, New York, NY, USA, 2021. Association for Computing Machinery.
[50] R. Baldoni, R. Beraldi, and R. Prakash. Flexible general purpose communication primitives for distributed systems. In *Proceedings. The Sixth IEEE International Symposium on High Performance Distributed Computing (Cat. No.97TB100183)*, pages 201–210, 1997.
[51] Eric A. Brewer. Towards robust distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*, 2000.
[52] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. Durable functions: Semantics for stateful serverless. In *OOPSLA*, pages 133:1–133:27. ACM, October 2021.
[53] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.
[54] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 13–24, New York, NY, USA, 2019. Association for Computing Machinery.
[55] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.
[56] Brian Choi, Randal Burns, and Peng Huang. *Understanding and Dealing with Hard Faults in Persistent Memory Systems*, page 441–457. Association for Computing Machinery, New York, NY, USA, 2021.
[57] Brian Choi, Randal Burns, and Peng Huang. Understanding and dealing with hard faults in persistent memory systems. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 441–457, New York, NY, USA, 2021. Association for Computing Machinery.
[58] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 133–146, New York, NY, USA, 2009. Association for Computing Machinery.
[59] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4):1726–1738, 2020.
[60] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review (SIGOPS)*, 2007.
[61] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, dec 2004.
[62] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 325–338, Santa Clara, CA, February 2020. USENIX Association.
[63] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises:

Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 54–70, New York, NY, USA, 2015. Association for Computing Machinery.
[64] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, apr 1988.
[65] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.
[66] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association.
[67] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.
[68] Google. Cloud Storage. http://www.cloud.google.com/storage, 2017. Last accessed: Dec, 2018.
[69] Google. Google Cloud Functions. https://cloud.google.com/functions, 2017. Last accessed: May 24, 2023.
[70] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, page 202–210, New York, NY, USA, 1989. Association for Computing Machinery.
[71] G. Guerraoui and A. Schiper. A generic multicast primitive to support transactions on replicated objects in distributed systems. In *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 334–342, 1995.
[72] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. uKharon: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 101–120, Carlsbad, CA, July 2022. USENIX Association.
[73] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back, 2018.
[74] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
[75] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. PASTE: A network programming interface for non-volatile main memory. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 17–33, Renton, WA, 2018.
[76] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*.
[77] Intel. Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2021.
[78] Intel. Persistent Memory Development Kit. https://pmem.io/pmdk/, 2021.
[79] Intel. Persistent Memory Development Kit : pmemkv, 2021. Accessed 27-09-2021.
[80] Intel. Persistent Memory Development Kit : The C++ bindings to libpmemobj, 2021. Accessed 27-09-2021.
[81] Intel. PMDK: libpmemobj examples. https://github.com/pmem/pmdk/tree/master/src/examples/libpmemobj, 2021.
[82] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 425–438, Santa Clara, CA, March 2016. USENIX Association.
[83] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery.
[84] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery.
[85] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 857–871, New York, NY, USA, 2021. Association for Computing Machinery.
[86] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99 In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.
[87] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. Slm-db: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, Boston, MA, February 2019. USENIX Association.
[88] Chandan Kalita, Gautam Barua, and Priya Sehgal. Durablefs: A file system for persistent memory. *CoRR*, abs/1811.00757, 2018.
[89] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
[90] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 201–217, New York, NY, USA, 2020. Association for Computing Machinery.
[91] The Linux kernel archives. DAX - Direct access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt, 2021.
[92] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504, 2019.
[93] Jeongchul Kim and Kyungyong Lee. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 477, New York, NY, USA, 2019. Association for Computing Machinery.
[94] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead logs and persistent SkipLists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 161–177, Carlsbad, CA, July 2022. USENIX Association.
[95] Martin Kleppmann and Jay Kreps. Kafka, samza and the unix philosophy of distributed data. 12 2015.
[96] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 789–794, Boston, MA, July 2018. USENIX Association.
[97] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.
[98] Leslie Lamport. *The Part-Time Parliament*, page 277–317. Association for Computing Machinery, New York, NY, USA, 2019.
[99] Long Hoang Le, Mojtaba Eslahi-Kelorazi, Paulo Coelho, and Fernando Pedone. Ramcast: Rdma-based atomic multicast. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 172–184, New York, NY, USA, 2021. Association for Computing Machinery.
[100] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.
[101] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 104–120, New York, NY, USA, 2017. Association for Computing Machinery.
[102] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 467–483, Savannah, GA, November 2016. USENIX Association.
[103] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.
[104] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. Pmfuzz: Test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 487–502, New York, NY, USA, 2021. ACM.
[105] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1187–1202, New York, NY, USA, 2020. Association for Computing Machinery.
[106] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 411–425, New York, NY, USA, 2019. Association for Computing Machinery.

[107] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. The fuzzylog: A partially ordered shared log. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 357–372, Carlsbad, CA, October 2018. USENIX Association.

[108] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301. USENIX Association, July 2021.

[109] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[110] Parisa Jalili Marandi, Samuel Benz, Fernando Pedone, and Kenneth P. Birman. The performance of paxos in the cloud. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 41–50, 2014.

[111] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. Chariots: A scalable shared log for data management in multi-datacenter cloud environments. In Gustavo Alonso, Floris Geerts, Lucian Popa, Pablo Barceló, Jens Teubner, Martín Ugarte, Jan Van den Bussche, and Jan Paredaens, editors, *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*, pages 13–24. OpenProceedings.org, 2015.

[112] Ian Neal, Andrew Quinn, and Baris Kasikci. Hippocrates: Healing persistent memory bugs without doing any harm. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 401–414, New York, NY, USA, 2021. Association for Computing Machinery.

[113] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 107–118, New York, NY, USA, 2015. Association for Computing Machinery.

[114] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 43–57, Oakland, CA, May 2015. USENIX Association.

[115] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, February 2019. USENIX Association.

[116] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutiu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 672–685, 2015.

[117] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM*, 64(5):76–84, apr 2021.

[118] Srinath Setty, Chunzhi Su, Jacob R. Lorch, Lidong Zhou, Hao Chen, Parveen Patel, and Jinglei Ren. Realizing the Fault-Tolerance promise of cloud storage using locks with intent. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 501–516, Savannah, GA, November 2016. USENIX Association.

[119] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[120] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.

[121] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.

[122] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, jul 2020.

[123] Dmitrii Ustiugov. *Data-centric Serverless Cloud Architecture*. PhD thesis, 2022.

[124] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3), feb 2015.

[125] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design amp; Implementation - Volume 6*, OSDI'04, page 7, USA, 2004. USENIX Association.

[126] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD 2017*, 2017.

[127] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[128] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with apache kafka. *Proc. VLDB Endow.*, 8(12):1654–1655, August 2015.

[129] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. vcorfu: A cloud-scale object store on a shared log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 35–49, Boston, MA, March 2017. USENIX Association.

[130] Xueliang Wei, Dan Feng, Wei Tong, Jingning LIU, and Liuqing Ye. Nico: Reducing software-transparent crash consistency cost for persistent memory. *IEEE Transactions on Computers*, 68(9):1313–1324, 2019.

[131] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. Anna: A kvs for any scale. *IEEE Trans. on Knowl. and Data Eng.*, 33(2):344–358, feb 2021.

[132] Yinjun Wu, Kwanghyun Park, Rathijit Sen, Brian Kroth, and Jaeyoung Do. Lessons learned from the early performance evaluation of intel optane dc persistent memory in dbms. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN '20, New York, NY, USA, 2020. Association for Computing Machinery.

[133] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, Santa Clara, CA, July 2017. USENIX Association.

[134] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[135] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204. USENIX Association, November 2020.

[136] L. Zhang. Building reliable software for persistent memory. 2019.

[137] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 897–911, USA, 2019. USENIX Association.

[138] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 1–12, New York, NY, USA, 2019. Association for Computing Machinery.

[139] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: A key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 194–209, New York, NY, USA, 2021. Association for Computing Machinery.

[140] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. *SIGARCH Comput. Archit. News*, 43(1):3–18, March 2015.

[141] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM performance with opportunistic delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 179–193, Carlsbad, CA, July 2022. USENIX Association.