The Dissertation Committee for Zhipeng Jia Certifies
that this is the approved version of the following dissertation:

Designing Systems for Emerging Serverless Applications

Committee:

Emmett Witchel, Supervisor

Simon Peter

Christopher J. Rossbach

Mahesh Balakrishnan

Jason Flinn

Designing Systems for Emerging Serverless Applications

by

Zhipeng Jia

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2022

To my grandparents: unfortunately they cannot witness my achievement

Acknowledgements

Pursuing a doctoral degree is for sure the most challenging thing I decided to do in my twenties. I have to admit, at the beginning, I underestimate the difficulty of such journey. When I am close to the destination, I truly realize this achievement is much more than my own's effort.

Without doubt, my Ph.D. supervisor, Emmett Witchel, is at the top of who I would like to express my gratitude. Essentially, Emmett taught me how to do systems research: he guided me to think critically for my research projects; he helped me to improve my paper writing and conference presentations; and most importantly, he inspired me about the nature of good research.

Beside my advisor, members of my Ph.D. committee are the same important to make this dissertation happen. I would like to thank them: Christopher J. Rossbach, Simon Peter, Jason Flinn, and Mahesh Balakrishnan. Thanks for all the time and effort you spent with my dissertation! Other than the dissertation, several of them also guided me at different stages of my graduate school life. Chris encouraged and inspired me many times going back to his Advanced Operating Systems course. Mahesh shared with me many advice on career choice. I would also like to thank James Bornholt for being committee member of my Research Preparation Exam.

Although the journey towards Ph.D. is not easy, my five-year graduate school life is full of joy with my friends at Austin. I would like to thank members from OSA research group: Zhiting Zhu, Yige Hu, and Tyler Hunt. We spent great time together discussing research problems and life issues. I would also like to thank Vance Miller and Ariel Szekely, for working with our group on GPU security project. There are other fellow graduate students at UT Austin I would like to thank for their time spent with me: Zhen Chen, Zeyuan Hu, Xinrui Hua, Xiangru Huang, Jiahui Liu, Zhan Shi, Jiaru Song, Wei Sun, Yifan Sun, Yingchen Wang, Lemeng Wu, Hangchen Yu, Yuxuan Zhang, Xingyi Zhou, and Jiacheng Zhuo.

Finally, I would like to thank my parents for supporting me, both emotionally and financially. In my late-twenties, I start to realize family members are ones who will listen to all kinds of my life complaints. It is a pity that in the past three years, due to global COVID pandemic, I had no chance to meet with them in person.

贾志鹏 (Zhipeng Jia)

The University of Texas at Austin

May, 2022

Abstract

Designing Systems for Emerging Serverless Applications

Zhipeng Jia, Ph.D.
The University of Texas at Austin, 2022

Supervisor: Emmett Witchel

Serverless computing has become increasingly popular for building scalable cloud applications. Its function-as-a-service (FaaS) paradigm allows users to upload cloud functions that are executed on a provider's infrastructure. Serverless infrastructure is highly elastic; users can smoothly invoke thousands of concurrent functions in the cloud. As a result, serverless computing has gained success in massively parallel workloads such as video processing, data analytics, machine learning, and distributed compilation.

However, there are emerging cloud applications that the current serverless infrastructure cannot support with efficiency and high performance. We identify two issues in the current infrastructure preventing serverless computing to support broader applications. First, invocation latencies of serverless functions are too high for latency-sensitive workloads, for example, interactive microservices. Second, fault tolerance is very difficult for stateful serverless applications, because serverless functions can fail during their execution while leaving critical state inconsistent.

This dissertation demonstrates system designs to address these two challenges. We first propose Nightcore, a low-latency FaaS runtime optimized for interactive microservices. Nightcore achieves low latency and high throughput by carefully considering various factors that have microsecond-scale overheads. We then propose Boki, a novel FaaS runtime for stateful serverless applications. Boki exports a shared log API for its functions, so that stateful serverless applications can use Boki shared logs to manage their state with durability, consistency, and fault tolerance.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Serverless computing is becoming popular for building cloud-native applications. Its function-as-a-service (FaaS) paradigm allows developers to execute their cloud functions on a cloud provider's infrastructure. The cloud provider is responsible for allocating, scaling, and monitoring execution environment and resources for users' functions. By providing elastic compute resources with fine-grained billing, serverless computing has shown success in massively parallel workloads, for example, video processing [80, 99], data analytics [110, 138], machine learning [92, 145], and distributed compilation [98].

While the current serverless infrastructure is attractive for high-throughput, stateless applications, it fails to support *low-latency* or *stateful* applications efficiently. Mainstream FaaS systems have relatively high runtime overhead. For example, invoking a warm nop function on AWS Lambda [17] has a median latency of 10.4ms (Table 3.1). Overheads of tens of milliseconds are negligible for batch processing workloads, where function bodies normally takes seconds to execute. But for latency-sensitive applications, such as interactive microservices, millisecond-scale overheads become unacceptable: measurements show that using FaaS for microservices can incur up to 12× overhead compared to non-serverless deployments [100, 109].

Moreover, in the current serverless environment, state management is difficult for stateful applications requiring high performance and strong data consistency. The current option for serverless state management is to rely on other cloud storage services such as object storage (e.g., Amazon S3 [24]) and cloud databases (e.g., DynamoDB [6]). When stateful applications are composed of multiple functions and serverless functions could fail at any point in their execution, existing state management options struggle to achieve strong consistency and fault tolerance while maintaining high performance and scalability [108, 142, 160]. There is an urgent need to re-think how to provide storage APIs for stateful serverless applications to manage critical state with strong consistency and fault tolerance.

This dissertation explores designs of serverless systems aiming to support emerging cloud applications that require microsecond-scale latencies and state storage with fault tolerance:

- To achieve low latencies for FaaS, we carefully consider engineering decisions for high performance without breaking the serverless paradigm. We present our low-latency FaaS runtime **Nightcore** in Chapter 3.

- To address data consistency and fault tolerance for serverless state management, we explore the potential of log-based storage in the serverless environment. We present our stateful serverless proposal **Boki** in Chapter 4.

1.1 Serverless computing for latency-sensitive microservices

As high-speed networks become prevalent in datacenters, cloud applications such as online services start to demand microsecond-scale processing latencies [134, 148]. When the scale of an online service grows (consider nowadays that a social network service needs to serve billions of users), the microservice architecture [100] becomes a practical engineering approach for building large-scale online services. In a microservice architecture, a large online application is built by connecting loosely coupled, single-purpose microservices, which communicate with each other via pre-defined APIs, mostly using remote procedure calls (RPC). While the interactive nature of online services implies an end-to-end service-level objectives (SLO) of a few tens of milliseconds, individual microservices face more strict latency SLOs – at the sub-millisecond-scale for leaf microservices [148, 165].

Previously, implementing individual microservices as RPC servers is the dominant approach for building microservice-based applications. With the popularity of serverless cloud computing, FaaS provides a new way of building microservice-based applications [12, 22, 56, 68], having the benefit of greatly reduced operational complexity. However, readily available FaaS systems have invocation latency overheads ranging from a few to tens of milliseconds [16, 70, 123] (also see Table 3.1), making them a poor choice for latency-sensitive interactive microservices, where RPC handlers only run for hundreds of microseconds to a few milliseconds [100, 122, 148, 149].

Nightcore [109] is a serverless function runtime designed and engineered to combine high performance with container-based isolation. Nightcore supports functions written in C/C++, Go, Node.js, and Python. To efficiently support latency-sensitive microservices, Nightcore has two performance goals which are not accomplished by existing FaaS systems: (1) invocation latency overheads are well within 100μs; (2) the invocation rate must scale to 100K/s with a low CPU usage. To meet its performance goals, Nightcore's design carefully considers various factors having microsecond-scale overheads, including scheduling of function requests, communication primitives, threading models for I/O, and concurrent function executions.

We use four realistic microservice workloads to evaluate Nightcore. Compared to OpenFaaS [47], a popular open-source FaaS runtime, Nightcore achieves $4.53\times$–$10.5\times$ higher throughput, while reducing tail latency by up to $10\times$.

1.2 Stateful serverless computing with shared logs

One key challenge in the current serverless paradigm is the mismatch between the stateless nature of serverless functions and the stateful applications built with them [104, 140, 146, 163]. Serverless applications are often composed of multiple functions, where application state is shared. However, managing shared state using current options, e.g., cloud databases or object stores, struggles to achieve strong consistency and fault tolerance while maintaining high performance and scalability [142, 160].

The shared log [83, 96, 154] is a popular approach for building storage systems that can simultaneously achieve scalability, strong consistency, and fault tolerance [82, 84, 85, 88]. A shared log offers a simple abstraction: a totally ordered log that can be accessed and appended concurrently. While simple, a shared log can efficiently support state machine replication [141], the well-understood approach for building fault-tolerant stateful services. The shared log API also frees distributed applications from the burden of managing the details of fault-tolerant consensus, because the consensus protocol is hidden behind the API [82]. Providing shared logs in FaaS can address the challenge of data consistency with fault tolerance for stateful serverless applications.

Boki [108] is a FaaS runtime that exports the shared log API to functions for storing shared state. Boki realizes the shared log API with a LogBook abstraction, where each function invocation is associated with a LogBook. For a Boki application, its functions share a LogBook, allowing them to manage application state with durability, consistency, and fault tolerance.

While Boki's design is inspired by previous shared log research [82, 83, 96, 154], serverless environment creates new challenges for Boki shared logs. In particular, serverless shared logs must be able to support diverse usage patterns efficiently. To satisfy different application needs, Boki's shared log abstraction (i.e., LogBooks) simultaneously provides high append throughput and low read latencies. Boki stores log records over variable numbers of shards to achieve high append throughput, while maintaining record cache on function nodes for low read latencies. In the serverless environment, resources are often shared by many applications. For best resource efficiency, Boki can support a high density of LogBooks with minimal metadata overheads, which is achieved by virtualizing application-facing LogBooks over internal physical shared logs.

Boki's design needs mechanisms for consistency and fault tolerance, and the *metalog* provides solutions to both. The metalog contains metadata that totally orders records for a Boki shared log. Boki performs log reads through log indices that are built in accordance with the metalog, where metalog positions are used to enforce read consistency. Boki handles machine failures by reconfiguration, and the metalog simplifies Boki's reconfiguration protocol. As the metalog exclusively controls the progress of a Boki shared log, sealing all the metalogs will allow a new configuration to be safely installed.

To make writing Boki applications easier, we build support libraries on top of the LogBook API aimed at three different serverless use cases: fault-tolerant workflows (BokiFlow), durable object storage (BokiStore), and serverless message queues (BokiQueue). Boki support libraries leverage techniques from Beldi [160], Tango [84], and vCorfu [154], while adapting them for the LogBook API. We use realistic cloud workloads to evaluate Boki support libraries, and our results suggest Boki can speed up important serverless applications by up to 4.2$\times$.

1.3 Organization of the dissertation

The rest of this dissertation is organized as follows. Chapter 2 provides background about serverless computing, where we also discuss what are challenges faced by today's serverless infrastructure. Chapter 3 introduces the design and implementation of Nightcore, our low-latency FaaS runtime optimized for latency-sensitive, interactive microservices. It also includes evaluation of Nightcore with realistic microservice workloads. Chapter 4 presents Boki, our proposal using shared logs to address challenges in stateful serverless computing. It details the design of Boki API, runtime, and Boki support libraries, It also evaluates Boki with selective cloud workloads. Chapter 5 discusses related work, and Chapter 6 concludes the dissertation.

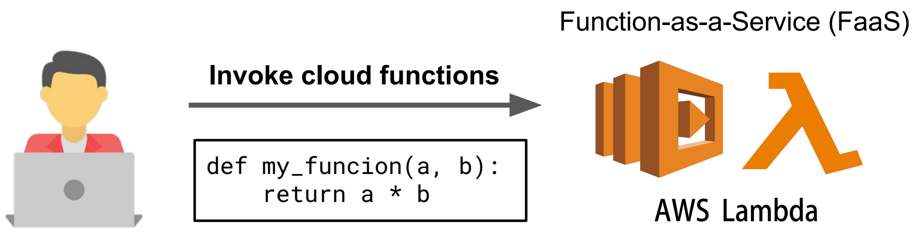
Chapter 2

The Current State of Serverless Computing

Cloud computing has become essential to computing's future [81]. Before cloud computing's popularity, traditional distributed applications were built on servers, which were the basic units of resource for compute and storage. In early days of cloud computing, cloud providers offered virtual machines (VMs) to ease the migration of server-based distributed applications to the cloud infrastructure.

Recently, serverless computing offers a new paradigm for building cloud-native applications [111]. In the new paradigm, distributed applications are built around fully managed cloud services, instead of servers or VMs. Those cloud services provide high-level abstractions for compute and storage, and are directly managed by the cloud provider. Relying on fully managed services can free cloud users from the burden of provisioning, maintaining, and monitoring the underlying machines running their applications.

Function-as-a-service (FaaS) plays a central role in the serverless paradigm. FaaS provides a simple yet powerful programming interface of stateless functions. Users can invoke their serverless functions which will be executed on the cloud infrastructure (Figure 2.1). AWS Lambda [17], Azure Functions [19], and Google Cloud Functions [23] are examples of commercial FaaS offerings.



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

Figure 2.1: In the FaaS paradigm, users can invoke cloud functions that are executed on a cloud provider's infrastructure (e.g., AWS Lambda [17]).

Current FaaS offerings provide two major benefits for cloud applications. The first benefit is *elasticity*. As the FaaS infrastructure is designed to support applications of diverse scale, developers need to do no extra work when their FaaS-based applications simply need more compute resource (i.e., more concurrently running functions). The second benefit is the *pay-as-you-go* billing model. Users of FaaS only pay for the time duration when their functions are actually running, and the billing granularity can be as low as one millisecond for individual functions [46]. These two benefits make FaaS particularly attractive for massively parallel but bursty workloads [80, 98, 99, 110, 138].

Unfortunately, FaaS by itself can only support *stateless* applications, but many important applications are inherently *stateful*. In the current serverless paradigm, stateful applications will use FaaS to execute their application logic, while relying on cloud storage services to store their application state. Cloud providers offer various storage services for different use cases, for example, object storage (e.g., Amazon S3 [24]), cloud database (e.g., DynamoDB [6]), and message queues (e.g., Amazon SQS [8]). Serverless functions of stateful applications will interact with these services to maintain their state (Figure 2.2).

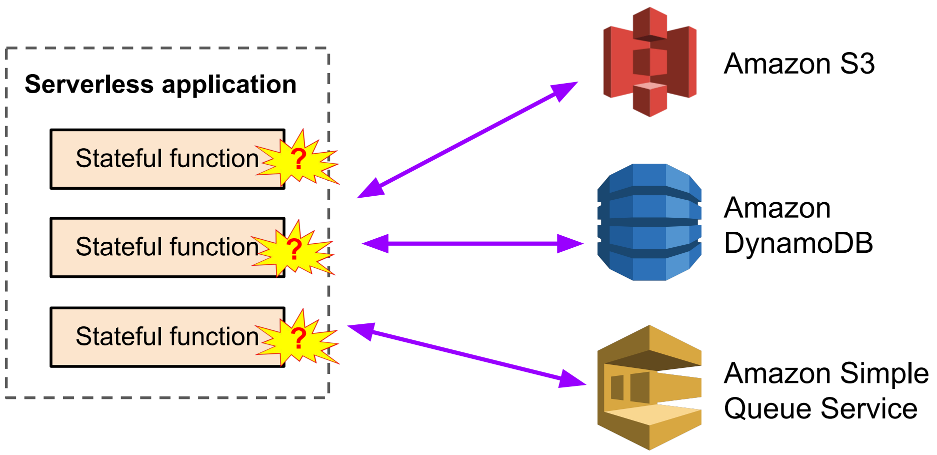


Figure 2.2: For a stateful serverless application, its functions will interactive with cloud storage services for maintaining application states. However, serverless functions could fail at any point of their execution, making it different to achieve state consistency with fault tolerance.

Serverless infrastructure in its current state is attractive for various cloud workloads, but there are emerging cloud applications for which serverless computing is not currently a viable option, either due to performance or functionality. We identify two challenges that prevent serverless computing to be adopted in broader applications. These two challenges faced by today's serverless infrastructure motivate system designs presented in this dissertation.

2.1 Challenge 1: High invocation latency in FaaS

To invoke a serverless function, the FaaS runtime has to find a machine to execute it. Serverless functions are normally executed within isolated environments, for example, containers or lightweight VMs [75, 143]. Individual serverless functions often have to bootstrap specific runtime environments, e.g., a Python environment. Because allocating and bootstrapping the execution environment can take a significant amount of CPU time, re-using a previous environment for later function invocations can improve overall performance. When an incoming function invocation is scheduled to use a previous environment, we call it a warm-start invocation. Otherwise, when the incoming invocation requires a new execution environment, we call it a cold-start invocation. Optimizing cold-start latency (i.e., the time for creating new execution environment) is one important research topic in serverless computing [75, 97].

However, even if we can avoid cold-start latencies by re-using execution environments, FaaS systems still incur runtime overheads for function invocations, e.g., caused by dispatching the function request to the executor machine. We use invocation latency to denote this intrinsic overhead from the underlying FaaS system. Invocation latencies cannot be avoided even for warm-start invocations.

Unfortunately, mainstream FaaS systems have invocation latency overheads ranging from a few to tens of milliseconds [16, 70, 123] (also see Table 3.1). Millisecond-scale overheads could be negligible for certain workloads, but certainly become unacceptable for latency-sensitive workloads. For example, it is popular to build online services with the microservice architecture, and the programming model of FaaS is naturally suitable for microservices [100]. But high invocation latencies from the FaaS runtime make serverless computing impractical for microservices: measurements show that using FaaS for microservices can incur up to $12\times$ overhead compared to non-serverless deployments [100, 109].

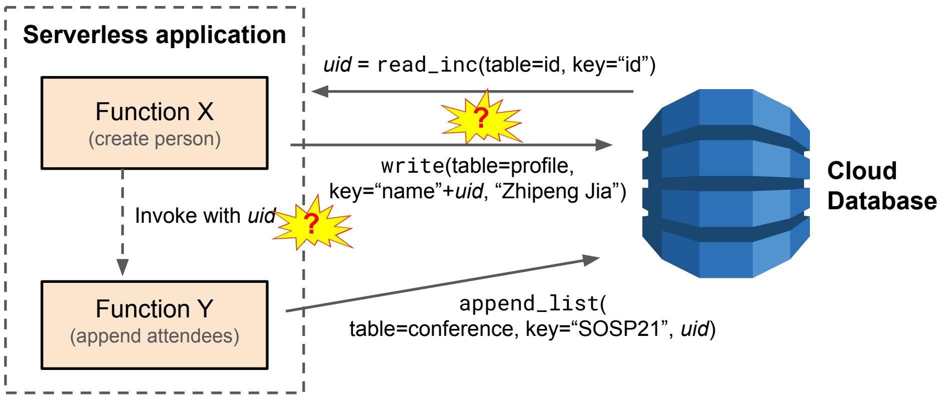


Figure 2.3: Example of a conference registration app using serverless workflows. This app uses cloud database to store registration states. Failures could happen within a function, or between the two functions. When failure happens, data stored in cloud database can be inconsistent (§ 2.2).

2.2 Challenge 2: State consistency with fault tolerance

To support stateful applications in the FaaS paradigm, cloud providers offer various storage services to suit different application needs, e.g., object storage, cloud databases, and message queues. These cloud storage services are fault-tolerant by themselves and some of them (such as cloud databases) support strong data consistency, however, serverless applications still struggle to achieve end-to-end consistency and fault tolerance for their critical states [142, 160].

This discrepancy arises because serverless functions can fail at any point during their execution, unbeknownst to external storage services which therefore cannot respond properly. Even worse, serverless applications often compose multiple functions for their functionality, e.g., serverless workflows, where function boundaries further complicate state consistency. We argue that fault tolerance and data consistency mechanisms used by cloud storage services cannot simply provide strong guarantees for serverless functions interacting with them without additional infrastructure.

We use a simple example to further explain the issue. Figure 2.3 demonstrates a conference registration app. This app uses serverless workflows for its application logic and a cloud database to store the registration state. The workflow consists of two functions X and Y. Function X is responsible for creating the attendee's profile in the database. It generates a *uid* for the new attendee and then invokes function Y with the *uid*. Function Y appends input *uid* to the attendance list, which finishes the registration process. If everything goes well, the process of this serverless workflow is fairly straightforward.

Unfortunately, with the current FaaS infrastructure, failures could happen at any point of function execution. For example, failures can happen during the execution of function X, between the two database update statements. Failures can also happen between function X and Y, in which case function Y is not successfully invoked. These potential failures can leave the registration state stored in the database inconsistent, for example, the profile for the new attendee is created but the *uid* is not added to the attendance list.

Olive [142] and Beldi [160] are previous studies aiming to address this challenge within the current cloud infrastructure. Their solutions implement write-ahead logs using the data model provided by cloud databases. The write-ahead log records all database statements, which provides protection for failures not within the database. However, building a logging layer over cloud databases not only has inevitable implementation complexity, but also incurs significant performance overheads for applications.

Based on observations from Olive and Beldi, we believe it is worthwhile to directly provide log storage for serverless functions. On the other hand, recent studies on distributed shared logs [82–84, 96, 154] demonstrate how to build a fault-tolerant, scalable log storage with high performance, and how applications can use shared logs for their fault tolerance and data consistency. These prior works inspire us to explore the potential of shared logs in stateful serverless computing (§ 4.1).

Chapter 3

Nightcore: Serverless Computing for Latency-Sensitive, Interactive Microservices

Serverless cloud computing enables a new way of building microservice-based applications [12, 22, 56, 68], having the benefit of greatly reduced operational complexity (§3.5). Serverless functions, or function as a service (FaaS), provide a simple programming model of *stateless* functions. These functions provide a natural substrate for implementing *stateless* RPC handlers in microservices, as an alternative to containerized RPC servers.

However, readily available FaaS systems, such as AWS Lambda [17], OpenFaaS [47], and Apache OpenWhisk [65], have invocation latency overheads ranging from a few to tens of milliseconds [16, 70, 123] (see Table 3.1). These latency overheads make them a poor choice for latency-sensitive interactive microservices, where RPC handlers only run for hundreds of microseconds to a few milliseconds [100, 122, 148, 149].

This chapter is based on the previous publication: "Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices", by Zhipeng Jia and Emmett Witchel in the Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021), pages 152–166, 2021 [109].

FaaS systems	**50th**	**99th**	**99.9th**
AWS Lambda	10.4 ms	25.8 ms	59.9 ms
OpenFaaS [47]	1.09 ms	3.66 ms	5.54 ms
Nightcore (external)	285 μs	536 μs	855 μs
Nightcore (internal)	39 μs	107 μs	154 μs

Table 3.1: Invocation latencies of a warm nop function.

This chapter presents Nightcore [109], a serverless function runtime designed and engineered to combine high performance with container-based isolation. To efficiently support interactive microservices, Nightcore has two performance goals which are not accomplished by existing FaaS systems: (1) invocation latency overheads are well within 100μs; (2) the invocation rate must scale to 100K/s with a low CPU usage.

We evaluate the Nightcore prototype on four interactive microservices, each with a custom workload. Three are from DeathStarBench [100] and one is from Google Cloud [38]. These workloads are originally implemented in RPC servers, and we port them to Nightcore, as well as OpenFaaS [47] for comparison. With containerized RPC servers as the baseline, Nightcore achieves $1.36\times$–$2.93\times$ higher throughput and up to 69% reduction in tail latency, while OpenFaaS only achieves 29%–38% of baseline throughput and increases tail latency by up to $3.4\times$ (§ 3.4). The evaluation shows that only by the carefully finding and eliminating microsecond-scale latencies can Nightcore use serverless functions to efficiently implement latency-sensitive microservices.

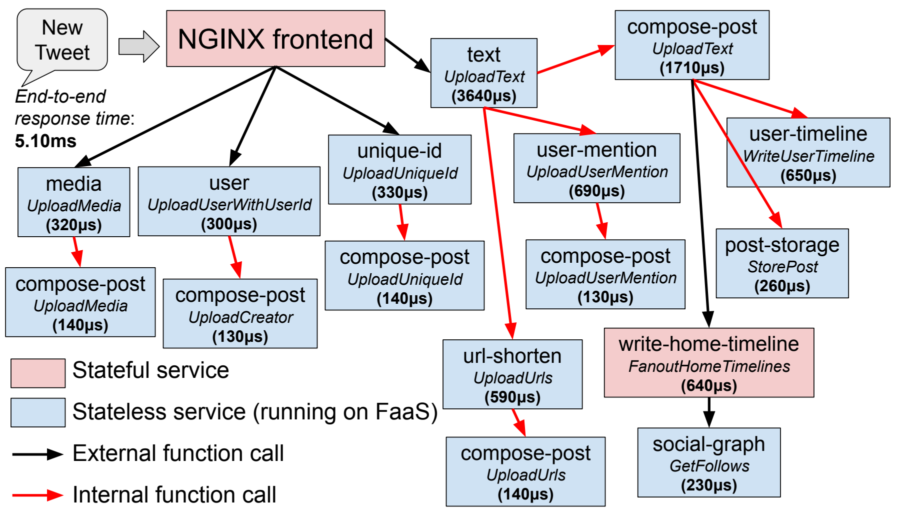

New Tweet
NGINX frontend
End-to-end response time: 5.10ms
text UploadText (3640µs)
compose-post UploadText (1710µs)
media UploadMedia (320µs)
user UploadUserWithUserId (300µs)
unique-id UploadUniqueId (330µs)
user-mention UploadUserMention (690µs)
user-timeline WriteUserTimeline (650µs)
compose-post UploadMedia (140µs)
compose-post UploadCreator (130µs)
compose-post UploadUniqueId (140µs)
compose-post UploadUserMention (130µs)
post-storage StorePost (260µs)
url-shorten UploadUrls (590µs)
write-home-timeline FanoutHomeTimelines (640µs)
compose-post UploadUrls (140µs)
social-graph GetFollows (230µs)
Stateful service
Stateless service (running on FaaS)
External function call
Internal function call

Figure 3.1: RPC graph of uploading new post in a microservice-based SocialNetwork application [100]. This graph omits stateful services for data caching and data storage.

3.1 Hunting for the "killer microseconds"

Nightcore has strict performance goals so that any microsecond-or-greater-scale performance overheads can be impactful, motivating a "hunt for the killer microseconds" [86] in the regime of FaaS systems.

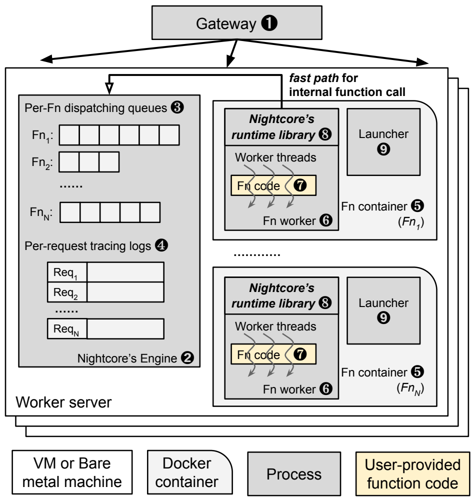
Existing FaaS systems like OpenFaaS [47] and Apache OpenWhisk [65] share a generic high-level design: all function requests are received by a *frontend* (mostly an API gateway), and then forwarded to independent *backends* where function code executes. The frontend and backends mostly execute on separate servers for fault tolerance, which requires invocation latencies that include at least one network round trip. Although datacenter networking performance is improving, round-trip times (RTTs) between two VMs in the same AWS region range from 101μs to 237μs [34]. Nightcore is motivated by noticing the prevalence of internal function calls made during function execution (see Figure 3.1). An *internal function call* is one that is generated by the execution of a microservice, not generated by a client (in which case it would be an external function call, received by the gateway). What we call internal function calls have been called "chained function calls" in previous work [145]. Nightcore schedules internal function calls on the same backend server that made the call, eliminating a trip to through the gateway and lowering latency (§ 3.2.2).

Nightcore's support for internal function calls makes most communication local, which means its inter-process communications (IPC) must be efficient. Popular, feature-rich RPC libraries like gRPC work for IPC (over Unix sockets), but gRPC's protocol adds overheads of $\sim$10μs [86], motivating Nightcore to design its own message channels for IPC (§ 3.2.1). Nightcore's message channels are built on top of OS pipes, and transmit fixed-size 1KB messages, because previous studies [122, 136] show that 1KB is sufficient for most microservice RPCs. Our measurements show Nightcore's message channels deliver messages in 3.4μs, while gRPC over Unix sockets takes 13μs for sending 1KB RPC payloads.

Previous work has shown microsecond-scale latencies in Linux's thread scheduler [86, 135, 148], leading dataplane OSes [87, 112, 127, 134, 135, 137] to build their own schedulers for lower latency. Nightcore relies on Linux's scheduler, because building an efficient, time-sharing scheduler for microsecond-scale tasks is an ongoing research topic [90, 112, 123, 134, 139]. To support an invocation rate of $\geq$100K/s, Nightcore's engine (§ 3.3.1) uses event-driven concurrency [32, 156], allowing it to handle many concurrent I/O events with a small number of OS threads. Our measurements show that 4 OS threads can handle an invocation rate of 100K/s. Furthermore, I/O threads in Nightcore's engine can wake function worker threads (where function code is executed) via message channels, which ensures the engine's dispatch suffers only a single wake-up delay from Linux's scheduler.

Existing FaaS systems do not provide concurrency management to applications. However, stage-based microservices create internal load variations even under a stable external request rate [106, 156]. Previous studies [48, 106, 155, 156] indicate overuse of concurrency for bursty loads can lead to worse overall performance. Nightcore, unlike existing FaaS systems, actively *manages* concurrency providing dynamically computed targets for concurrent function executions that adjust with input load (§ 3.2.3). Nightcore's managed concurrency flattens CPU utilization (see Figure 3.4) such that overall performance and efficiency are improved, as well as being robust under varying request rates (§ 3.4.2).

By leveraging aforementioned optimization techniques, Nightcore manages to achieve its performance goals. Shown in Table 3.1, the median latency overhead of internal function calls in Nightcore is 39μs, while the 99-percentile tail is 107μs. The results demonstrate the runtime overhead of Nightcore is orders of magnitude better than previous FaaS systems, which is the key for Nightcore's success in latency-sensitive microservice-based applications.



❶ Gateway	• Accept external function requests • Load balance requests to worker servers
❷ Engine	• The main Nightcore process on each worker server, which communicates with Gateway ❶, launchers ❽, and worker threads inside Fn workers ❻ • Maintain per-function dispatching queues ❸ and per-request tracing logs ❹
❸ Dispatching queues	• Function requests queued here • Dispatch function requests to worker threads in Fn worker ❻
❹ Tracing logs	• Track life-cycle of all function invocations, by recording receive, dispatch, and completion timestamps
❺ Fn container	• Execution environment for individual functions • Consists of Fn worker ❻ and Launcher ❽ processes
❻ Fn worker process	• Multiple worker threads execute user-provided function code ❼, and call a runtime library ❽ for fast, internal function call • Implementation tailored to each supported programming language
❼ User-provided Fn code	• Stateless function code written in supported programming language (C/C++, Go, Python, or Node.js) • Executed on worker threads within Fn worker process ❻
❽ Runtime library	• Fast path for internal function call: talk directly with Engine to enqueue the function call ❸, entirely bypassing Gateway ❶
❾ Launcher	• Launch new Fn worker ❻ or worker threads

Figure 3.2: Architecture of Nightcore (§ 3.2.1).

3.2 Nightcore design

Nightcore is designed to run serverless functions with sub-millisecond-scale execution times, and to efficiently process internal function calls, which are generated during the execution of a serverless function (not by an external client). Nightcore exposes a serverless function interface that is similar to AWS Lambda: users provide stateless function handlers written in supported programming languages. The only addition to this simple interface is that Nightcore's runtime library provides APIs for fast internal function invocations.

3.2.1 System architecture

Figure 3.2 depicts Nightcore's design which mirrors the design of other FaaS systems starting with the separation of *frontend* and *backend*. Nightcore's *frontend* is an API gateway for serving external function requests and other management requests (e.g., to register new functions), while the *backend* consists of a number of independent **worker servers**. This separation eases availability and scalability of Nightcore, by making the *frontend* API gateway fault tolerant and horizontally scaling *backend* worker servers. Each worker server runs a Nightcore engine process and function containers, where each function container has one registered serverless function, and each function has only one container on each worker server. Nightcore's engine directly manages function containers and communicates with worker threads within containers.

Internal function calls. Nightcore optimizes internal function calls locally on the same worker server, without going through the API gateway. Figure 3.2 depicts this fast path in Nightcore's runtime library which executes inside a function container. By optimizing the locality of dependent function calls, Nightcore brings performance close to a monolithic design. At the same time, different microservices remain logically independent and they execute on different worker servers, ensuring there is no single point of failure. Moreover, Nightcore preserves the engineering and deployment benefits of microservices such as diverse programming languages and software stacks.

Nightcore's performance optimization for internal function calls assumes that an individual worker server is capable of running most function containers from a single microservice-based application [1]. We believe this is justified because we

[1]Nightcore also needs to know which set of functions form a single application. In practice, this knowledge comes directly from the developer, e.g., Azure Functions allow developers to organize related functions as a single *function app* [43].

measure small working sets for stateless microservices. For example, when running SocialNetwork [100] at its saturation throughput, the 11 stateless microservice containers consume only 432 MB of memory, while the host VM is provisioned with 16 GB. As current datacenter servers have growing numbers of CPUs and increasing memory sizes (e.g., AWS EC2 VMs have up to 96 vCPUs and 192 GB of memory), a single server is able to support the execution of thousands of containers [145, 164]. When it is not possible to schedule function containers on the same worker server, Nightcore falls back to scheduling internal function calls on different worker servers through the gateway.

Gateway. Nightcore's gateway (Figure 3.2①) performs load balancing across worker servers for incoming function requests and forwards requests to Nightcore's engine on worker servers. The gateway also uses external storage (e.g., Amazon's S3) for saving function metadata and it periodically monitors resource utilizations on all worker servers, to know when it should increase capacity by launching new servers.

Engine. The engine process (Figure 3.2②) is the most critical component of Nightcore for achieving microsecond-scale invocation latencies, because it invokes functions on each worker server. Nightcore' engine responds to function requests from both the gateway and from the runtime library within function containers. It creates low-latency message channels to communicate with function workers and launchers inside function containers (§ 3.3.1). Nightcore's engine is event driven

(Figure 3.5) allowing it to manage hundreds of message channels using a small number of OS threads. Nightcore's engine maintains two important data structures: (1) Per-function dispatching queues for dispatching function requests to function worker threads (Figure 3.2③); (2) Per-request tracing logs for tracking the life cycle of all inflight function invocations, used for computing the proper concurrency level for function execution (Figure 3.2④).

Function containers. Function containers (Figure 3.2⑤) provide isolated environments for executing user-provided function code. Inside the function container, there is a launcher process, and one or more worker processes depending on the programming language implementation (see § 3.3.2 for details). Worker threads within worker processes receive function requests from Nightcore's engine, and execute user-provided function code. Worker processes also contain a Nightcore runtime library, exposing APIs for user-provided function code. The runtime library includes APIs for fast internal function calls without going through the gateway. Nightcore's internal function calls directly contact the dispatcher to enqueue the calls that are executed on the same worker server without having to involve the gateway.

Nightcore has different implementations of worker processes for each supported programming language. The notion of "worker threads" is particularly malleable because different programming languages have different threading models. Futhermore, Nightcore's engine does not distinguish worker threads from worker processes, as it maintains communication channels with each individual worker thread. For clarity of exposition we assume the simplest case in this Section (which holds for the C/C++ implementation), where "worker threads" are OS threads (details for other languages in § 3.3.2).

Isolation in Nightcore. Nightcore provides container-level isolation between different functions, but does not guarantee isolation between different invocations of the same function. We believe this is a reasonable trade-off for microservices, as creating a clean isolated execution environment within tens of microseconds is too challenging for current systems. When using RPC servers to implement microservices, different RPC calls of the same service can be concurrently processed within the same process, so Nightcore's isolation guarantee is as strong as containerized RPC servers.

Previous FaaS systems all have different trade-offs between isolation and performance. OpenFaaS [67] allows concurrent invocations within the same function worker process, which is the same as Nightcore. AWS Lambda [15] does not allow concurrent invocations in the same container/MicroVM but allows execution environments to be re-used by subsequent invocations. SAND [77] has two levels of isolation–different applications are isolated by containers but concurrent invocations within the same application are only isolated by processes. Faasm [145] leverages the software-based fault isolation provided by WebAssembly, allowing a new execution environment to be created within hundreds of microseconds, but it relies on language-level isolation which is weaker than container-based isolation.

Message channels. Nightcore's message channels are designed for low-latency message passing between its engine and other components, which carry fixed-size 1KB messages. The first 64 bytes of a message is the header which contains the message type and other metadata, while the remaining 960 bytes are message payload. There are three types of messages relevant to function invocations:

(1) *Dispatch*, used by engine for dispatching function requests to worker threads (④ in Figure 3.3).
(2) *Completion*, used by function worker threads for sending outputs back to the engine (⑥ in Figure 3.3), as well as by the engine for sending outputs of internal function calls (⑦ in Figure 3.3).
(3) *Invoke*, used by Nightcore's runtime library for initiating internal function calls (② in Figure 3.3).

When payload buffers are not large enough for function inputs or outputs, Nightcore creates extra shared memory buffers for exchanging data. In our experiments, these overflow buffers are needed for less than 1% of the messages for most workloads, though HipsterShop needs them for 9.7% of messages. When overflow buffers are required, they fit within 5KB 99.9% of the time. Previous work [122] has shown that 1KB is sufficient for more than 97% of microservice RPCs.

3.2.2 Processing function requests

Figure 3.3 shows an example with both an external and internal function call. Suppose code of Fn_x includes an invocation of Fn_y. In this case, Fn_y is invoked via Nightcore's runtime API (①). Then, Nightcore's runtime library generates a

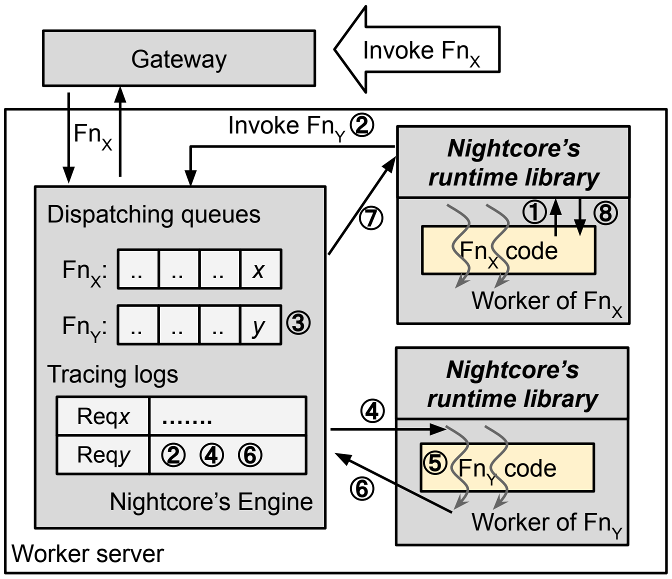


Figure 3.3: Diagram of an internal function call (§ 3.2.2).

unique ID (denoted by req_y) for the new invocation, and sends an internal function call request to Nightcore's engine (②). On receiving the request, the engine writes req_y's receive timestamp (also ②). Next, the engine places req_y in the dispatching queue of Fn_y ③. Once there is an idle worker thread for Fn_y and the concurrency level of Fn_y allows, the engine will dispatch req_y to it, and records req_y's dispatch timestamp in its tracing log (④). The selected worker thread executes Fn_y's code (⑤) and sends the output back to the engine (⑥). On receiving the output, the engine records request req_y's completion timestamp (also ⑥), and directs the function output back to Fn_x's worker (⑦). Finally, execution flow returns back to user-provided Fn_x ⑧.

3.2.3 Managing concurrency for function executions (τ_k)

Nightcore maintains a pool of worker threads in function containers for concurrently executing functions, but deciding the size of thread pools can be a hard problem. One obvious approach is to always create new worker threads when needed, thereby maximizing the concurrency for function executions. However, this approach is problematic for microservice-based applications, where one function often calls many others. Maximizing the concurrency of function invocations with high fanout can have a domino effect that overloads a server. The problem is compounded when function execution time is short. In such cases, overload happens too quickly for a runtime system to notice it and respond appropriately.

To address the problem, Nightcore adaptively manages the number of concurrent function executions, to achieve the highest useful concurrency level while preventing instantaneous server overload. Following *Little's law*, the ideal concurrency can be estimated as the product of the average request rate and the average processing time. For a registered function Fn_k, Nightcore's engine maintains exponential moving averages of its invocation rate (denoted by λ_k) and function execution time (denoted by t_k). Both are computed from request tracing logs. Nightcore uses their product $\lambda_k \cdot t_k$ as the concurrency hint (denoted by τ_k) for function Fn_k.

When receiving an invocation request of Fn_k, the engine will only dispatch the request if there are fewer than τ_k concurrent executions of Fn_k. Otherwise, the request will be queued, waiting for other function executions to finish. In other words, the engine ensures the maximum concurrency of Fn_k is τ_k at any moment. Note that Nightcore's approach is adaptive because τ_k is computed from two exponential moving averages (λ_k and t_k), that change over time as new function requests are received and executed. To realize the desired concurrency level, Nightcore must also maintain a worker thread pool with at least τ_k threads. However, the dynamic nature of τ_k makes it change rapidly (see Figure 3.7), and frequent creation and termination of threads is not performant. To modulate the dynamic values of τ_k, Nightcore allows more than τ_k threads to exist in the pool, but only uses τ_k of them. It terminates extra threads when there are more than $2\tau_k$ threads.

Nightcore's managed concurrency is fully automatic, without any knowledge or hints from users. The concurrency hint (τ_k) changes frequently at the scale of microseconds, to adapt to load variation from microsecond-scale microservices (§ 3.4.2) . Figure 3.4 demonstrates the importance of managing concurrency levels instead of maximizing them. Even when running at a fixed input rate, CPU utilization varies quite a bit for both OpenFaaS and Nightcore when the runtime maximizes the concurrency. On the other hand, managing concurrency with hints has a dramatic "flatten-the-curve" benefit for CPU utilization.

3.3 Implementation

Nightcore's API gateway and engine consists of 8,874 lines of C++. Function workers are supported in C/C++, Go, Node.js, and Python, requiring 1,588 lines of C++, 893 lines of Go, 57 lines of JavaScript, and 137 lines of Python.

Nightcore's engine (its most performance-critical component) is imple-

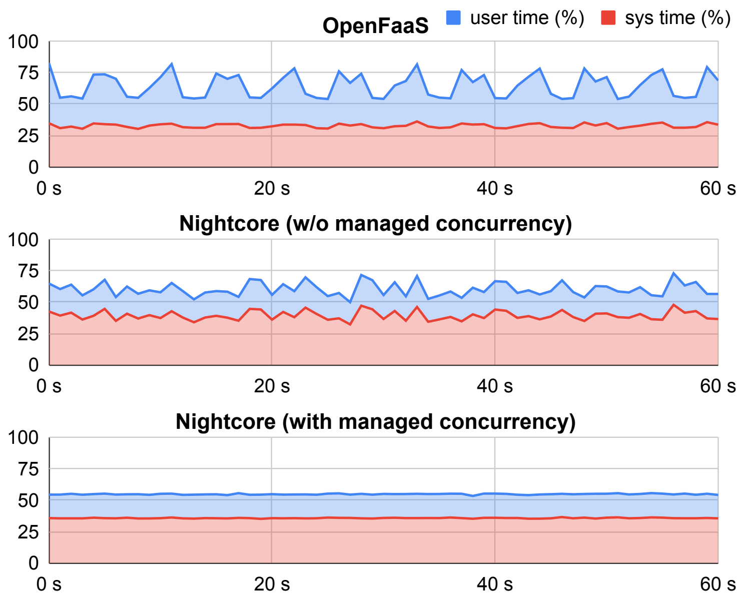


Figure 3.4: CPU utilization timelines of OpenFaaS, and Nightcore (without and with managed concurrency), running SocialNetwork microservices [100] at a fixed rate of 500 QPS for OpenFaaS and 1200 QPS for Nightcore.

mented in C++. Garbage collection can have a significant impact for latency-sensitive services [158] and short-lived routines [36, 37]. Both OpenFaaS [47] and Apache OpenWhisk [65] are implemented with garbage-collected languages (Go and Scala, respectively), but Nightcore eschews garbage collection in keeping with its theme of addressing microsecond-scale latencies.

3.3.1 Nightcore's engine

Figure 3.5 shows the event-driven design of Nightcore's engine as it responds to I/O events from the gateway and message channels. Each I/O thread maintains a fixed number of persistent TCP connections to the gateway for re-

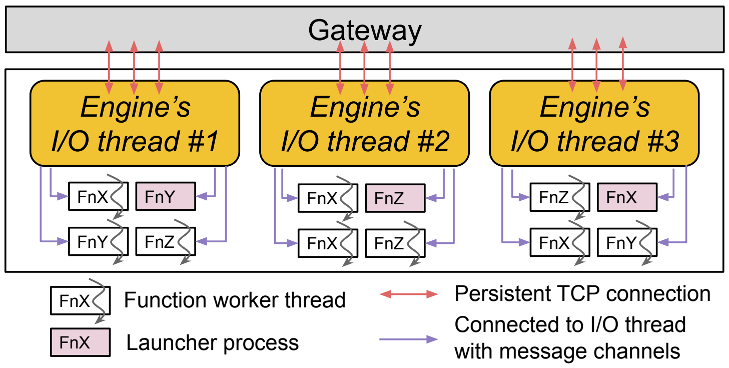


Figure 3.5: Event-driven I/O threads in Nightcore's engine (§ 3.3.1).

ceiving function requests and sending back responses, while message channels are assigned to I/O threads with a round-robin policy. Individual I/O threads can only read from and write to their own TCP connections and message channels. Shared data structures including dispatching queues and tracing logs are protected by mutexes, as they can be accessed by different I/O threads.

Event-driven I/O threads. Nightcore's engine adopts `libuv` [41], which is built on top of the `epoll` system call, to implement its event-driven design. `libuv` provides APIs for watching events on file descriptors, and registering handlers for those events. Each I/O thread of the engine runs a `libuv` event loop, which polls for file descriptor events and executes registered handlers.

Message channels. Nightcore's messages channels are implemented with two Linux pipes in opposite directions to form a full-duplex connection. Meanwhile, shared memory buffers are used when inline payload buffers are not large enough for function inputs or outputs (§ 3.2). Although shared memory allows fast IPC at memory speed, it lacks an efficient mechanism to notify the consumer thread when data is available. Nightcore's use of pipes and shared memory gets the best of both worlds. It allows the consumer to be eventually notified through a blocking read on the pipe, and at the same time, it provides the low latency and high throughput of shared memory when transferring large message payloads.

As the engine and function workers are isolated in different containers, Nightcore mounts a shared `tmpfs` directory between their containers, to aid the setup of pipes and shared memory buffers. Nightcore creates named pipes in the shared `tmpfs`, allowing function workers to connect. Shared memory buffers are implemented by creating files in the shared `tmpfs`, which are `mmap`ed with the `MAP_SHARED` flag by both the engine and function workers. Docker by itself supports sharing IPC namespaces between containers [40], but the setup is difficult for Docker's cluster mode. Nightcore's approach is functionally identical to IPC namespaces, as Linux's System V shared memory is internally implemented by `tmpfs` [58].

Communications between function worker threads. Individual worker threads within function containers connect to Nightcore's engine with a message channel for receiving new function requests and sending responses (④ and ⑥ in Figure 3.3). A worker thread can be either *busy* (executing function code) or *idle*. During the execution of function code, the worker thread's message channel is also used by Nightcore's runtime library for internal function calls (② and ⑦ in Figure 3.3). When a worker thread finishes executing function code, it sends a response message with the function output to the engine and enters the idle state. An idle worker thread is put to sleep by the operating system, but the engine can wake it by writing a function request message to its message channel. The engine tracks the busy/idle state of each worker so there is no queuing at worker threads, the engine only dispatches requests to idle workers.

Mailbox. The design of Nightcore's engine only allows individual I/O threads to write data to message channels assigned to it (shown as violet arrows in Figure 3.5). In certain cases, however, an I/O thread needs to communicate with a thread that does not share a message channel. Nightcore routes these requests using per-thread mailboxes. When an I/O thread drops a message in the mailbox of another thread, `uv_async_send` (using `eventfd` [33] internally) is called to notify the event loop of the owner thread.

Computing concurrency hints (τ_k). To properly regulate the amount of concurrent function executions, Nightcore's engine maintains two exponential moving averages λ_k (invocation rate) and t_k (processing time) for each function Fn_k (§ 3.2.3). Samples of invocation rates are computed as $1/$(interval between consecutive Fn_k invocations), while processing times are computed as intervals between dispatch and completion timestamps, excluding queueing delays (the interval between receive and dispatch timestamps) from sub-invocations. Nightcore uses a coefficient $\alpha = 10^{-3}$ for computing exponential moving averages.

3.3.2 Function workers

Nightcore executes user-provided function code in its function worker processes (§ 3.2.1). As different programming languages have different abstractions for threading and I/O, Nightcore has different function worker implementations for them.

Nightcore's implementation of function workers also includes a runtime library for fast internal function calls. Nightcore's runtime library exposes a simple API `output := nc_fn_call(fn_name, input)` to user-provided function code for internal function calls. Furthermore, Nightcore's runtime library provides Apache Thrift [11] and gRPC [39] wrappers for its function call API, easing porting of existing Thrift-based and gRPC-based microservices to Nightcore.

C/C++. Nightcore's C/C++ function workers create OS threads for executing user's code, loaded as dynamically linked libraries. These OS threads map to "worker threads" in Nightcore's design (§ 3.2.1 and Figure 3.2). To simplify the implementation, each C/C++ function worker process only runs one worker thread, and the launcher will fork more worker processes when the engine asks for more worker threads.

Go. In Go function workers, "worker threads" map to goroutines, the user-level threads provided by Go's runtime, and the launcher only forks one Go worker process. Users' code are compiled together with Nightcore's Go worker implementation, as Go's runtime does not support dynamic loading of arbitrary Go code [2]. Go's runtime allows dynamically setting the maximum number of OS threads for running goroutines (via `runtime.GOMAXPROCS`), and Nightcore's implementation sets it to $\lceil$worker goroutines$/8\rceil$.

Node.js and Python. Node.js follows an event-driven design where all I/O is asynchronous without depending on multi-threading, while Python is the same when using the asyncio [13] library for I/O. In both cases, Nightcore implements its message channel protocol within their event loops. As there are no parallel threads [3] inside Node.js and Python function workers, launching a new "worker thread" simply means creating a message channel, while the engine's notion of "worker threads" becomes event-based concurrency [32]. Also, `nc_fn_call` is an asynchronous API in Node.js and Python workers, rather than being synchronous in C/C++ and Go workers. For Node.js and Python functions, the launcher only forks one worker process.

3.4 Evaluation

We conduct all of our experiments on Amazon EC2 C5 instances in the `us-east-2` region, running Ubuntu 20.04 with Linux kernel 5.4.41. We enable hyperthreading, but disable transparent huge pages.

[2]Go partially supports dynamic code loading via a plugin [49], but it requires the plugin and the loader be compiled with a same version of the Go toolchain, and all their dependency libraries have exactly the same versions.

[3]Node.js supports worker threads [71] for running CPU-intensive tasks, but they have worse performance for I/O-intensive tasks.

	Ported services	**RPC framework**	**Languages**
SocialNetwork	11	Thrift [11]	C++
MovieReviewing	12	Thrift	C++
HotelReservation	11	gRPC [39]	Go
HipsterShop	13	gRPC	Go, Node.js, Python

Table 3.2: Microservice workloads in evaluation (from [38, 100]).

3.4.1 Methodology

Microservice workloads. Nightcore is designed to optimize microservice workloads, so we evaluate it on the four most realistic, publicly available, interactive microservice code bases: SocialNetwork, MovieReviewing, HotelReservation, and HipsterShop. The first three are from DeathStarBench [100], while HipsterShop is a microservice demo from Google Cloud Platform [38]. The workloads are summarized in Table 3.2.

For the SocialNetwork workload, we tested two load patterns: (1) a pure load of ComposePost requests (shown in Figure 3.1) (denoted as "write"); (2) a mixed load (denoted as "mixed"), that is a combination of 30% CompostPost, 40% ReadUserTimeline, 25% ReadHomeTimeline, and 5% FollowUser requests.

HipsterShop itself does not implement data storage, and we modify it to use MongoDB for saving orders and Redis for shopping carts. We also add Redis instances for caching product and ad lists. HipsterShop includes two services written in Java and C#, which are not languages supported by Nightcore. Thus we

SocialNetwork		Movie	Hotel	Hipster
write	mixed	Reviewing	Reservation	Shop
66.7%	62.3%	69.2%	79.2%	85.1%

Table 3.3: Percentage of internal function calls (§ 3.4.1).

re-implement their functionality in Go and Node.js.

For each benchmark workload, we port their stateless mid-tier services to Nightcore, as well as OpenFaaS [47] for comparison. For other stateful services (e.g., database, Redis, NGINX, etc.), we run them on dedicated VMs with sufficiently large resources to ensure they are not bottlenecks. For Nightcore and OpenFaaS, their API gateways also run on a separate VM. This configuration favors OpenFaaS because its gateway is used for both external and internal function calls and is therefore more heavily loaded than Nightcore's gateway.

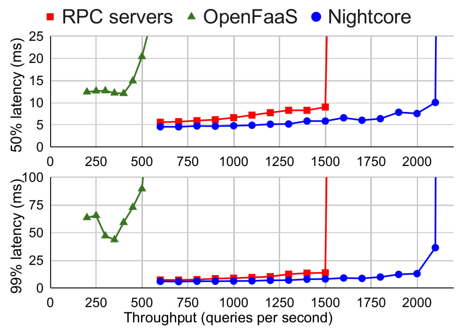
We use `wrk2` [35] as the load generator for all workloads. In our experiments, the target input load (queries per second (QPS)) is run for 180 seconds, where the first 30 seconds are used for warming up the system, and 50th and 99th percentile latencies of the next 150 seconds are reported. Following this methodology, variances of measured latencies are well within 10% before reaching the saturation throughput.

Internal function calls. One major design decision in Nightcore is to optimize internal function calls (§ 3.2), so we quantify the percentage of internal function calls in our workloads in Table 3.3. The results show that internal function calls dominate external calls, sometimes by more than a factor of $5\times$.

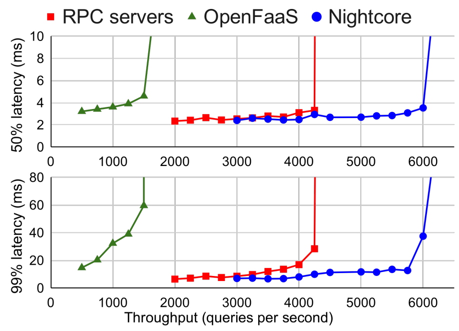
Cold-Start latencies. There are two components of cold-start latencies in a FaaS system. The first arises from provisioning a function container. Our prototype of Nightcore relies on unmodified Docker, thus does not include optimizations. However, state-of-the-art techniques such as Catalyzer [97] achieve startup latencies of 1–14ms. These techniques can be applied to Nightcore as they become mainstream. The second component of cold-start latency is provisioning the FaaS runtime inside the container. Our measurement shows that Nightcore's function worker process takes only 0.8ms to be ready for executing user-provided function code.

Systems for comparison. We compare Nightcore with two other systems: (1) RPC servers running in Docker containers which are originally used for implementing stateless microservices in the evaluation workloads; (2) OpenFaaS [47], a popular open source FaaS system, where we use the OpenFaaS watchdog [67] in HTTP mode to implement function handlers, and Docker for running function containers.

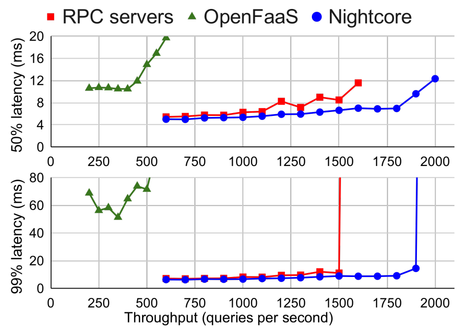
We also tested the SocialNetwork application on AWS Lambda. Even when running with a light input load and with provisioned concurrency, Lambda cannot meet our latency targets. Executing the "mixed" load pattern shows median and 99% latencies are 26.94ms and 160.77ms, while they are 2.34ms and 6.48ms for containerized RPC servers. These results are not surprising given the measurements in Table 3.1.



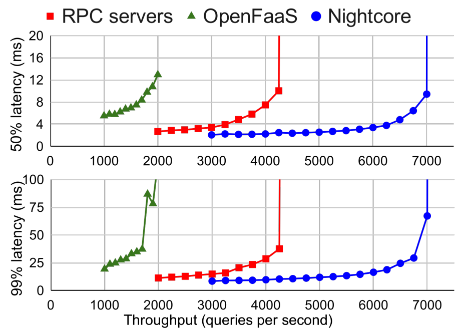
(a) SocialNetwork (write)



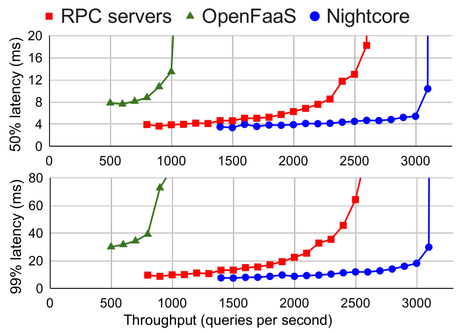
(b) SocialNetwork (mixed)



(c) MovieReviewing



(d) HotelReservation



(e) HipsterShop

Figure 3.6: Comparison of Nightcore with RPC servers and OpenFaaS using one VM.

3.4.2 Benchmarks

Single worker server. We start with evaluating Nightcore with one worker server. All systems use one `c5.2xlarge` EC2 VM, which has 8 vCPUs and 16GiB of memory. For Nightcore and OpenFaaS, this VM is its single worker server, that executes all serverless functions. On the worker VM, Nightcore's engine uses two I/O threads. For RPC servers, this VM runs all ported stateless microservices, such that all inter-service RPCs are local.

Figure 3.6 demonstrates experimental results on all workloads. For all workloads, results show the trend that OpenFaaS' performance is dominated by containerized RPC servers, while Nightcore is superior to those RPC servers. OpenFaaS' performance trails the RPC servers because all inter-service RPCs flow through OpenFaaS's gateway and watchdogs, which adds significant latency and CPU processing overheads. On the other hand, Nightcore's performance is much better than OpenFaaS, because Nightcore optimizes the gateway out of most inter-service RPCs, and its event-driven engine handles internal function calls with microsecond-scale overheads.

Compared to RPC servers, Nightcore achieves $1.27\times$ to $1.59\times$ higher throughput and up to 34% reduction in tail latencies, showing that Nightcore has a lower overhead for inter-service communications than RPC servers, which we will discuss more in § 3.4.3.

We also tested Nightcore under variable load to demonstrate how its ability to manage concurrency (§ 3.2.3) adapts to changing loads. Figure 3.7 shows the

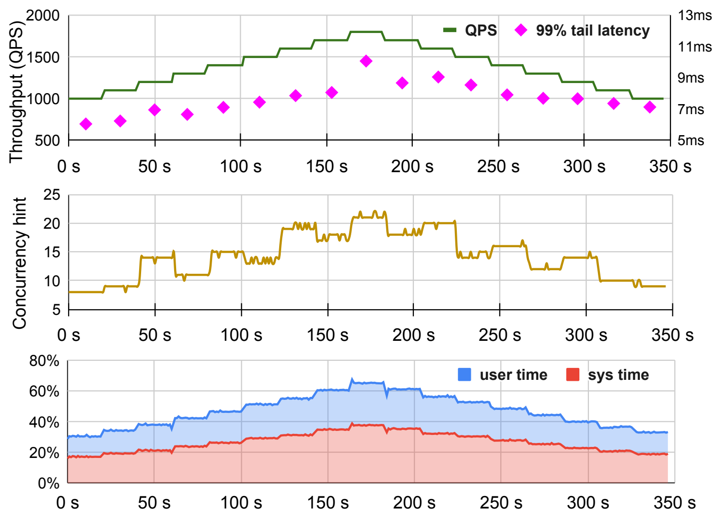


Figure 3.7: Nightcore running SocialNetwork (write) with load variations. The upper chart shows tail latencies under changing QPS, the middle chart shows how the concurrency hint (τ_k) of the post microservice changes with time, and the lower chart is a timeline of CPU utilization.

results with a corresponding timeline of CPU utilization, indicating that Nightcore can promptly change its concurrency level under increasing loads. When the input load reaches its maximum (of 1800 QPS), the tail latency also reaches its maximum (of 10.07 ms).

	Base QPS	Median latency (ms)				99% tail latency (ms)			
		1 server	2 server	4 server	8 server	1 server	2 server	4 server	8 server
SocialNetwork (mixed)	2000	3.40	2.64	2.39	2.64	10.93	8.36	7.18	8.07
	2300	3.37	2.65	2.43	2.61	13.95	10.34	8.20	10.63
MovieReviewing	800	7.24	7.93	7.35	8.10	9.26	11.42	10.97	16.31
	850	7.24	7.54	7.57	8.57	9.31	11.18	12.24	25.01
HotelReservation	3000	3.48	3.29	3.08	4.32	18.27	15.98	14.98	18.09
	3300	5.56	4.43	5.50	4.43	31.92	22.66	22.54	20.83
HipsterShop	1400	6.05	5.70	6.23	5.68	19.68	17.42	19.10	15.02
	1500	7.95	7.51	8.32	7.06	25.39	23.74	23.81	20.53

Table 3.4: Evaluation of Nightcore's scalability, where n worker servers run n times the base QPS input. For each workload, the base QPS is selected to be close to the saturation throughput when using one server.

	SocialNetwork (mixed)			MovieReviewing			HotelReservation			HipsterShop		
	QPS	median	tail	QPS	median	tail	QPS	median	tail	QPS	median	tail
RPC servers	1.00x	3.21	23.98	1.00x	14.45	25.57	1.00x	5.54	19.73	1.00x	10.68	48.13
	1.17x	110.01	>1000	1.20x	30.80	>1000	1.22x	10.43	43.46	1.17x	15.61	80.89
OpenFaaS	0.29x	4.57	81.60	0.30x	10.06	113.47	0.28x	5.80	18.96	0.29x	9.29	32.13
	0.33x	6.72	368.38	0.40x	13.32	>1000	0.33x	16.21	103.81	0.38x	24.93	86.59
Nightcore	1.33x	2.64	8.07	1.28x	8.10	16.31	2.67x	4.32	18.09	1.87x	5.68	15.02
	1.53x	2.61	10.63	1.36x	8.57	25.01	2.93x	4.43	20.83	2.00x	7.06	20.53

Table 3.5: Comparison of Nightcore with other systems using 8 VMs. Median and 99% tail latencies are shown in milliseconds. For each workload, the saturation throughput of the RPC servers is the baseline QPS (1.00x in the table) for comparison.

Multiple worker servers. We also evaluate Nightcore in a distributed setting, where multiple VMs are used as Nightcore's worker servers. We use `c5.xlarge` EC2 instances for worker VMs, which have 4 vCPUs and 8GiB of memory.

Table 3.4 evaluates Nightcore's scalability, where up to 8 VMs are used for worker servers and the input load is scaled with the number of VMs. The similar (or decreasing) median and tail latencies show that Nightcore's scalability is nearly linear–e.g., increasing the input load $8\times$ and providing 8 servers does not change median and tail latencies significantly. The only exception is MovieReviewing, where the tail latency of running on 8 server is $2.7\times$ worse than 1 server. However, we observe that scaling this workload with RPC servers also suffers increased tail latencies.

Next we compare RPC servers and OpenFaaS to Nightcore with 8 worker VMs. For RPC servers, 8 VMs run stateless services, where each VM runs one replica of each service and load balancing is transparently supported by RPC client libraries. For OpenFaaS and Nightcore, 8 VMs run their function handlers. Table 3.5 summarizes the experimental results, demonstrating that Nightcore achieves $1.36\times$ to $2.93\times$ higher throughput and up to 69% reduction in tail latency than an RPC server (while OpenFaaS consistently underperforms an RPC server). The advantage of Nightcore over RPC servers is more significant in the distributed setting, because there are inter-host RPCs in the case of replicated RPC servers, while there is no network traffic among Nightcore's worker VMs.

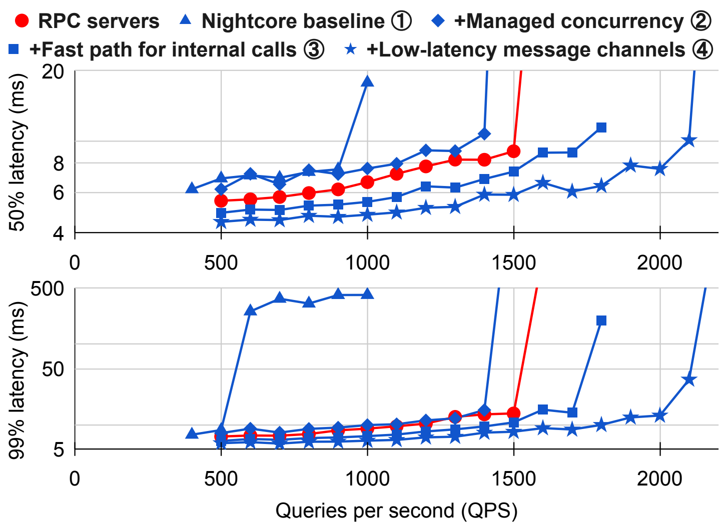


Figure 3.8: Comparison of Nightcore with RPC servers, running SocialNetwork (write) using one VM. Designs of Nightcore are progressively added to show their effectiveness (§ 3.4.3).

3.4.3 Analysis

Evaluating Nightcore's design. We quantify the value of Nightcore's design decisions in Figure 3.8. The figure shows the performance effect of adding managed concurrency for function executions (§ 3.2.3), a fast path for internal function call (§ 3.2.2), and low-latency message channels as IPC primitives (§ 3.2.1). The Nightcore baseline ① maximizes concurrent function executions (i.e., concurrency management is disabled), all internal function calls go through the gateway, and Nightcore's message channels are replaced with TCP sockets. This baseline Nightcore design can achieve only one third the throughput of RPC servers while meeting tail latency targets (①). When we add managed concurrency (②), Nightcore's performance comes close to RPC servers, as tail latencies are significantly improved.

Optimizing the gateway out of the processing path for internal function calls (③) brings Nightcore's performance above the RPC servers. Finally, Nightcore's low-latency message channels boost performance further (④), resulting in $1.33\times$ higher throughput than RPC servers.

Communication overheads. Microservice-based applications are known to have a high communication-to-computation ratio [100, 122, 149]. When using RPC servers to implement microservices and running them in containers, inter-service RPCs pass through network sockets virtualized by the container runtime, via overlay networks [168]. Container overlay networks allow a set of related containers running on multiple host machines to use independent IP addresses, without knowing if other containers reside on the same host. While this approach works for the general case, even containers on the same host pay the processing costs of the full network stack.

On the other hand, Nightcore keeps most inter-service calls on the same host and uses Linux pipes for intra-host communications. Eliminating most inter-host networking explains Nightcore's performance advantage over containerized RPC servers in the distributed setting. But Nightcore also has a noticeable advantage over containerized RPC servers for intra-host communications, shown in Figure 3.6. To further understand this advantage, we collect stacktrace samples for both Nightcore and containerized RPC servers running with a single VM, and Table 3.6 summarizes the results. For RPC servers, TCP-related system calls and netrx softirq consume 47.6% of non-idle CPU time, both of which are used for

	RPC servers	**Nightcore**
do_idle	41.6%	60.4%
user space	18.3%	14.8%
irq / softirq		
− netrx	7.1%	6.8%
− others	2.0%	1.6%
syscall		
− tcp socket	20.7%	7.6%
− poll / epoll	2.5%	1.1%
− futex	2.2%	0.1%
− pipe	0%	3.7%
− unix socket	1.1%	0%
− others	3.1%	3.1%
uncategorized	1.4%	0.8%

Table 3.6: Breakdowns of stacktrace samples, when running SocialNetwork (write) at 1200 QPS on one VM. Unix sockets are used by Thrift RPC servers for inter-thread synchronizations.

inter-service communications. In contrast, Nightcore spends much less CPU time in TCP-related system calls, because only communication with services running on other hosts (e.g., database and Redis) uses TCP sockets. Both systems spend roughly the same amount of CPU time in netrx softirqs, which is caused only by inter-host networking.

3.4.4 Discussion

A goal for Nightcore is to avoid modifying Linux, because we want Nightcore to be easier to adopt for existing microservice workloads. Nightcore therefore relies on existing OS abstractions to achieve its performance goals, creating a challenge to efficiently use the operating systems' existing I/O abstractions and to find individual “killer microseconds.”

In our experience with Nightcore, we find there is no single dominant “killer microsecond.” There are multiple factors with significant contributions, and all must be addressed. Profiling the whole system for microsecond-scale optimization opportunities is challenging given the overheads introduced by profiling itself. In Nightcore, we implement low-overhead statistics collectors, and use eBPF programs [20] for kernel-related profiling.

3.5 Microservice background

Latency-sensitive interactive microservices. Online services must scale to high concurrency, with response times small enough (a few tens of milliseconds) to deliver an interactive experience [78, 94, 157]. Once built with monolithic architectures, interactive online services are undergoing a shift to microservice architectures [1, 4, 5, 53, 59], where a large application is built by connecting loosely coupled, single-purpose microservices. On the one hand, microservice architectures provide software engineering benefits such as modularity and agility as the scale and complexity of the application grows [45, 62]. On the other hand, staged designs for online services inherently provide better scalability and reliability, as shown in pioneering works like SEDA [156]. However, while the interactive nature of online services implies an end-to-end service-level objectives (SLO) of a few tens of milliseconds, individual microservices face more strict latency SLOs – at the sub-millisecond-scale for leaf microservices [148, 165].

Microservice architectures are more complex to operate compared to monolithic architectures [31, 44, 45], and the complexity grows with the number of microservices. Although microservices are designed to be loosely coupled, their failures are usually very dependent. For example, one overloaded service in the system can easily trigger failures of other services, eventually causing cascading failures [3]. Overload control for microservices is difficult because microservices call each other on data-dependent execution paths, creating dynamics that cannot be predicted or controlled from the runtime [48, 60, 128, 166]. Microservices are often comprised of services written in different programming languages and frameworks, further complicating their operational problems. By leveraging fully managed cloud services (e.g., Amazon's DynamoDB [6], ElasticCache [7], S3 [24], Fargate [14], and Lambda [17]), responsibilities for scalability and availability (as well as operational complexity) are mostly shifted to cloud providers, motivating *serverless microservices* [26, 42, 52, 55–57, 68, 69].

Serverless microservices. Simplifying the development and management of online services is the largest benefit of building microservices on serverless infrastructure. For example, scaling the service is automatically handled by the serverless runtime, deploying a new version of code is a push-button operation, and monitoring is integrated with the platform (e.g., CloudWatch [2] on AWS). Amazon promotes serverless microservices with the slogan "no server is easier to manage than no server" [56]. However, current FaaS systems have high runtime overheads (Table 3.1) that cannot always meet the strict latency requirement imposed by *interactive* microservices. Nightcore fills this performance gap.

Nightcore focuses on mid-tier services implementing *stateless* business logic in microservice-based online applications. These mid-tier microservices bridge the user-facing frontend and the data storage, and fit naturally in the programming model of serverless functions. Online data intensive (OLDI) microservices [148] represent another category of microservices, where the mid-tier service fans out requests to leaf microservices for parallel data processing. Microservices in OLDI applications are mostly *stateful* and memory intensive, and therefore are not a good fit for serverless functions. We leave serverless support of OLDI microservices as future work.

The programming model of serverless functions expects function invocations to be short-lived, which seems to contradict the assumption of service-oriented architectures which expect services to be long-running. However, FaaS systems like AWS Lambda allows clients to maintain long-lived connections to their API gateways [9], making a serverless function "service-like". Moreover, because AWS Lambda re-uses execution contexts for multiple function invocations [15], users' code in serverless functions can also cache reusable resources (e.g., database connections) between invocations for better performance [21].

Optimizing FaaS runtime overheads. Reducing start-up latencies, especially cold-start latencies, is a major research focus for FaaS runtime overheads [77, 91, 97, 130, 132, 145]. Nightcore assumes sufficient resources have been provisioned and relevant function containers are in warm states which can be achieved on AWS Lambda by using provisioned concurrency (AWS Lambda strongly recommends provisioned concurrency for latency-critical functions [50]). As techniques for optimizing cold-start latencies [130, 132] become mainstream, they can be applied to Nightcore.

Invocation latency overheads of FaaS systems are largely overlooked, as recent studies on serverless computing focus on data intensive workloads such as big data analysis [110, 138], video analytics [80, 99], code compilation [98], and machine learning [92, 145], where function execution times range from hundreds of milliseconds to a few seconds. However, a few studies [89, 123] point out that the millisecond-scale invocation overheads of current FaaS systems make them a poor substrate for microservices with microsecond-scale latency targets. For serverless computing to be successful in new problem domains [101, 111, 123], it must address microsecond-scale overheads.

3.6 Summary

Nightcore is motivated by a realistic problem in today's cloud computing: making latency-sensitive microservices practical on FaaS. Frequent inter-service calls, strict μs-scale SLOs, and complicated dynamics between microservices make this problem particularly challenging. Nightcore overcomes these challenges by diverse techniques: optimizing locality of inter-service calls, low runtime overheads for common serverless events, and actively managing concurrency for resource efficiency. Nightcore is the first FaaS runtime achieving low-latency and high throughput, while preserving container isolation and the convenience of supporting diverse languages.

Although Nightcore does not introduce many new techniques for latency reduction, the experience of Nightcore provides takeaways that can inspire future studies on microservices and serverless computing:

- Nightcore demonstrates the runtime overhead of FaaS can be optimized to μs-scale by a careful implementation just in the software layer;
- In a FaaS system, locality of function calls is important to low latency (which is required by microservices), though this creates challenges for the function scheduler;
- Microservices can be more efficient on FaaS than traditional approaches such as RPC servers.

Optimizing Nightcore justifies one of Lampson's early hints [121]: "make it fast, rather than general or powerful", because fast building blocks can be used more widely. As computing becomes more granular [123], we anticipate more microsecond-scale applications will come to serverless computing. Designing and building this next generation of services will require careful attention to microsecond-scale overheads. Nightcore is publicly available at GitHub: `https://github.com/ut-osa/nightcore`.

Chapter 4

Boki: Stateful Serverless Computing with Shared Logs

State management remains a major challenge in the current FaaS paradigm [104, 140, 146, 163]. Because of the stateless nature of serverless functions, current serverless applications rely on cloud storage services (e.g., Amazon S3 and DynamoDB) to manage their state. However, current cloud storage cannot simultaneously provide low latency, low cost, and high throughput [117, 138]. Relying on cloud storage also complicates data consistency in stateful workflows [160], because functions in a workflow could fail in the middle which leaves inconsistent workflow state stored in the database.

The shared log [83, 96, 154] is a popular approach for building storage systems that can simultaneously achieve scalability, strong consistency, and fault tolerance [25, 82, 84, 88, 102, 116, 152, 154]. A shared log offers a simple abstraction: a totally ordered log that can be accessed and appended concurrently. While simple, a shared log can efficiently support state machine replication [141], the

This chapter is based on the previous publication: "Boki: Stateful Serverless Computing with Shared Logs", by Zhipeng Jia and Emmett Witchel in the Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 2021), pages 691—707, 2021 [108]. This chapter also includes improvements of Boki which we have done after the SOSP publication. We have submitted a manuscript of Boki with these new improvements to the ACM Transactions on Computer Systems.

well-understood approach for building fault-tolerant stateful services [84, 154]. The shared log API also frees distributed applications from the burden of managing the details of fault-tolerant consensus, because the consensus protocol is hidden behind the API [82]. Providing shared logs to serverless functions can address the dual challenges of consistency and fault tolerance (§ 4.1).

This chapter presents Boki [1], a FaaS runtime that exports the shared log API to functions for storing shared state. Boki realizes the shared log API with a LogBook abstraction, where each function invocation is associated with a LogBook (§ 4.2). For a Boki application, its functions share a LogBook, allowing them to share and coordinate updates to state. In Boki, LogBooks enable stateful serverless applications to manage their state with durability, consistency, and fault tolerance.

The shared log API is simple to use and applicable to diverse applications [82, 84, 85, 154], so the challenge of Boki is to achieve high performance and strong consistency while conforming to the serverless environment (§ 4.1.2). Data locality is one challenge for serverless storage, because disaggregated storage is strongly preferred in the serverless environment [104, 140, 146]. Boki separates the read and write path, where read locality is optimized with a cache on function nodes and writes are optimized with scale-out bandwidth. Boki will scatter writes over variable numbers of shards while providing consistent reads and fault tolerance. In Boki, high performance, read consistency and fault tolerance are achieved by a single log-based mechanism, the *metalog*. The metalog contains metadata that totally

[1] Boki means bookkeeping in Japanese.

orders a log's data records. Because Boki uses a compact format for the metalog, durability and consensus are vital, but high data throughput is not. Therefore Boki stores and updates metalogs using a simple primary-driven design.

Boki handles machine failures by reconfiguration, similar to previous shared log systems [82, 96, 154]. Because the metalog controls Boki's internal state transitions, sealing the metalog (making it no longer writable) pauses state transitions. Therefore, Boki implements reconfiguration by sealing the metalog, changing the system configuration, and starting a new metalog.

Boki's metalog allows easy adoption of state-of-the-art techniques from previous shared log designs because it make log ordering, consistency, and fault tolerance into independent modules (§ 4.3.1). Boki adapts ordering from Scalog [96] and fault tolerance from Delos's [82] sealing protocol. Another benefit of the metalog is it decouples read consistency from data placement, enabling indices and caches for log records to be co-located with functions. Without interfering with read consistency, cloud providers can build simple caches which increase data locality when scheduling functions on nodes where their data is likely to be cached.

Boki's shared log designs are implemented on top of Nightcore (described in Chapter 3). Nightcore has no specialized mechanism for state management, Boki provides it; while Nightcore's design for I/O efficiency benefits Boki. Boki achieves append throughput of 1.2M Ops/s within a single LogBook, while maintaining a p99 latency of 6.3ms. With LogBook engines co-located with functions, Boki achieves a read latency of 86μs for best-case LogBook reads.

To ease the usage of Boki's LogBook API for end applications, we build support libraries on top of the LogBook API aimed at three different serverless use cases :fault-tolerant workflows (BokiFlow), durable object storage (BokiStore), and serverless message queues (BokiQueue). Our evaluations of Boki support libraries show (§ 4.6):

- BokiFlow executes workflows 3.8–4.2× faster than Beldi [160];
- BokiStore achieves 1.20–1.28× higher throughput than MongoDB, while executing transactions 2.0–2.5× faster;
- BokiQueue achieves 2.16× higher throughput and up to 17× lower latency than Amazon SQS [8], while achieving 1.26× higher throughput and up to 1.6× lower latency than Apache Pulsar [10].

4.1 Shared log approach for stateful serverless

In the current FaaS paradigm, stateful applications struggle to achieve fault tolerance and strong consistency of their critical state. For example, consider a travel reservation app built with serverless functions. This app has a function for booking hotels and another function for booking flights. When processing a travel reservation request, both functions are invoked, but both functions can fail during execution, leaving inconsistent state. Using current approaches for state management such as cloud object stores or even cloud databases, it is difficult to ensure the consistency of the reservation state given the failure model [160].

The success of log-based approaches for data consistency and fault tolerance motivates the usage of shared logs for stateful FaaS. For example, Olive [142] proposes a client library interacting with cloud storage, where a write-ahead redo log is used to achieve exactly-once semantics in face of failures. Beldi [160] extends Olive's log-based techniques for transactional serverless workflows. State machine replication (SMR) [141] is another general approach for fault tolerance, where application state is replicated across servers by a command log. The command log is traditionally backed by consensus algorithms [131, 133, 151]. But recent studies demonstrate a shared log can provide efficient abstraction to support SMR-based data structures [84, 154] and protocols [82, 85]. Boki provides shared logs to serverless functions, so that Boki's applications can leverage well-understood log-based mechanisms to efficiently achieve data consistency and fault tolerance.

4.1.1 Use cases

By examining demands in serverless computing, we identify three important cases where shared logs provide a solution. Boki provides support libraries for these use cases (§ 4.4).

Fault-tolerant workflows. Workflows orchestrating stateful functions create new challenges for fault tolerance and transactional state updates. Beldi [160] addresses these challenges via logging workflow steps. Beldi builds an atomic logging layer on top of DynamoDB. We adapt Beldi's techniques to the LogBook API without building an extra logging layer.

Durable object storage. Previous studies like Tango [84] and vCorfu [154] demonstrate that shared logs can support high-level data structures (i.e., objects), that are consistent, durable, and scalable. Motivated by Cloudflare's Durable Objects [72], we build a library for stateful functions to create durable JSON objects. Our object library is more powerful than Cloudflare's because it supports transactions across objects, using techniques from Tango [84].

Serverless message queues. One constraint in the current FaaS paradigm is that functions cannot directly communicate with each other via traditional approaches [98], e.g., network sockets. Shared logs can naturally be used to build message queues [96] that offer indirect communication and coordination among functions. We build a queue library that provides shared queues among serverless functions.

4.1.2 Technical challenges

While prior shared log designs [82, 83, 96, 154] provide inspiration, the serverless environment creates new challenges.

Elasticity and data locality. Serverless computing strongly benefits from disaggregation [79, 101], which offers elasticity. However, current serverless platforms choose physical disaggregation, which reduces data locality [104, 146]. Boki achieves both elasticity and data locality, by decoupling the read and the write paths for log data and co-locating read components with functions.

Resource efficiency. Boki aims to support a high density of LogBooks efficiently, so it multiplexes many LogBooks on a single physical log. Multiplexing LogBooks can address performance problems that arise from a skewed distribution of LogBook sizes. But this approach creates a challenge for LogBook reads: how to locate the records of a LogBook. Boki proposes a log index to address this issue, with the metalog providing the mechanism for read consistency (§ 4.3.4).

The ephemeral nature of FaaS. Shared logs are used for building high-level data structures via state machine replication (SMR) [84, 154]. To allow fast reads, clients keep in-memory copies of the state machines, e.g., Tango [84] has local views for its SMR-based objects. However, serverless functions are ephemeral – their in-memory state is not guaranteed to be preserved between invocations. This limitation forces functions to replay the full log when accessing a SMR-based object. Boki introduces auxiliary data (§ 4.2) to enable optimizations like local views in Tango (§ 4.4.4). Auxiliary data are designed as cache storage on a per-log-record basis, while their relaxed durability and consistency guarantees allow a simple and efficient mechanism to manage their storage (§ 4.3.4).

4.2 Boki's LogBook API

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

```
struct LogRecord {
   uint64_t seqnum;      string data;
   vector<tag_t> tags;   string auxdata;
};

// Append a new log record.
status_t logAppend(vector<tag_t> tags, string data, uint64_t* seqnum);

// Read the next/previous record whose seqnum >= ‘min_seqnum‘,
// or <= ‘max_seqnum‘.
status_t logReadNext(uint64_t min_seqnum, tag_t tag, LogRecord* record);
status_t logReadPrev(uint64_t max_seqnum, tag_t tag, LogRecord* record);

// Alias of logReadPrev(kMaxSeqNum, tag, record).
status_t logCheckTail(tag_t tag, LogRecord* record);

// Trim the LogBook until ‘trim_seqnum‘, i.e., delete all log records
// whose seqnum < ‘trim_seqnum‘.
status_t logTrim(uint64_t trim_seqnum);

// Set auxiliary data for the record of ‘seqnum‘.
status_t logSetAuxData(uint64_t seqnum, string auxdata);
```

Figure 4.1: Boki’s LogBook API (§ 4.2).

with multiple function invocations, so that applications can share state among their function instances.

Like previous shared log systems [82, 83, 96, 154], Boki exposes *append*, *read*, and *trim* APIs for writing, reading, and deleting log records. Figure 4.1 lists Boki’s LogBook API.

Read consistency. LogBook guarantees *monotonic reads* and *read-your-writes* when reading records. These guarantees imply a function has a monotonically increasing view of the log tail. Moreover, a child function inherits its parent function’s view of the log tail, if two functions share the same LogBook. This property is important for serverless applications that compose multiple functions (§ 4.3.4).

Sequence numbers (seqnum). The `logAppend` API returns a unique seqnum for the newly appended log record. The seqnums determine the relative order of records within a LogBook. They are monotonically increasing but *not* guaranteed to be consecutive. Boki's `logReadNext` and `logReadPrev` APIs enable bidirectional log traversals, by providing lower and upper bounds for seqnums (§ 4.3.2).

Log tags. Every log record has a set of tags, that is specified in `logAppend`. Log tags enable selective reads, where only records with the given tag are considered (see the `tag` parameter in `logReadNext` and `logReadPrev` APIs). Records with same tags form abstract streams within a single LogBook. Having sub-streams in a shared log for selective reads is important for reducing log replay overheads, that is used in Tango [84] and vCorfu [154] (§ 4.3.4).

Auxiliary data. LogBook's auxiliary data is designed as per-log-record cache storage, which is set by the `logSetAuxData` API. Log reads may return auxiliary data along with normal data if found. Auxiliary data can cache object views in a shared-log-based object storage. These object views can significantly reduce log replay overheads (§ 4.4.4).

As auxiliary data is designed to be used only as a cache, Boki does not guarantee its durability, but provides best effort support. Moreover, Boki does not maintain the consistency of auxiliary data, i.e., Boki trusts applications to provide consistent auxiliary data for the same log record. Relaxing durability and consistency allows Boki to have a simple yet efficient backend for storing auxiliary data (§ 4.3.4).

4.3 Boki design

Boki's design combines a FaaS system with shared log storage. Boki internally stores multiple independent, totally ordered logs. User-facing LogBooks are multiplexed onto internal physical logs for better resource efficiency (§ 4.1.2). A Boki physical log has an associated *metalog*, playing the central role in ordering, consistency, and fault tolerance.

4.3.1 Metalog is "the answer to everything" in Boki

Every shared log system must answer three questions because they store log records across a group of machines. The first is how to determine the global total order of log records. The second is how to ensure read consistency as the data are physically distributed. The third is how to tolerate machine failures. Table 4.1 shows different mechanisms used by previous shared log systems to address these three issues, whereas in Boki, the *metalog* provides the single solution to all of them.

In Boki, every physical log has a single associated *metalog*, to record its internal state transitions. Boki sequencers append to the metalog, while all other components subscribe to it. In particular, appending, reading, and sealing the

	Ordering log records	**Read consistency**	**Failure handling**
vCorfu	A dedicated sequencer	Stream replicas	Hole-filling protocol
Scalog	Paxos and aggregators	Sharding policy	Paxos
Boki	Appending *metalog* entries	Tracking *metalog* positions	Sealing the *metalog*

Table 4.1: Comparison between vCorfu [154], Scalog [96], and Boki. Boki's metalog provides a unified approach for log ordering, read consistency, and fault tolerance (§ 4.3.1).

metalog provide mechanisms for log ordering, read consistency, and fault tolerance:

- *Log ordering.* The primary sequencer appends metalog entries to decide the total order for new records, using Scalog [96]'s high-throughput ordering protocol. (§ 4.3.3)

- *Read consistency.* Different LogBook engines update their log indices independently, however, read consistency is enforced by comparing metalog positions. (§ 4.3.4)

- *Fault tolerance.* Boki is reconfigured by sealing metalogs, because a sealed metalog pauses state transitions for the associated log. When all current metalogs are sealed, a new configuration can be safely installed. (§ 4.3.5)

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

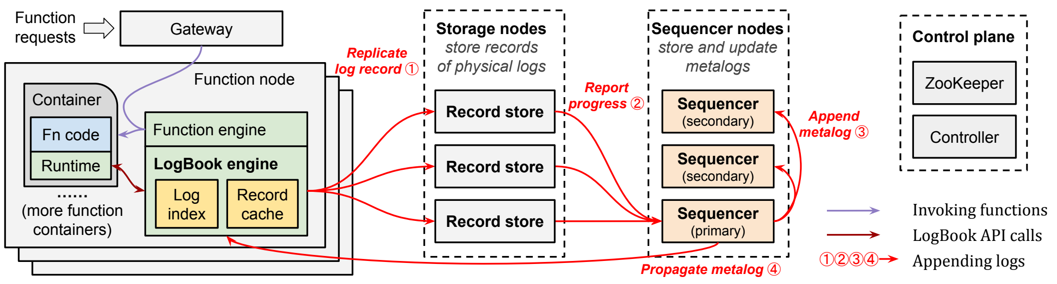


Figure 4.2: Architecture of Boki (§ 4.3.2), where red arrows show the workflow of log appends (§ 4.3.3).

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

4.3.2 Architecture

Figure 4.2 depicts Boki's architecture, which is based on Nightcore (Chapter 3), a state-of-the-art FaaS system for microservices. In Nightcore's design, there is a gateway for receiving function requests and multiple function nodes for running serverless functions. On each function node, an engine process communicates with the Nightcore runtime within function containers via low-latency message channels.

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

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

Sequencer nodes. Sequencer nodes run Boki sequencers that store and update metalogs using a primary-driven protocol (see § 4.3.1). Sequencers append new metalog entries to order physical log records as detailed in § 4.3.3. Similar to storage nodes, individual sequencer nodes can be configured to back different metalogs.

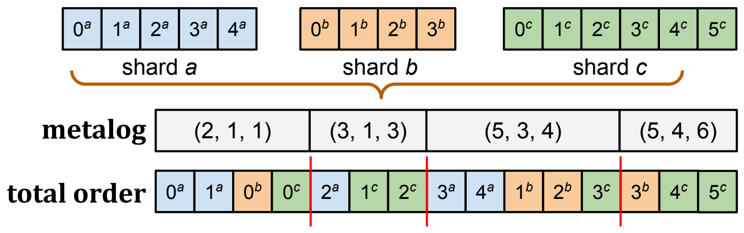
LogBook engines. In Nightcore, the engine processes running on function nodes are responsible for dispatching function requests. Boki extends Nightcore's engine by adding a new component serving LogBook calls. We refer the new part as LogBook engine, to distinguish it from the part serving function requests.

LogBook API requests are forwarded to LogBook engines by Boki's runtime, which is linked with user supplied function code. LogBook engines maintain indices for physical logs, in order to efficiently serve LogBook reads (detailed in § 4.3.4). LogBook engines subscribe to the metalog, and incrementally update their indices in accordance with the metalog. LogBook engines also cache log records for faster reads, using their unique sequence numbers as keys. Co-locating LogBook engines with functions means that, in the best case, LogBook reads can be served without leaving the function node.

Control plane. Boki's control plane uses ZooKeeper [105] for storing its configuration. Boki's configuration includes (1) the set of storage, sequencers, and indices constituting each physical log; (2) addresses of gateway, function, storage, and sequencer nodes; (3) parameters of consistent hashing [114] used for the mapping between LogBooks and physical logs. Every Boki node maintains a ZooKeeper session to keep synchronized with the current configuration. ZooKeeper sessions are also used to detect failures of Boki nodes.

Boki's controller (see the control plane in Figure 4.2) is responsible for global reconfiguration. Reconfiguration happens when node failures are detected, or when instructed by the administrator to scale the system, e.g., by changing the number of physical logs (see § 4.6.1 for reconfiguration latency measurements). We define the duration between consecutive reconfigurations as a *term*. Terms have a monotonically increasing *term_id*.

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



0^a 1^a 2^a 3^a 4^a
shard *a*
0^b 1^b 2^b 3^b
shard *b*
0^c 1^c 2^c 3^c 4^c 5^c
shard *c*
metalog
(2, 1, 1)
(3, 1, 3)
(5, 3, 4)
(5, 4, 6)
total order
0^a 1^a 0^b 0^c 2^a 1^c 2^c 3^a 4^a 1^b 2^b 3^c 3^b 4^c 5^c

Figure 4.3: An example showing how the metalog determines the total order of records across shards. Each metalog entry is a vector, whose elements correspond to shards. In the figure, log records between two red lines form a delta set, which is defined by two consecutive vectors in the metalog (§ 4.3.3).

within a LogBook to be consecutive, whose records can be physically interspersed with other LogBooks.

4.3.3 Workflow of log appends

When appending a LogBook (shown by the red arrows in Figure 4.2), the new record is appended to the associated physical log. For simplicity, in this section, the term *log* always refers to physical logs.

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

The LogBook engine replicates a new record to all storage nodes backing its shard (① in Figure 4.2). Storage nodes then need to update the sequencers with the information of what records they have stored. The monotonic nature of *local_id* enables a compact progress vector, v. Suppose the log has M shards. We use a vector v of length M to represent a set of log records. The set consists of, for all shards j, records with *local_id* $< v^j$. If shard j is not assigned to this node, we set the j-th element of its progress vector as ∞. Every storage node maintains their progress vectors, and periodically communicates them to the primary sequencer (② in Figure 4.2).

By taking the element-wise minimum of progress vectors from all storage nodes, the primary sequencer computes the global progress vector. Based on the definition of progress vectors, we can see the global progress vector represents the set of log records that are fully replicated. Finally, the primary sequencer periodically appends the latest global progress vector to the metalog (③ in Figure 4.2), which effectively orders log records across shards.

We now explain how the total order is determined by the metalog. Consider a newly appended global progress vector, denoted by v_i. By comparing it with the previous vector in the metalog (denoted by v_{i-1}), we can define the delta set of log records between these two vectors: for all shards j, records satisfying $v^j_{i-1} \leq$ *local_id* $< v^j_i$. This delta set exactly covers log records that are added to the total order by the new metalog entry v_i. Records within a delta set are ordered by (*shard*, *local_id*). Figure 4.3 shows an example of metalog and its corresponding total order. In this figure, between two consecutive red lines is a delta set.

The LogBook engine initiating the append operation learns about its completion by its subscription to the metalog (④ in Figure 4.2). The metalog allows the LogBook engine to compute the final position of the new record in the log, used to

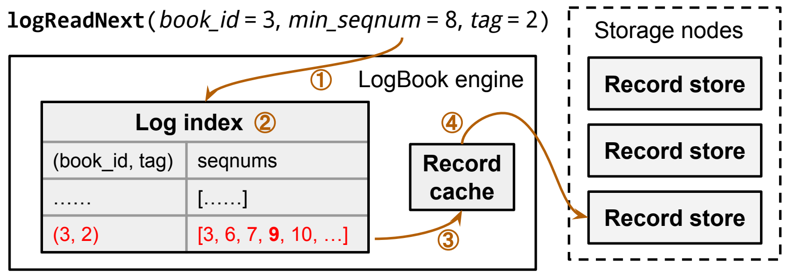


Figure 4.4: Workflow of LogBook reads (§ 4.3.4): ① Locate a LogBook engine stores the index for the physical log backing *book_id* $= 3$; ② Query the index row $(book\_id, tag) = (3, 2)$ to find the metadata of the result record (seqnum $= 9$ in this case); ③ Check if the record is cached; ④ If not cached, read it from storage nodes.

construct the sequence number returned by `logAppend`.

4.3.4 From physical logs to LogBooks

Building indices for LogBooks. Boki virtualizes LogBooks by multiplexing them on physical logs, which creates a problem for efficient reads – avoiding consulting every log shard. Previous systems [154] have used fixed sharding, where a LogBook maps to some fixed shard, so that a single storage node has all of its records. But then a single storage node becomes the bottleneck for a LogBook's write throughput. For performance and operational advantages, Boki does not place records from a LogBook using a fixed policy. Boki will store LogBook records in any shard and it builds a log index for locating records when reading LogBooks.

Boki's log index is compact, only including necessary metadata of log records, so that a single machine can store the entire index. Log indices are stored and maintained by LogBook engines, leading to locality benefits because LogBook

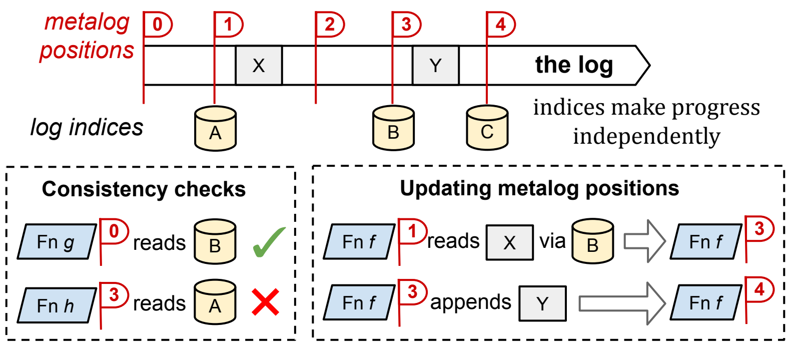


Figure 4.5: Consistency checks by comparing metalog positions (§ 4.3.4). For a function, if reading from a log index whose progress is behind its metalog position, it could see stale states. For example, function h has already seen record X, so that it cannot perform future log reads through index A.

engines reside on function nodes. Every physical log has multiple copies of the log index maintained by different LogBook engines, for higher read throughput and better read locality.

The structure of the log index is designed to fit the semantic of LogBook read APIs. First, the log index groups records by their *book_id*, because a read can only target a single LogBook. The API semantics for `logReadNext` and `logReadPrev` (see Figure 4.1) allow selective reads by log tags (tags are specified by users in `logAppend`). Both APIs seek for records sequentially by providing bounds for seqnums, e.g., `logReadNext` finds the first record whose seqnum $\geq$ *min_seqnum*. Putting them together, Boki's log index groups records by (*book_id*,*tag*). For each (*book_id*,*tag*), it builds an index row as an array of records, sorted by their seqnums. Figure 4.4 depicts the workflow of LogBook reads using the index.

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

The challenge of enforcing read consistency comes from multiple copies of the log index, which are maintained by different LogBook engines. Keeping these copies consistent makes the system vulnerable to "slowdown cascades" [76, 129], i.e., the slowdown of a single node can prevent the whole system from making progress.

Boki uses *observable consistency* [93, 129], where consistency checks are delayed to the time of data reads. The metalog position defines the version of the log index a function reads. A log index whose version is determined by metalog position l means the log index includes all records ordered by the l-prefix of the metalog.

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

Therefore, Boki maintains a metalog position for each function and that position provides consistent LogBook reads. LogBook engines subscribe to the metalog to periodically update their indices. Consistency checks are performed by comparing a function's metalog position with the index version. Figure 4.5 depicts the mechanism. If a consistency check fails, the read is suspended by the engine until its index has caught up. Successful reads and appends from a function update the function's metalog position, ensuring the consistency of future reads. A child function inherits the metalog position from its parent function, so that consistency guarantees hold across function boundaries.

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

Auxiliary data. Described in the LogBook API (§ 4.2), the auxiliary data of log records have relaxed requirements of durability and consistency. This allows a very simple store of auxiliary data that reuses the record cache within LogBook engines. The relaxed consistency of auxiliary data does not even require Boki to exchange them between nodes. Therefore, for `logSetAuxData` calls, Boki simply caches the provided auxiliary data on the same function node. To serve reads from the user function Boki checks if there is auxiliary data in the local cache. If found, it is returned along with the result record.

4.3.5 Reconfiguration protocol

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

The main part of Boki's reconfiguration protocol is to seal all current metalogs. A sealed metalog cannot have any more entries appended, so the corresponding physical log is sealed as well. Boki employs Delos [82]'s log sealing protocol, that is surprisingly simple but fault-tolerant. To seal a metalog, the controller sends the seal command to all relevant sequencers. On receiving the seal command, the primary sequencer stops issuing new metalog entries, while secondary sequencers commit to reject future metalog entries from the primary sequencer. The sealing is completed when a quorum of sequencers have acknowledged the seal command (see the Delos paper [82] for details).

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

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

4.4 Boki support libraries

In this section, we present Boki support libraries, designed for three different stateful FaaS paradigms that benefit from the LogBook API: fault-tolerant workflows (§ 4.4.1), durable object storage (§ 4.4.2), and queues for message passing (§ 4.4.3)

4.4.1 BokiFlow: fault-tolerant workflows

We build a support library called BokiFlow for fault-tolerant workflows. BokiFlow adapts Beldi [160]'s techniques to ensure exactly-once semantics and support transactions for serverless workflows.

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

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

```
def write(table, key, val):
    # Append write-ahead log for this DB update
    tag = hashLogTag([ID, STEP])
    logAppend(tags: [tag], data: [table, key, val])

    # Always consider the first log record for this step,
    # so that during workflow re-execution the original log
    # record is used
    record = logReadNext(tag: tag, minSeqnum: 0)

    # The write-ahead log also determines a total order for
    # DB writes, where sequence numbers of log records are
    # used as "version numbers"
    rawDBWrite(table, key,
        cond: "Version < {record.seqnum}",
        update: "Value = {val}; Version = {record.seqnum}")
    STEP = STEP + 1

def invoke(callee, input):
    # Generate UUID for child function and store it
    # in pre-invoke log record
    tagPre = hashLogTag([ID, STEP, "pre"])
    logAppend(tags: [tagPre], data: {"calleeId": UUID()})

    # Read calleeId from log record for child function,
    # so that we use original UUID during re-execution
    record = logReadNext(tag: tagPre, minSeqnum: 0)
    calleeId = record.data["calleeId"]

    # Invoke child function with the given input
    retVal = rawInvoke(callee, [calleeId, input])

    # Post-invoke record stores return value of child function
    tagPost = hashLogTag([ID, STEP, "post"])
    logAppend(tags: [tagPost], data: {"retVal": retVal})

    record = logReadNext(tag: tagPost, minSeqnum: 0)
    STEP = STEP + 1
    return record.data["retVal"]
```

Figure 4.6: Pseudocode demonstrating BokiFlow's write and invoke functions (§ 4.4.1.3). `hashLogTag` computes a hashing-based log tag for the provided tuple. Variable `ID` stores the unique ID of the current workflow. Variable `STEP` stores the step number, which is increased by 1 for every operations within the workflow.

4.4.1.1 Distinctions between BokiFlow and Beldi.

Because the LogBook API is very different from a cloud database API, BokiFlow needs new techniques to address issues caused by these differences. There are three ways BokiFlow distinguishes itself from Beldi.

Atomic "test-and-append". Beldi requires an atomic operation to check if the current step is previously logged and it logs the step only if the check fails. Beldi relies on conditional updates provided by a cloud database for this operation. Unfortunately, the LogBook API does not support conditional log appends. Shown in Figure 4.6, BokiFlow uses a different mechanism based on log tags provided by LogBooks. The pseudocode shows how BokiFlow uses log tags to distinguish the log records of workflow steps. BokiFlow always reads log records immediately after appends, and only honors the first record of a step. This allows BokiFlow to recognize completed steps during workflow re-execution, by checking if the appended record is the first one.

Idempotent database update. For a workflow step that updates the database, Beldi requires the database update and logging of this step to be a single atomic operation. Because Beldi stores its logs along with user data in the same database, it can use the atomic scope provided by the database (e.g., a row in DynamoDB) for this requirement. However, BokiFlow's LogBook is not in the same atomic scope as user data, so no mechanism exists to update both in a single atomic operation. Instead, BokiFlow makes data updates *idempotent*. Pseudocode in Figure 4.6 demonstrates the approach, where the `rawDBWrite` statement uses the sequence number of the step log as the "version" of the database update. During workflow re-execution, re-executing this database update will fail the update condition.

Locks. Beldi provides locks for mutual exclusion; locks also serve as building blocks for Beldi's transactions (§ 4.4.1.2). Implementing locks requires an atomic "test-and-set" operation, where Beldi uses conditional updates provided by the database. BokiFlow implements locks as registers backed by replicated state machines using the LogBook API. For a BokiFlow lock, its register stores the lock holder (unique identifiers such as UUID), or a special `EMPTY` value. The most natural way to "test" a lock is to execute a predicate on the current state machine. The most natural "set" is to append an update. When we try to combine these operations into a "test-and-set", the LogBook API cannot linearize the result because other BokiFlow clients may also append updates to the same state machine. BokiFlow's solution is to include the log position of the current state machine in the log record of the proposed update. On log replay, only choose the first of any updates that were concurrently proposed. In this way, the total order provided by the LogBook API becomes a mechanism for linearizability.

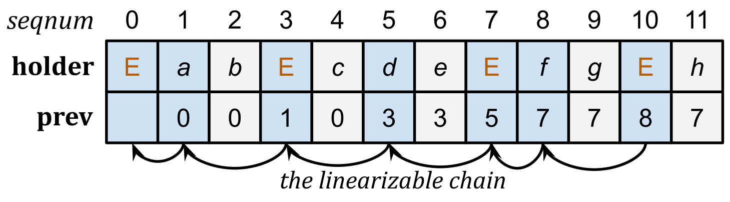
Pseudocode in Figure 4.7a demonstrates BokiFlow locks. The lock uses the *prev* field to store the log position, as shown in Figure 4.7. The "prev" pointers form a linearizable chain of state machine updates. This technique provides a general approach for building linearizable replicated state machines with the LogBook API.

```
def checkLockState(key):
    tail = None         # Tail of the "linearizable chain"
    nextSeqnum = 0
    while True:
        record = logReadNext(tag: key, minSeqnum: nextSeqnum)
        if record == None:
            break       # No more log record for this lock
        if tail == None or record.data["prev"] == tail.seqnum:
            # This record is part of the linearizable chain
            tail = record
        nextSeqnum = record.seqnum + 1
    return tail

def tryLock(key, holderId):
    record = checkLockState(key)
    if record.data["holder"] == EMPTY:
        # Presume the lock is not held, and append log
        # record to acquire
        logAppend(tags:[key], data:{"holder": holderId,
                                    "prev": record.seqnum})
        record = checkLockState(key)
    if record.data["holder"] == holderId:  # Lock succeeded
        # The acquire record will be used for unlock later
        return record
    return None    # Lock failed

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

(a) Pseudocode of BokiFlow's lock operations.



seqnum 0 1 2 3 4 5 6 7 8 9 10 11
holder E a b E c d e E f g E h
prev 0 0 1 0 3 3 5 7 7 8 7
the linearizable chain

(b) An example log behind a BokiFlow lock. Holders $\{a, d, f\}$ acquire the lock. *prev* pointers in the log form an implicit linearizable chain, which alternates successful acquire and release attempts. Holders share the same *prev* pointers, e.g., holders $\{a, b, c\}$, mean they try to acquire the lock concurrently. Also note holder c's tryLock record is after holder a's lock release record, which is a valid outcome from interleaving.

Figure 4.7: Locks in BokiFlow (§ 4.4.1).

4.4.1.2 Transactions in BokiFlow

Same as Beldi, BokiFlow supports transactions that can span across function boundaries with a workflow. Transactions provide stronger guarantees than the workflow's exactly-once semantic: 1. transactions guarantee isolation so that they will not observe data writes from other concurrently running workflows; 2. transactions can be aborted and data writes made within an aborted transaction will never be observed by another workflow.

When initiating a transaction, BokiFlow assigns a unique *txn_id* for it and creates a log for recording operations made within this transaction. Records of transaction logs are appended to the same LogBook used by workflow logs, but use *txn_id* as log tags. These transaction identifiers are passed between functions if the transaction spans across function boundaries.

On committing a transaction, BokiFlow follows the transaction log to commit data writes to DynamoDB. Similar to Beldi, BokiFlow uses locks to guarantee consistency and isolation. BokiFlow maintains per-key locks for tables in DynamoDB. Within a transaction, locks are acquired for database read and write operations. To prevent deadlock, acquiring a lock is a non-blocking operation.

4.4.1.3 Walk-through of BokiFlow operations

BokiFlow inherits Beldi's log-based techniques for exactly-once execution semantics [142, 160]. In § 4.4.1.1, we discussed the distinctions between Beldi and BokiFlow. To make it easier to understand BokiFlow's techniques without prior knowledge of Beldi, we will walk through BokiFlow's write and invoke operations in detail, based on pseudocode in Figure 4.6.

We first explain the write operation. `write`(*table*, *key*, *val*) updates a key-value pair in a DynamoDB table. Same as in Beldi, database writes are logged before execution for fault tolerance. Line 4 in Figure 4.6 shows the write-ahead logging, where the log record is tagged by the workflow ID and the current step number. The log tag allows BokiFlow to uniquely distinguish this write operation. When failure happens and the workflow is re-executed, `logAppend` (line 4) will be executed again, appending a new and different log record. But the next `logReadNext` (line 8) will read the original record because it looks for the *smallest* sequence number. This append-then-read approach can even guarantee consistency under an extreme condition where concurrent instances of the same workflow are executing, caused by, e.g., unreliable detections of workflow failures. Finally, `rawDBWrite` (line 12) performs the real DynamoDB writes. We previously explain BokiFlow uses sequence numbers to version writes for the effect of idempotent updates. Because sequence numbers are critical for achieving idempotence, it is important that during workflow re-execution, the same log record must be used for each write operation.

We then look at the invoke operation that calls child functions in a workflow. Unlike the write operation, the invoke operation needs two log records: one before invoking the child function, and the other after invoking. Same as Beldi, BokiFlow generates a unique ID for each function within a workflow, and function IDs must be preserved during workflow re-executions to ensure deterministic recovery. The pre-invoke log record (line 20 in Figure 4.6) stores the unique ID for the child function, so that the previous function ID can be retrieved during workflow re-execution (line 24). The post-invoke log record (line 29) stores the return value of the child function call. Pseudocode in Figure 4.6 shows that *invoke* always calls into the child function even during workflow re-execution, which can be redundant. Our implementation will optimize this redundancy by checking the post-invoke log record before calling into the child function. If the post-invoke log record exists, we can immediately return with the *retVal* in the log record.

4.4.2 BokiStore: durable object storage

The second support library we built is BokiStore, providing durable object storage for stateful functions. BokiStore employs Tango's [84] techniques for building replicated data structures over a shared log. BokiStore's objects are represented as JSON objects. Objects are identified by unique string names. Figure 4.8 shows the BokiStore APIs for reading and modifying fields of JSON objects. BokiStore stores all object updates within a LogBook. Reading object fields requires replaying the log to re-construct the object's state. Log records containing object updates are tagged with object names, so that objects can be re-constructed by only reading relevant records.

Transactions. BokiStore supports transactions for reading and modifying multiple objects. BokiStore's log-based transaction protocol largely follows Tango. To start a transaction, BokiStore first appends a *txn_start* record with its *txn_id*. For all subsequent object reads within the transaction, BokiStore only replays the log

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

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

Figure 4.8: Demonstration of BokiStore API (§ 4.4.2).

up to the position of its *txn_start* record. This essentially takes a snapshot of the entire object storage at the *txn_start* position, which achieves *snapshot isolation*.

When committing the transaction, BokiStore appends a *txn_commit* record, including its *txn_id* and all object writes made within the transaction. The *txn_commit* record is *speculative* – by itself, it does not indicate the success of this transaction. The commit outcome of a transaction is determined by replaying the log up to its *txn_commit* record. A transaction succeeds in committing if and only if there is no conflicting write made between its *txn_start* and *txn_commit* records (i.e., within the conflict window). Figure 4.9 depicts a transaction log. In this example, *TxnB* is a failed transaction and it is ignored when determining the commit outcome of *TxnC*.

Transactions in BokiStore can be marked as read-only when creation (see

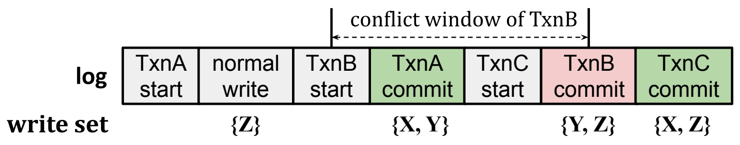


Figure 4.9: Transactions in BokiStore (§ 4.4.2). *TxnB* fails due to conflict with *TxnA*. For *TxnC*, despite its write set overlaps with *TxnB*'s, *TxnC* still succeeds due to the failure of *TxnB*.

Figure 4.8). This feature makes read-only transactions simpler to implement: they do not need to append actual *txn_start* and *txn_commit* records, because there is no need for conflict detection. To achieve isolation, when starting the transaction, BokiStore simply caches the current tail position of the log, instead of appending a real *txn_start* record. For object reads within the transaction, BokiStore only replays log records until the cached position.

4.4.3 BokiQueue: message queues

Queues are the most common data structure for message passing. The final support library we build is BokiQueue which provides serverless queues. BokiQueue provides a push and pop API for sending and receiving messages. Like BokiStore, BokiQueue uses the log to store all writes, i.e., push and pop operations. The outcome of a pop is determined by replaying the log. To improve the scalability of BokiQueue, we use vCorfu [154]'s composable state machine replication (CSMR) technique, that divides a single queue into multiple SMR-backed queue shards. Each queue shard is consumed by a single consumer, which reduces contention. A queue producer can choose an arbitrary queue shard to push. In our implementation, we simply use round-robin.

4.4.4 Optimizing log replay with auxiliary data

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

Boki's auxiliary data (§ 4.2) is motivated by the need to provide per-log-record cache storage. In BokiStore, for every object write that generates a log record, the auxiliary data of the record stores a snapshot view of the object. When reading an object, BokiStore seeks from the log tail to find the first relevant record having a cached object view in its auxiliary data. Then BokiStore replays the log from this position to re-construct the target object state. During replay, for records missing cached object views, their auxiliary data are filled with object views. Figure 4.10 demonstrates this accelerated replay process.

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

In BokiFlow's log-based locks (shown in Figure 4.7a), auxiliary data of a record is used for caching the current tail of the linearizable chain. This allows the

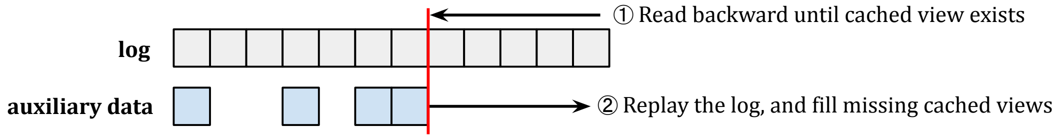

① Read backward until cached view exists
log
auxiliary data
② Replay the log, and fill missing cached views

Figure 4.10: Use auxiliary data to cache object views in BokiStore, which can avoid a full log replay (§ 4.4.4).

`checkLockState` function to optimize its log replay as illustrated in Figure 4.10.

4.4.5 Garbage collector functions

The FaaS paradigm simplifies garbage collection (GC) in shared-log-based storage systems. Boki support libraries use garbage collector functions to trim useless log records, in order to prevent unlimited growth of LogBooks. These functions are periodically invoked and they reclaim space via LogBook's `logTrim` API (Figure 4.1). The `logTrim` API trims a prefix of the log: it takes a single parameter *trim_seqnum* and deletes all log records with sequence numbers less than *trim_seqnum*. Given the API semantic, garbage collector functions have to efficiently figure out the safe trim position. We then describe specific mechanisms used by different Boki support libraries.

BokiFlow. BokiFlow follows Beldi's GC strategy [160]: the garbage collector function scans for completed workflows whose completion timestamp is old enough, and marks these workflows as recyclable. When a prefix of the log only contains records from recyclable workflows, `logTrim` can be called to reclaim space.

BokiStore. In BokiStore, the log stores a history of writes for individual objects. The garbage collector function periodically materializes object states in the log, so that log records corresponding to the old history can be safely deleted.

To scale this strategy with more objects, BokiStore uses multiple GC functions for materializing object states in parallel. Each GC function is responsible for a shard of objects, and the shard numbers are determined by hashing object names.

One of the GC functions is designated as *master*, who is responsible for actually calling `logTrim`. Other GC functions periodically store safe trim positions for their shards in the log (with some special tag), so that the *master* can determine the global trim position. The *master* GC function also takes extra care to ensure the trim position is not within any ongoing transactions.

BokiQueue. In BokiQueue, each queue shard is consumed in FIFO order, where log records of popped elements become useless. For each queue shard, its consumer can determine the safe trim position, and periodically stores the position in the log with some special tag. A dedicated GC function reads trim positions from all shards, and calls `logTrim` accordingly.

4.5 Implementation

The Boki prototype is based on Nightcore [66], where we add 13,133 lines of code, mostly in C++. Boki's support libraries are implemented in Go, consisting of 3,569 lines of code. One of the support libraries, BokiFlow, derives from the Beldi codebase [30]. The LogBook API makes Beldi's techniques easier to implement, so that BokiFlow shrinks the Beldi library from 1,823 lines to 1,137 lines, or a 38% reduction.

Boki uses 64-bit integers as the tag type for LogBook records. In Boki support libraries, when we need other types (e.g., strings) as the log tag, we use their hash values instead and store the original string in the record data. Boki employs Dynamo [95]'s variant of consistent hashing (strategy 3 in their paper) to uniformly map between LogBooks and physical logs.

4.5.1 Storage backend

Boki provides two different options as its storage backend for log records.

The first option is to use a third-party key-value store library. LogBook records are stored with their unique sequence numbers (seqnum) as keys, and log data and other metadata are serialized as values. Current implementation supports RocksDB [54] and Tkrzw [63]'s TreeDBM. RocksDB is a key-value store based on log-structured merge-tree (LSM), and Tkrzw is B-tree-based.

The second options is that Boki implements an on-disk journal for storing log records. We refer to this storage option as JournalStore in our evaluation (§ 4.6.1). Boki's journal is implemented by append-only files. Every I/O thread has its own journal file. Boki uses a size limit for individual journal files, and I/O threads will create new journal files when current ones reach the size limit. To allow reading log records by their seqnums, Boki maintains a separate hash table to locate log records within journal files. To facilitate log trims, for each journal file, Boki maintains a bitmap indicating trimmed log records. Boki gradually reads trim commands stored in the metalog, and masks bitmaps of relevant journal files. Boki removes a journal file when its bitmap is fully masked.

When using Boki's own on-disk journal as log storage, new log records are flushed to journal files before storage nodes report progress to sequencers (② in Figure 4.2). In contrast, when using third-party key-value store libraries as storage backend, log records will be flushed to the key-value store after the metalog is propagated (④ in Figure 4.2). Therefore, using Boki's on-disk journal achieves a stronger durability guarantee than using key-value store libraries. Boki also provides the option to use an on-disk journal along with key-value store, which can combine the benefits of stronger durability with the flexibility of using third-party key-value store libraries. We compare these different options in § 4.6.1 (also see Table 4.3).

In our previous presentation of Boki [108], only the first option (using key-value store library) is described and evaluated. Boki's on-disk journal is a new storage option added in this work.

4.6 Evaluation

In this section, we first evaluate Boki with microbenchmarks to explore its performance characteristics (§ 4.6.1). We then evaluate Boki's support libraries using realistic workloads (§ 4.6.2, § 4.6.3, and § 4.6.4). Finally, we analyze how Boki's techniques benefit its use cases (§ 4.6.5).

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

Unless otherwise noted, the following Boki settings are fixed in our experiments: (1) the ZooKeeper cluster in the control plane has 3 nodes; (2) the replication factors of both physical logs (n_{data}) and metalogs (n_{meta}) equal 3; (3) one single physical log configured for all LogBooks; (4) for each physical log, there are 4 LogBook engines that store its index (though functions can read their LogBooks via remote engines); (5) the record cache per LogBook engine is 1GB (for both record data and auxiliary data § 4.2).

4.6.1 Microbenchmarks

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

- *What is the append throughput of a single LogBook?* We use an append-only workload to measure the throughput, and how the throughput scales with more resources. In this workload, each function is a loop of appending 1KB log records. Boki is configured to use JournalStore (the second option mentioned in § 4.5) as the storage backend. Results are shown in Table 4.2a. From the table, we

	Concurrent functions / Storage (S) nodes			
	320/4S	640/8S	1280/16S	2560/32S
$n_{\text{meta}} = 3$	156.1	314.5	654.8	1142.7
$n_{\text{meta}} = 5$	155.9	323.1	639.4	1153.8

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

	1 PhyLog	**2 PhyLogs**	**4 PhyLogs**
100 LogBooks	161.8	324.5	696.9
100K LogBooks	162.8	310.3	665.9

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

Table 4.2: Boki's throughput in append-only microbenchmark. Boki is configured to use JournalStore backend for storing LogBook records (§ 4.6.1).

see that when Boki is configured with 64 nodes, the append throughput scales to 1.2M Ops/s under 2,560 concurrent appending functions. At this point, the median latency is 1.94ms, and the p99 tail latency is 6.33ms. We also increase the replication factor of metalogs (n_{meta}) to 5, that provides higher durability for a metalog but potentially affects the metalog's append latency. However, it demonstrates similar LogBook throughput and scalability as $n_{\text{meta}} = 3$.

- *Can Boki efficiently virtualize LogBooks?* We use the same append-only workload, but log appends are uniformly distributed over many LogBooks. We use 1, 2, and 4 physical logs to virtualize 100 and 100K LogBooks. Boki is configured with 4 function and 4 storage nodes when using one physical log, and resources

	Throughput	**Latency (ms)**	
	(KOp/s)	*median*	*99% tail*
RocksDB (LSM)	764.2	1.39	53.2
Tkrzw (B-tree)	651.1	1.58	25.7
Stronger durability with on-disk journal			
JournalStore	654.8	1.66	5.42
RocksDB with journal	655.4	1.47	79.3
Tkrzw with journal	448.6	1.98	45.6

Table 4.3: Comparison of Boki's different storage backends, using append-only microbenchmark (§ 4.6.1). Using Boki's on-disk journal achieves stronger durability, though results in slightly lower throughput. Also note JournalStore achieves the lowest tail latency among all options.

are added linearly with more physical logs. Table 4.2b shows the results. From the table, we can see Boki is capable of virtualizing LogBooks with high density.

- *How do Boki's log storage options compare to each other?* As described in § 4.5.1, Boki supports multiple options for storing log records. We use the append-only workload to compare throughput and latencies of different options. In the evaluation, Boki is configured with 16 storage nodes, and we use 1,280 concurrent appending functions. Table 4.3 shows the result. From the result, we see using Boki's on-disk journal can achieve a stronger durability guarantee but with the cost of lower append throughput and higher latency. Notably, Boki's JournalStore can achieve very low 99% tail latency (5.42ms) compared with other options.

- *How fast can Boki functions read LogBook records?* We use an append-and-read workload to measure read latencies, where each function loops a procedure that first appends a log record, then reads the appended record 4 times. We configure Boki with 8 function and 8 storage nodes. Table 4.4 shows the results when using

	Local LogBook (LB) engine		**Remote**
	cache hit	*cache miss*	**LB engine**
median	0.09ms	0.29ms	0.43ms
99% tail	0.40ms	0.75ms	0.99ms

Table 4.4: Boki's read latencies under different scenarios (§ 4.6.1).

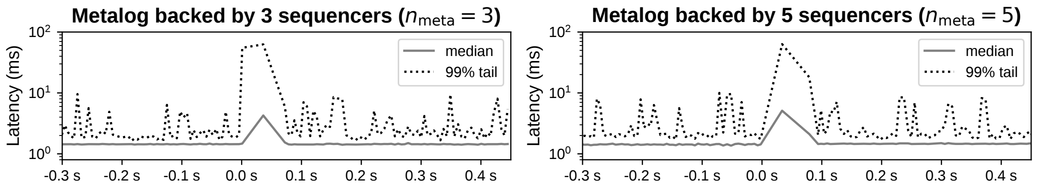
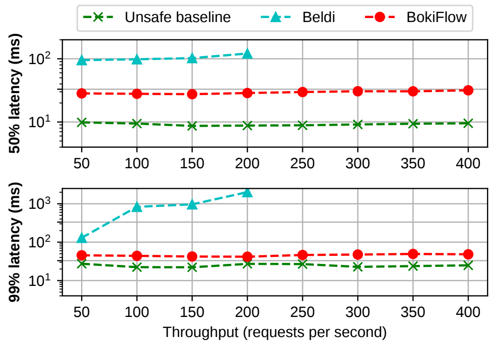


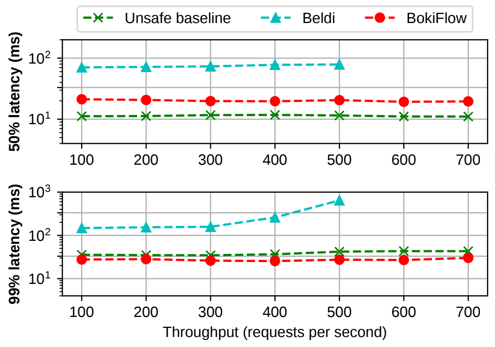
Figure 4.11: Log append latencies during reconfiguration (§ 4.6.1). The x-axis shows the timeline (in seconds). The reconfiguration starts at $t = 0$.

JournalStore for log storage. For other storage options, we observe similar latency numbers. For *remote engine* case, we enforce Boki to use remote LogBook engines for log reads. Cache hits take 86μs and never leave the local LogBook engine, retrieving the result from the record cache (§ 4.3.4).

- *What is the impact of reconfiguration?* We use the append-only workload to evaluate the impact of reconfiguration. In the experiment, Boki is reconfigured to a new set of sequencer nodes. New sequencer nodes are provisioned before the reconfiguration, to factor out provisioning delays from the experiment. Figure 4.11 shows the results. We see that Boki recovers to normal append latency after reconfiguration within 100ms. The actual reconfiguration protocol, executed by the controller, takes 15.7ms and 18.1ms, in experiments of $n_{\text{meta}} = 3$ and $n_{\text{meta}} = 5$, respectively.



(a) Movie review workload.



(b) Travel reservation workload.

Figure 4.12: Comparison of BokiFlow with Beldi [160]. BokiFlow takes advantage of the LogBook API. "Unsafe baseline" refers to running workflows without Beldi's techniques, where it cannot guarantee exactly-once semantics or support transactions (§ 4.6.2).

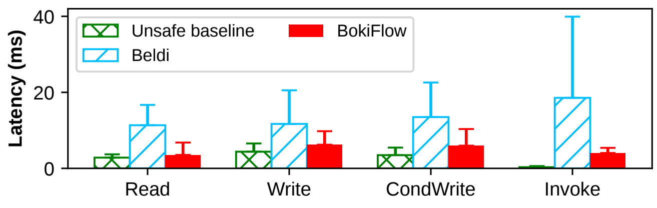


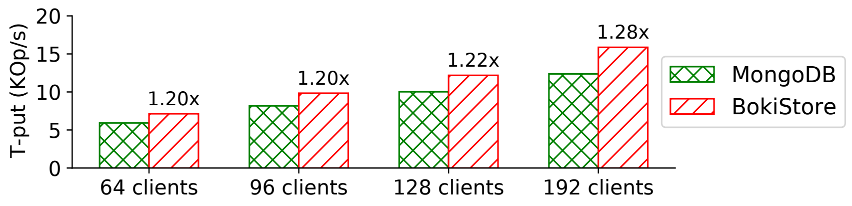
Figure 4.13: Microbenchmarks of Beldi primitive operations (§ 4.6.2). Main bars show median latencies, while error bars show 99% latencies.

4.6.2 BokiFlow: fault-tolerant workflows

We evaluate BokiFlow by comparing it with Beldi [160]. We use Beldi's workflow workloads, which model movie reviews and travel reservations. Both of them are adapted from DeathStarBench [27, 100] microservices. For a fair comparison, we port Beldi and its workloads to Nightcore, the underlying FaaS runtime of Boki. Both BokiFlow and Beldi store user data in DynamoDB [6]. BokiFlow stores workflow logs in a LogBook, while Beldi uses its linked DAAL technique to store logs in DynamoDB. For both systems, they are configured with 8 function nodes and Boki is configured with 3 storage nodes. Boki uses JournalStore as the storage backend.

Figure 4.12 shows the results. In both workloads, BokiFlow achieves much lower latencies than Beldi for all throughput values. In the movie workload, when running at 200 requests per second (RPS), BokiFlow's median latency is 28.7ms, $4.2\times$ lower than Beldi (121ms). In the travel workload, BokiFlow's median latency is 20.5ms at 500 RPS, $3.8\times$ lower than Beldi (78ms). In this experiment, we also run a baseline without Beldi's techniques, where it cannot guarantee exactly-once semantics or support transactions for workflows. When comparing BokiFlow with this baseline, we see that exactly-once semantics and transactions increase median latency by $3.3\times$ in the movie workload, and by $1.8\times$ in the travel workload.

We then run the microbenchmark that evaluates Beldi's primitive operations (Figure 13 in the Beldi paper [160]). Results are shown in Figure 4.13. The *Invoke* operation shows the largest differences among the three implementations and *Invoke* operations are very frequent in microservice-based workflows. In the baseline without workflow logs, the *Invoke* operation is very fast (well below 1ms). The underlying FaaS runtime, Nightcore, is heavily optimized to reduce invocation latencies. In BokiFlow, the *Invoke* operation needs needs 5 LogBook appends, thus it has a median latency of 4.0ms. Two of the five log appends are demonstrated in Figure 4.6 and the other three appends are made within the child function. For comparison, *Invoke* operation in Beldi also need 5 log appends, but has a median latency of 19ms, because of multiple DynamoDB updates for each log append.



(a) Throughput of BokiStore compared with MongoDB.

Request types	50% latency		99% latency	
	Mongo	Boki	Mongo	Boki
`UserLogin` (non-txn read)	**0.86**	1.41	**3.32**	5.47
`UserProfile` (non-txn read)	**0.86**	0.99	**3.57**	4.93
`GetTimeline` (read-only txn)	7.57	**3.24**	25.01	**10.09**
`NewTweet` (read-write txn)	7.72	**5.42**	21.39	**10.56**

(b) Latencies (in ms) under 192 clients. Although non-transactional reads in BokiStore are slower than MongoDB, transactions in BokiStore are up to 2.5× faster. Best performing result is in bold.

Figure 4.14: Evaluating BokiStore on Retwis workload (§ 4.6.3). Boki uses Journal-Store as storage backend.

These results justify the value of shared logs for the serverless environment, where building logging layers using a cloud database is difficult to make performant.

4.6.3 BokiStore: durable object storage

Retwis workload. To evaluate BokiStore, we build a transaction workload inspired by Retwis, a simplified Twitter clone [64]. The Retwis workload has been used as a transaction benchmark in previous work [161, 162]. We re-implement the Retwis workload in Go, requiring 1,458 lines of code. Our implementation uses BokiStore objects to store users, tweets, and timelines. For comparison, we also implement a version that uses MongoDB [61] to store objects, because MongoDB also employs a JSON-derived data model.

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

In the experiment, we configure Boki with 8 function nodes and 3 storage nodes using JournalStore. MongoDB is configured with 3 replicas. To ensure snapshot isolation in MongoDB transactions, we use a write concern of "majority" [74] and a read concern of "snapshot" [51]. For BokiStore, we configure LogBook engines on all 8 function nodes to have log index for the target LogBook, which achieves best data locality. We analyze the performance impact of using remote LogBook engines in § 4.6.5.

Figure 4.14 shows the results. From the figure, we see BokiStore achieves 1.20–1.28× higher throughput than MongoDB. When breaking down latency details by request types, we see BokiStore has considerable advantages over MongoDB in transactions (up to 2.5× faster). On the other hand, BokiStore is slower than MongoDB for non-transactional reads. This is caused by the log-structure nature of BokiStore, where log replay incurs overheads for data reads.

Comparison with Cloudburst. Cloudburst [146] is a recently proposed stateful FaaS runtime, which exports a put/get interface (i.e., key-value store) for functions to store state. BokiStore can also be used as a key-value store, by using keys as

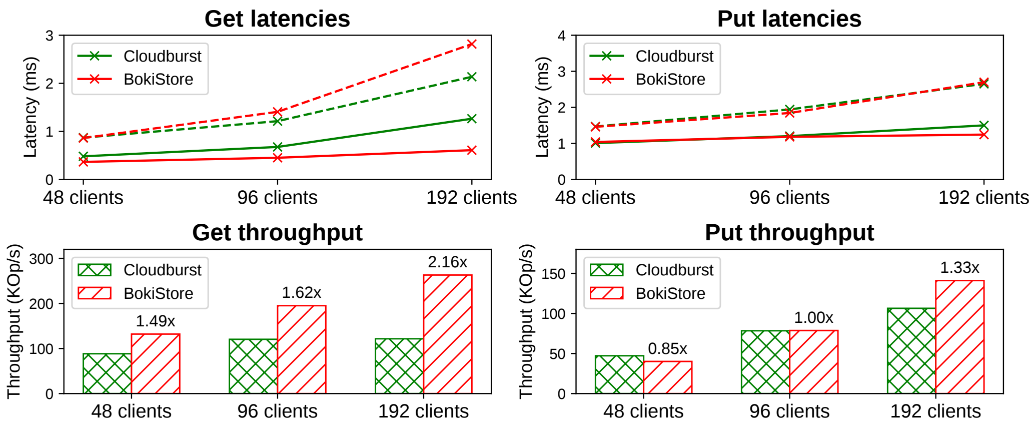


Figure 4.15: Comparison of BokiStore with Cloudburst [146]. We measure the latencies and throughput for put and get operations, using different numbers of concurrent clients. In the latency charts, solid lines show median latencies, and dashed lines show 99% tail latencies. BokiStore not only provides stronger consistency guarantees, but also achieves higher performance than Cloudburst (§ 4.6.3).

object names and storing values in the corresponding BokiStore object. However, BokiStore provides stronger consistency guarantees (sequential) than Cloudburst (causal). BokiStore also supports transactions reading and modifying multiple keys, which are not supported by Cloudburst.

We use a microbenchmark to compare Cloudburst's performance with BokiStore. Both systems use 8 storage nodes and 8 function nodes in the experiment. Figure 4.15 shows the result. BokiStore can achieve up to 2.16× higher throughput than Cloudburst on get operations. For put operations, BokiStore achieves 1.33× higher throughput when the concurrency is high. BokiStore provides higher throughput and lower median latency at 192 clients than Cloudburst, but it does have higher tail latency.

4.6.4 BokiQueue: message queues

We evaluate BokiQueue by comparing it with Amazon Simple Queue Service (SQS) [8] and Apache Pulsar [10]. Amazon SQS is a fully managed message queue service from AWS, while Pulsar is a popular open source distributed message queue. Similar to BokiQueue, both SQS and Pulsar use sharding to improve the data throughput of their message queues. In the experiment, we configure Boki with 8 function nodes and 3 storage nodes. Boki is configured with Tkrzw as storage backend for best performance. For Pulsar, we run its broker services on function nodes for better locality, and use the 3 storage nodes for queue data.

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

Table 4.5 shows the results. When the P:C ratio is 1:4, the queue is lightly loaded. We see both BokiQueue and Pulsar achieve double the throughput of Amazon SQS. BokiQueue achieves up to 1.6× lower latencies than Pulsar. When the P:C ratio is 4:1, the queue is saturated. Amazon SQS suffers significant queueing delays, limiting its throughput. BokiQueue and Pulsar have very similar throughput, while BokiQueue achieves 1.36× lower latency than Pulsar in the case of 256 producers. Finally, when the P:C ratio is 1:1, the queue is balanced. BokiQueue consistently achieves higher throughput and lower latency than both Amazon SQS and Pulsar.

Producer/ Consumer	**Throughput**			**Delivery latency (ms)**		
	SQS	Pulsar	Boki	SQS	Pulsar	Boki
16P/64C	2.25	5.05	**5.21**	6.27 (52.5)	4.01 (12.3)	**2.97 (5.05)**
32P/128C	4.03	9.67	**10.4**	6.01 (51.3)	6.70 (12.8)	**3.18 (6.12)**
64P/256C	7.62	14.1	**15.5**	6.08 (56.5)	7.39 (13.7)	**4.67 (14.8)**
64P/16C	2.34	**8.71**	7.92	33.9 (228)	6.20 (12.7)	**5.10 (14.9)**
128P/32C	5.35	**14.6**	14.1	53.9 (370)	7.38 (14.0)	**5.48 (19.2)**
256P/64C	9.77	19.1	**21.1**	99.8 (764)	7.81 (33.7)	**5.75 (20.2)**
64P/64C	6.37	10.0	**10.5**	7.22 (76.0)	6.77 (12.9)	**3.15 (6.64)**
128P/128C	10.1	17.8	**21.0**	7.24 (79.6)	7.74 (21.4)	**3.81 (9.53)**
256P/256C	18.5	25.0	**31.5**	12.1 (84.5)	8.21 (39.5)	**5.64 (17.5)**

Table 4.5: Comparison of BokiQueue with Amazon SQS [8] and Pulsar [10] (§ 4.6.4). Boki is configured with Tkrzw as storage backend, which achieves best performance for BokiQueue. Throughput is measured in 10^3 message/s. Delivery latency is the duration that a message stays in the queue. Latencies are shown in the form of "median (99% tail)". For each row in the table, best performing result is in bold.

Combining these three cases, BokiQueue achieves 1.70–2.16× higher throughput than Amazon SQS, and up to 17× lower latency. Compared with Pulsar, BokiQueue achieves 1.10–1.26× higher throughput, and up to 1.6× lower latency.

4.6.5 Analysis

The importance of auxiliary data. We describe in § 4.4.4 the log replay optimization using LogBook's auxiliary data. We use Retwis workload to demonstrate its importance for BokiStore. We run an experiment that disables this optimization. Furthermore, to demonstrate the efficiency of Boki's storage mechanism for auxiliary data, we modify Boki to store auxiliary data in a dedicated Redis instance.

Table 4.6 shows the results. From the table, we see that the log replay

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

Table 4.6: The importance of log replay optimization using auxiliary data (§ 4.6.5). The table shows Retwis throughput (in Op/s).

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

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

To experiment with the impact from using remote LogBook engines we limit the ratio of log reads that are locally processed, with the remainder processed remotely. Table 4.7 shows the results. We see even under a poor locality of LogBook engines, the performance drop is moderate (e.g., 77% of maximum throughput at 25% local reads).

Read locality also comes from the record cache included in LogBook engines. The cache stores both record data and auxiliary data for LogBook records. We ex-

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

Table 4.7: Locality impact from LogBook engines (§ 4.6.5). The table shows Retwis throughput (in Op/s), when adjusting the percentage of reads processed by local LogBook engines.

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

Table 4.8: LogBook engines maintain local cache for log records, and the cache size has performance impact for Boki's applications (§ 4.6.5). The table shows Retwis throughput (in Op/s).

periment with different cache sizes to analyze its impact on BokiStore performance. Results are shown in Table 4.8. We observe a sharp dorp in throughput when the cache size is decreased to 16MB. The cause of this drop is insufficient cache storage for auxiliary data. Auxiliary data is important for BokiStore performance, and a small record cache decreases the effectiveness of the log replay optimization. We modify Boki to backup auxiliary data on storage nodes, so that under a cache miss, storage nodes can also return auxiliary data. With this mechanism, small cache sizes no longer cause a sharp dorp in performance.

Log index versus fixed sharding. In § 4.3.4, we motivate the log index design because it allows records from a LogBook to be placed in arbitrary log shards. An

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

Table 4.9: Append throughput (in KOp/s) when log appends are distributed over 128 LogBooks under a uniform or Zipf distribution.

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

Table 4.10: Scaling read-only transactions with LogBook engines (§ 4.6.5). The experiment runs Retwis workload under a fixed write rate.

alternative approach is fixed sharding used in previous systems such as vCorfu [154]. We use the append-only microbenchmark to demonstrate the advantage of Boki's approach. For comparison, we modify Boki to use a fixed sharding approach, where a hashing function maps each LogBook to a log shard. Results are shown in Table 4.9. When log appends are uniformly distributed over LogBooks, the two approaches show no difference. However, when the distribution is skewed, fixed sharding suffers from uneven loads between log shards, while Boki's log index approach is unaffected.

Scaling LogBook engines. We then demonstrate the scalability of LogBook engines, by running read-only transactions in the Retwis workload. The workload is a mixture of read-only transactions (GetTimeline) and read-write transactions (NewTweet). In the experiment, we add more function nodes to scale LogBook engines, while always using 3 storage nodes. Every LogBook engine maintains a log index for the target LogBook. We fix the rate of NewTweet to 700 requests per second. Results are shown in Table 4.10. The results demonstrate Boki can scale from 8 LogBook engines to 48, thereby providing 4.63× higher read throughput.

Garbage collection (GC). As discussed in § 4.4.5, Boki provides the `logTrim` API for its support libraries to reclaim space from old and useless log records. We demonstrate the effectiveness of Boki's GC mechanism in BokiStore and BokiQueue. For BokiStore, we run a workload where 96 concurrent functions modify 1,000 BokiStore objects. For BokiQueue, we run a workload with 200 producer and 200 consumer functions. Boki is configured to use JournalStore as the storage backend. Figure 4.16 shows the state of one storage node. From the figure, we can see GC effectively controls disk utilization, while not affecting write throughput.

For comparison, we also run the same BokiStore and BokiQueue workloads without garbage collection. In both workloads, we found GC has no influence on the throughput. However, enabling GC in BokiQueue can reduce the tail latency of message delivery from 29.5ms to 8.90ms. For the BokiStore workload, we observe no difference in request latencies.

To quantify CPU overhead, we compute the average CPU utilization over the time span shown in Figure 4.16. In BokiStore experiments with GC disabled, the average system and user utilization is 162% and 79% (241% total). For comparison, enabling GC increases CPU utilization by 13%: system and user utilization rise to 178% and 95% (273% total). In BokiQueue experiments with GC disabled, the

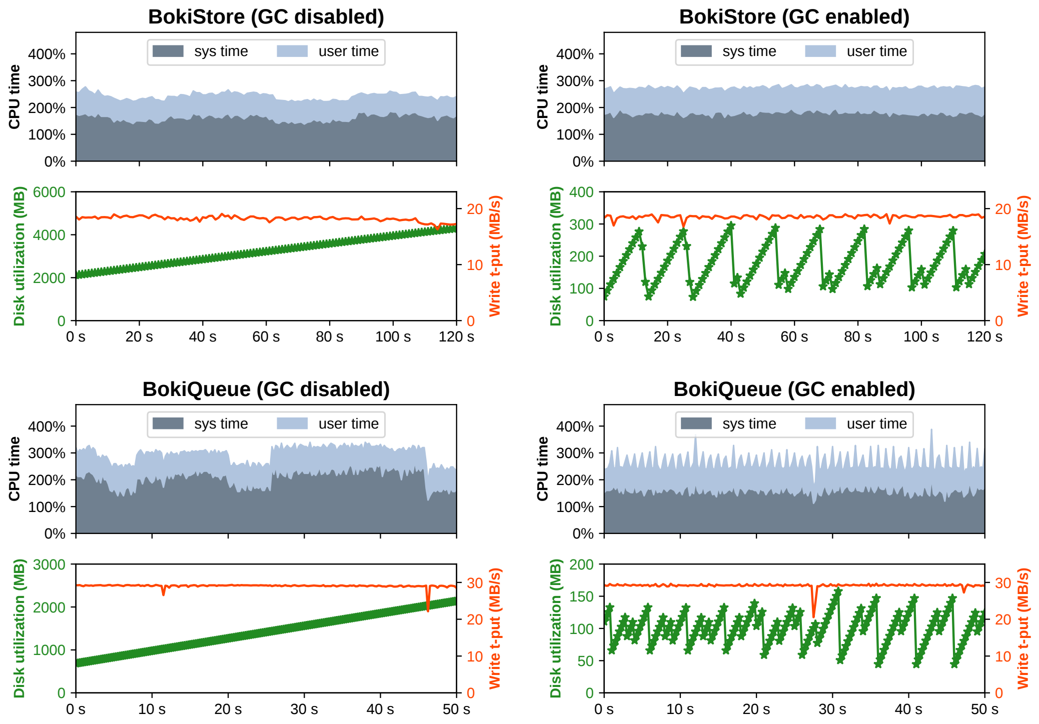


Figure 4.16: Demonstration of garbage collection (GC) in BokiStore and BokiQueue (§ 4.6.5). All figures show the state of one storage node: the upper chart shows CPU time; and the lower chart shows disk utilization and write throughput. One storage node has 4 CPU cores. For BokiStore experiments, we show a duration of 120 seconds in the middle of running. For BokiQueue experiments, we show a duration of 50 seconds.

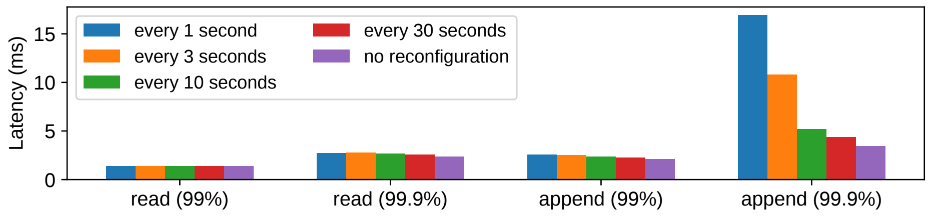


Figure 4.17: Sensitivity study of LogBook latencies to reconfiguration frequency (§ 4.6.5). Reconfigurations have little impact on log read latencies, but can significantly affect tail latencies of log appends when they are frequent. In all tested frequencies, throughput of log reads and appends is not affected (same as "no reconfiguration"). Data are collected over a 5-minute period.

average system and user utilization is 208% and 88% (296% total). Enabling GC decreases system utilization to 159% and increases user utilization to 105% (264% total). GC raises CPU utilization by 13% in BokiStore, but lowers it by 12% in BokiQueue.

Sensitivity study of reconfigurations. We finally study how reconfiguration frequency affects Boki's performance. In the experiment, Boki is configured with a single physical log using $n_{\text{meta}} = 3$. To allow reconfigurations without frequently allocating new nodes, we provision redundant nodes for Boki. In the experiment, 8 sequencer nodes are provisioned, while only 3 of them are active at one time because $n_{\text{meta}} = 3$. Reconfigurations are manually triggered periodically with a fixed frequency, from every 1 second to every 30 seconds. For each reconfiguration, 3 sequencer nodes are randomly chosen to store the metalog in the new term. We run a workload of log appends and reads (check tail), where the ratio between appends and reads is 1:4. 320 concurrent functions are executed over 8 function nodes. Results are shown in Figure 4.17. For read operations, we see that even frequent reconfigurations have little impact on their latencies. But for append operations, when reconfigurations become very frequent, their tail latencies increase significantly.

4.7 Summary

State management has become a major challenge in serverless computing. Boki is the first system that allows stateful serverless functions to manage state using distributed shared logs. Boki's shared log abstraction (i.e., LogBooks) can support diverse serverless use cases, including fault-tolerant workflows, durable object storage, and message queues. We build Boki support libraries for these use cases, allowing end applications to take advantages of LogBooks with minimal effort.

Boki shared logs achieve elasticity, data locality, and resource efficiency, enabled by a novel metalog design. The metalog orders shared log records with high throughput and it provides read consistency while allowing service providers to optimize the write and read path of the shared log in different ways. In Boki, the metalog provides a unified solution to the problems of log ordering, consistency, and fault tolerance.

To justify the value of shared logs in stateful serverless computing, we use realistic cloud workloads to evaluate Boki support libraries. Evaluation results suggest the shared-log-based approach for serverless state management can lead up to 4.3× performance advantages. Boki and its support libraries are open source on GitHub: `https://github.com/ut-osa/boki`.

Chapter 5

Related Work

5.1 Serverless computing

Serverless computing enables a new way of building cloud applications [12, 22, 56, 68], having the benefit of greatly reduced operational complexity. Serverless functions, or function as a service (FaaS), provide a simple programming model of *stateless* functions. It allow developers to upload simple functions to the cloud provider which are invoked on demand. The cloud provider manages the execution environment of serverless functions. While being simple and highly elastic, FaaS has empowered diverse applications including video processing [80, 99], data analytics [110, 138], machine learning [92, 145], distributed compilation [98], and transactional workflows [160].

Recent research on serverless computing has mostly focused on data intensive workloads [80, 92, 98, 99, 110, 138, 145], leading invocation latency overheads to be largely overlooked. SAND [77] features a local message bus as the fast path for chained function invocations. However, SAND only allows a single, local call at the end of a function, while Nightcore supports arbitrary calling patterns (e.g., Figure 3.1). Faasm [145]'s chained function calls have the same functionality as Nightcore's internal function calls, but they are executed within the same process, relying on WebAssembly for software-based fault isolation. One previous work [89] also notices that FaaS systems have to achieve microsecond-scale overheads for efficient support of microservices, but they demonstrate only a proof-of-concept FaaS runtime that relies on Rust's memory safety for isolation and lacks end-to-end evaluations on realistic microservices.

5.2 Microservices

The emergence of microservices for building large-scale cloud applications has prompted recent research on characterizing their workloads [100, 147, 150, 167], as well as studying their hardware-software implications [100, 147, 148]. Microservices have a higher communication-to-computation ratio than traditional workloads [100] and frequent microsecond-scale RPCs, so prior work has studied various software and hardware optimization opportunities for microsecond-scale RPCs, including transport layer protocols [113, 118], a taxonomy of threading models [148], heterogeneous NIC hardware [124], data transformations [136], and CPU memory controllers [149]. The programming model of serverless functions maps inter-service RPCs to internal function calls, allowing Nightcore to avoid inter-host networking and transparently eliminate RPC protocol overheads. X-Containers [144] is a recently proposed LibOS-based container runtime, that improves the efficiency of inter-service communications for mutually trusting microservices. For comparison, Nightcore still relies on the current container mechanism (provided by Docker), which does not require microservices to trust each other.

5.3 System supports for microsecond-scale I/Os

Prior work on achieving microsecond-scale I/O has been spread across various system components, ranging from optimizing the network stack [107, 113, 118]; designs for a dataplane OS [87, 112, 127, 134, 135, 137]; thread scheduling for microsecond-scale tasks [90, 112, 134, 139, 148]; and novel filesystems leveraging persistent memory [120, 126, 159]. Additionally, the efficiency of I/O is also affected by the user-facing programming model [103, 156] and the underlying mechanism for concurrency [119, 153]. A recent paper from Google [86] argues that current systems are not tuned for microsecond-scale events, as various OS building blocks have microsecond-scale overheads. Eliminating these overheads requires a tedious hunt for the "killer microseconds." Inspired by this work, the design of Nightcore eliminates many of these overheads, making it practical for a microsecond-scale serverless system.

5.4 Stateful serverless computing

State management remains a key challenge in the current serverless environment [104, 140]. To meet the increasing demand for stateful serverless, there are recent attempts from industry, e.g., Cloudflare's Durable Objects [72] and Azure's Entity Functions [28]. These systems are still in their early stages and have seen limited adoption.

There are also proposals from academia, e.g., Pocket [117], Cloudburst [146], Faasm [145], and Jiffy [115]. These projects have different focus, e.g., heterogeneous storage technology [117], lightweight isolation [145], and auto-scaling [146]. These systems all export put-get interfaces (i.e., a key-value store) for functions to manage state (Jiffy [115] also supports file and FIFO queue interfaces). Boki is the first to study a different interface for serverless state management, the shared log API. Boki's shared log approach is motivated by the fault-tolerance and consistency challenges encountered by stateful serverless applications, which the put-get interface cannot easily address.

A recent article [140] argues future serverless abstractions will be general-purpose, where cloud providers expose a few basic building blocks, e.g., cloud functions (FaaS) for computation and serverless storage for state management. The shared log and key-value store are both promising storage building blocks, which can work together to enable new serverless applications.

5.5 Distributed shared logs

Recent studies on distributed shared logs [82–85, 96, 125, 154] heavily inspire the design of Boki. A shared log is a powerful primitive for achieving strong data consistency in the presence of failures, because it can be used for state machine replication (SMR) [141], the canonical approach for building fault-tolerant services.

Boki leverages Scalog [96]'s high-throughput ordering protocol. Virtual consensus in Delos [82] inspires Boki's design of metalogs. Materialized streams in vCorfu [154] inspire the design of log tags in the LogBook API, and LogBook's virtualization. However, Boki's metalog design distinguishes it from these prior works. The logical decoupling provided by the metalog allows existing techniques to be adopted smoothly, while enabling new techniques, e.g., the log index for read efficiency. For applications, Tango [84]'s techniques enable serverless durable objects [72] backed by shared logs.

5.6 Fault-tolerant workflows

Orchestrating serverless functions as workflows is an important serverless paradigm, provided by all major cloud providers [18, 29, 73]. Workflows aim at providing *exactly-once* execution semantics, but stateful serverless functions (SSF) complicate this goal.

Beldi [160] proposes solutions for current serverless platforms. Beldi's mechanism is inspired by Olive [142]'s log-based fault tolerance protocol. In a Beldi workflow, during execution of SSF operations, the actions are logged. Beldi periodically re-executes SSFs that encounter failures. The operation log is used to prevent duplicated execution of operation, so that *at-most-once* execution semantics are guaranteed. On the other hand, re-execution for failed SSFs ensures *at-least-once* execution semantics.

Beldi's log-based fault-tolerant mechanism motivates Boki's shared log approach for stateful serverless computing. However, their techniques would need to be adapted for use with shared logs (§ 4.4.1), mostly because the workflow log is not co-located with user data in the same database.

Chapter 6

Conclusion

While serverless computing is becoming popular, the current serverless infrastructure faces challenges from emerging cloud applications. On the one hand, latency-sensitive workloads such as interactive microservices impose strict requirements for FaaS runtime overheads. On the other hand, stateful applications that require strong guarantees for their critical states demand storage APIs designed for data consistency and fault tolerance from the serverless ecosystem.

This dissertation propose two novel system designs, Nightcore and Boki, to address these new challenges. Nightcore is a FaaS runtime with microsecond-scale overheads, specifically optimizing for latency-sensitive, interactive microservices. Boki is a serverless runtime for stateful applications, by providing shared logs for functions. Boki shared logs allow stateful applications to coordinate critical state with consistency and fault tolerance.

A recent article [140] anticipates future serverless computing will be general-purpose, where the serverless infrastructure fully hides details of servers from cloud applications. Towards this goal, we not only have to continue to optimize current function-as-a-service systems for broader range of stateless applications, but also need to re-think how to provide state storage abstractions to attract more stateful applications.

We believe this dissertation makes one useful step towards shaping the future of serverless computing. No matter if serverless computing will be successful in near future, we can already observe the demands for low latencies, data consistency, and fault tolerance in today's cloud computing. As the economic impact of cloud computing is growing, we hope ideas behind this dissertation can inspire future cloud computing research.

Bibliography

[1] 4 Microservices Examples: Amazon, Netflix, Uber, and Etsy. [Accessed Jan, 2021].

[2] Accessing Amazon CloudWatch logs for AWS Lambda. [Accessed Dec, 2020].

[3] Addressing Cascading Failures. [Accessed Jan, 2021].

[4] Adopting Microservices at Netflix: Lessons for Architectural Design. [Accessed Jan, 2021].

[5] Airbnb's 10 Takeaways from Moving to Microservices. [Accessed Jan, 2021].

[6] Amazon DynamoDB | NoSQL Key-Value Database | Amazon Web Services. [Accessed Jan, 2021].

[7] Amazon ElastiCache- In-memory data store and cache. [Accessed Jan, 2021].

[8] Amazon SQS | Message Queuing Service | AWS. [Accessed Apr, 2021].

[9] Announcing WebSocket APIs in Amazon API Gateway. [Accessed Dec, 2020].

[10] Apache Pulsar. [Accessed Apr, 2021].

[11] Apache Thrift - Home. [Accessed Jan, 2021].

[12] Architecture: Scalable commerce workloads using microservices. [Accessed Jan, 2021].

[13] asyncio — Asynchronous I/O. [Accessed Jan, 2021].

[14] AWS Fargate - Run containers without having to manage servers or clusters. [Accessed Jan, 2021].

[15] AWS Lambda execution context - AWS Lambda. [Accessed Jan, 2021].

[16] AWS Lambda FAQs. [Accessed Jan, 2021].

[17] AWS Lambda – Serverless Compute - Amazon Web Servicesy. [Accessed Jan, 2021].

[18] AWS Step Functions. [Accessed Jan, 2021].

[19] Azure Functions Serverless Compute | Microsoft Azure. [Accessed Jan, 2021].

[20] BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more. [Accessed Jan, 2021].

[21] Best practices for working with AWS Lambda functions. [Accessed Dec, 2020].

[22] Building serverless microservices in Azure - sample architecture. [Accessed Jan, 2021].

[23] Cloud Functions | Google Cloud. [Accessed Jan, 2021].

[24] Cloud Object Storage | Store and Retrieve Data Anywhere | Amazon Simple Storage Service (S3). [Accessed Jan, 2021].

[25] CorfuDB. [Accessed Apr, 2021].

[26] Coursera Case Study. [Accessed Jan, 2021].

[27] delimitrou/DeathStarBench: Open-source benchmark suite for cloud microservices. [Accessed Jan, 2021].

[28] Durable entities - Azure Functions. [Accessed Jan, 2021].

[29] Durable Functions Overview - Azure | Microsoft Docs. [Accessed Apr, 2021].

[30] eniac/Beldi. [Accessed Apr, 2021].

[31] Enough with the microservices. [Accessed Jan, 2021].

[32] Event-based Concurrency (Advanced). [Accessed Jan, 2021].

[33] eventfd(2) - Linux manual page. [Accessed Jan, 2021].

[34] firecracker/network-performance.md at master · firecracker-microvm/firecracker. [Accessed Jan, 2021].

[35] giltene/wrk2: A constant throughput, correct latency recording variant of wrk. [Accessed Jan, 2021].

[36] Go, don't collect my garbage. [Accessed Jan, 2021].

[37] Go memory ballast: How I learnt to stop worrying and love the heap. [Accessed Jan, 2021].

[38] GoogleCloudPlatform/microservices-demo. [Accessed Jan, 2021].

[39] gRPC - A high-performance, open source universal RPC framework. [Accessed Jan, 2021].

[40] IPC settings | Docker run reference. [Accessed Jan, 2021].

[41] libuv | Cross-platform asynchronous I/O. [Accessed Jan, 2021].

[42] Lyft Case Study. [Accessed Jan, 2021].

[43] Manage your function app. [Accessed Jan, 2021].

[44] Microservice Trade-Offs. [Accessed Jan, 2021].

[45] Microservices - Wikipedia. [Accessed Jan, 2021].

[46] New for AWS Lambda – 1ms Billing Granularity Adds Cost Savings. [Accessed Apr, 2022].

[47] OpenFaaS | Serverless Functions, Made Simple. [Accessed Jan, 2021].

[48] Performance Under Load. [Accessed Jan, 2021].

[49] plugin - The Go Programming Language. [Accessed Jan, 2021].

[50] Provisioned Concurrency for Lambda Functions. [Accessed Jan, 2021].

[51] Read Concern "snapshot" — MongoDB Manual. [Accessed Apr, 2021].

[52] Remind Case Study. [Accessed Jan, 2021].

[53] Rewriting Uber Engineering: The Opportunities Microservices Provide. [Accessed Jan, 2021].

[54] RocksDB | A persistent key-value store | RocksDB. [Accessed Apr, 2021].

[55] Serverless and Microservices: a match made in heaven? [Accessed Dec, 2020].

[56] Serverless Microservices - Microservices on AWS. [Accessed Jan, 2021].

[57] Serverless Microservices reference architecture. [Accessed Dec, 2020].

[58] shm_overview(7) — Linux manual page. [Accessed Jan, 2021].

[59] Splitting Up a Codebase into Microservices and Artifacts. [Accessed Jan, 2021].

[60] "Stop Rate Limiting! Capacity Management Done Right" by Jon Moore. [Accessed Jan, 2021].

[61] The most popular database for modern apps | MongoDB. [Accessed Apr, 2021].

[62] Thoughts on (micro)services. [Accessed Jan, 2021].

[63] Tkrzw: a set of implementations of DBM. [Accessed Apr, 2021].

[64] Tutorial: Design and implementation of a simple Twitter clone using PHP and the Redis key-value store. [Accessed Apr, 2021].

[65] Uncovering the magic: How serverless platforms really work! [Accessed Jan, 2021].

[66] ut-osa/nightcore: Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. [Accessed Apr, 2021].

[67] Watchdog - OpenFaaS. [Accessed Jan, 2021].

[68] What are Microservices? | AWS. [Accessed Jan, 2021].

[69] What is a serverless microservice? | Serverless microservices explained. [Accessed Dec, 2020].

[70] Why so slow? - Binaris Blog. [Accessed Jan, 2021].

[71] Worker threads | Node.js v14.8.0 Documentation. [Accessed Jan, 2021].

[72] Workers Durable Objects Beta: A New Approach to Stateful Serverless. [Accessed Jan, 2021].

[73] Workflows | Google Cloud. [Accessed Apr, 2021].

[74] Write Concern — MongoDB Manual. [Accessed Apr, 2021].

[75] 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.

[76] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[77] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[78] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[79] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020.

[80] 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.

[81] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[82] 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.

[83] 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.

[84] 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.

[85] 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 28th Symposium on Operating Systems Principles*, SOSP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[86] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, March 2017.

[87] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[88] Philip A. Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1295–1309, New York, NY, USA, 2015. Association for Computing Machinery.

[89] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *Proceedings of the 2018 USENIX Confer-*

ence on Usenix Annual Technical Conference, USENIX ATC '18, page 645–650, USA, 2018. USENIX Association.

[90] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Lightweight preemptible functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 465–477. USENIX Association, July 2020.

[91] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: Skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[92] 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.

[93] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is believing: A client-centric specification of database isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, page 73–82, New York, NY, USA, 2017. Association for Computing Machinery.

[94] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[95] 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. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[96] 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.

[97] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[98] 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.

[99] 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.

[100] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[101] Pedro García-López, Aleksander Slominski, Simon Shillaker, Michael Behrendt, and Barnard Metzler. Serverless end game: Disaggregation enabling transparency. *arXiv preprint arXiv:2006.01251*, 2020.

[102] S. Guo, R. Dhamankar, and L. Stewart. Distributedlog: A high performance replicated log service. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1183–1194, 2017.

[103] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Proceedings of the*

10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, page 135–148, USA, 2012. USENIX Association.

[104] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[105] 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*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[106] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. Perfiso: Performance isolation for commercial latency-sensitive services. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 519–531, USA, 2018. USENIX Association.

[107] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. Mtcp: A highly scalable user-level tcp stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, page 489–502, USA, 2014. USENIX Association.

[108] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery.

[109] 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 2021, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery.

[110] 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.

[111] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[112] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *Proceedings of the 16th USENIX Conference on*

Networked Systems Design and Implementation, NSDI'19, page 345–359, USA, 2019. USENIX Association.

[113] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter rpcs can be general and fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 1–16, USA, 2019. USENIX Association.

[114] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 654–663, New York, NY, USA, 1997. Association for Computing Machinery.

[115] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. Jiffy: Elastic far-memory for stateful serverless analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 697–713, New York, NY, USA, 2022. Association for Computing Machinery.

[116] Martin Kleppmann and Jay Kreps. Kafka, samza and the unix philosophy of distributed data. *IEEE Data Eng. Bull.*, 38(4):4–14, 2015.

[117] 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.

[118] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, Renton, WA, July 2019. USENIX Association.

[119] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, USA, 2007. USENIX Association.

[120] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[121] Butler W. Lampson. Hints for computer system design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, page 33–48, New York, NY, USA, 1983. Association for Computing Machinery.

[122] N. Lazarev, N. Adit, S. Xiang, Z. Zhang, and C. Delimitrou. Dagger: Towards efficient rpcs in cloud microservices with near-memory reconfigurable nics. *IEEE Computer Architecture Letters*, 19(2):134–138, 2020.

[123] Collin Lee and John Ousterhout. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 149–154, New York, NY, USA, 2019. Association for Computing Machinery.

[124] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-efficient microservices on smartnic-accelerated servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, Renton, WA, July 2019. USENIX Association.

[125] 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.

[126] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, July 2017. USENIX Association.

[127] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM*

Symposium on Operating Systems Principles, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[128] Ben Maurer. Fail at scale: Reliability in the face of rapid change. *Queue*, 13(8):30–46, September 2015.

[129] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 453–468, Boston, MA, March 2017. USENIX Association.

[130] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[131] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 358–372, New York, NY, USA, 2013. Association for Computing Machinery.

[132] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[133] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[134] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 361–377, USA, 2019. USENIX Association.

[135] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[136] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus prime: Accelerating data transformation in servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1203–1216, New York, NY, USA, 2020. Association for Computing Machinery.

[137] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the*

26th Symposium on Operating Systems Principles, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[138] 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.

[139] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 145–160, USA, 2018. USENIX Association.

[140] 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, April 2021.

[141] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[142] 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.

[143] 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.

[144] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 121–135, New York, NY, USA, 2019. Association for Computing Machinery.

[145] 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.

[146] 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, July 2020.

[147] A. Sriraman and T. F. Wenisch. μsuite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*,

pages 1–12, 2018.

[148] Akshitha Sriraman and Thomas F. Wenisch. μtune: Auto-tuned threading for OLDI microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, Carlsbad, CA, October 2018. USENIX Association.

[149] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis. The nebula rpc-optimized architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 199–212, 2020.

[150] T. Ueda, T. Nakaike, and M. Ohara. Workload characterization for microservices. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

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

[152] 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 *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1041–1052, New York, NY, USA, 2017. Association for Computing Machinery.

[153] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 268–281, New York, NY, USA, 2003. Association for Computing Machinery.

[154] 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.

[155] Matt Welsh and David Culler. Adaptive overload control for busy internet servers. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, page 4, USA, 2003. USENIX Association.

[156] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, October 2001.

[157] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 50–61, New York, NY, USA, 2011. Association for Computing Machinery.

[158] Mingyu Wu, Ziming Zhao, Yanfei Yang, Haoyu Li, Haibo Chen, Binyu Zang, Haibing Guan, Sanhong Li, Chuansheng Lu, and Tongbao Zhang. Platinum: A cpu-efficient concurrent garbage collector for tail-reduction of interactive services. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 159–172. USENIX Association, July 2020.

[159] 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.

[160] 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.

[161] Irene Zhang, Niel Lebeck, Pedro Fonseca, Brandon Holt, Raymond Cheng, Ariadna Norberg, Arvind Krishnamurthy, and Henry M. Levy. Diamond: Automating data management and storage for wide-area, reactive applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 723–738, Savannah, GA, November 2016. USENIX Association.

[162] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems*

Principles, SOSP '15, page 263–278, New York, NY, USA, 2015. Association for Computing Machinery.

[163] 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.

[164] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 379–391, New York, NY, USA, 2013. Association for Computing Machinery.

[165] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 456–468, 2016.

[166] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 149–161, New York, NY, USA, 2018. Association for Computing Machinery.

[167] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun

Zhao. Benchmarking microservice systems for software engineering research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 323–324, New York, NY, USA, 2018. Association for Computing Machinery.

[168] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: Os kernel support for a low-overhead container overlay network. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 331–344, USA, 2019. USENIX Association.

Vita

Zhipeng Jia was born and raised in an historic city — Nanjing, China. After completing his work at Nanjing Foreign Language School in 2013, he entered Tsinghua University, Beijing, for undergraduate study. At Tsinghua, he studied at the Institute for Interdisciplinary Information Sciences (IIIS), led by Turing Award laureate Andrew Chi-Chih Yao. He received the degree of Bachelor of Engineering in Computer Science and Technology from Tsinghua University in 2017. In August 2017, he entered the doctoral program in the Department of Computer Science at the University of Texas at Austin, where he received an M.S. degree in Computer Science in December 2021.

Permanent address: zhipeng.jia@outlook.com

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.