Sequoia: Enabling Quality-of-Service in Serverless Computing

Ali Tariq
University of Colorado Boulder
Boulder, Colorado
ali.tariq@colorado.edu

Austin Pahl
University of Colorado Boulder
Boulder, Colorado
austin.pahl@colorado.edu

Sharat Nimmagadda
University of Colorado Boulder
Boulder, Colorado
sharat.nimmagadda@colorado.edu

Eric Rozner
University of Colorado Boulder
Boulder, Colorado
eric.rozner@colorado.edu

Siddharth Lanka
University of Colorado Boulder
Boulder, Colorado
sai.lanka@colorado.edu

ABSTRACT

Serverless computing is a rapidly growing paradigm that easily harnesses the power of the cloud. With serverless computing, developers simply provide an event-driven function to cloud providers, and the provider seamlessly scales function invocations to meet demands as event-triggers occur. As current and future serverless offerings support a wide variety of serverless applications, effective techniques to manage serverless workloads becomes an important issue. This work examines current management and scheduling practices in cloud providers, uncovering many issues including inflated application run times, function drops, inefficient allocations, and other undocumented and unexpected behavior. To fix these issues, a new quality-of-service function scheduling and allocation framework, called Sequoia, is designed. Sequoia allows developers or administrators to easily define how serverless functions and applications should be deployed, capped, prioritized, or altered based on easily configured, flexible policies. Results with controlled and realistic workloads show Sequoia seamlessly adapts to policies, eliminates mid-chain drops, reduces queuing times by up to 6.4×, enforces tight chain-level fairness, and improves run-time performance up to 25×.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; *n-tier architectures.*

KEYWORDS

Serverless Computing, Quality-of-Service, Measurement

ACM Reference Format:
Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling Quality-of-Service in Serverless Computing. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3419111.3421306

1 INTRODUCTION

In serverless computing, also referred to as Functions-as-a-Service (FaaS), application developers provide an event-driven function to cloud providers, and the cloud provider is responsible for seamlessly scaling function invocations to meet demands as event triggers occur. Serverless is powerful and expressive, with applications designed for video processing [29, 41], HPC and scientific computing [36, 51, 89, 93], machine learning [35, 39, 50], data analytics [44, 55], chatbots [103], backends [31, 67], IoT [69, 102], and even general applications [40, 92]. Indeed, a recent study of a production serverless offering indicates applications range from single functions to hundreds of functions in size, with function execution times ranging from less than a second to the order of minutes [88]. Therefore, the future promises a fast-growing serverless-native ecosystem [71], in which diverse *serverless function chains*, where serverless functions call subsequent serverless functions to create compositions, must be supported over a common infrastructure.

As serverless function chains become more common, complex, and relied upon, tools must be provided to ensure administrators and serverless developers can effectively manage these new workloads. Better manageability will more

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SoCC '20, October 19–21, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421306

easily enable serverless applications to achieve service-level agreements (SLAs) by ensuring predictable and efficient cloud performance and hence maximizing revenue [27, 37, 60, 95]. Beyond SLAs, management is important to developers or administrators for a variety of reasons. For example, managing where functions or chains can run (*e.g.,* public or private cloud) is important for privacy and regulatory reasons. Management ensures how multiple applications, or functions within applications, can consume resources, ensuring important workloads or functions are prioritized when needed. Additionally, controlling consumption simplifies budgeting operational expenditures.

As shown in this paper, the current state of serverless function chain management leaves much to be desired. Policies to manage serverless functions and function chains are relatively simple: scheduling policies today typically implement basic first-come-first-served algorithms. When limits are imposed on serverless workloads running in parallel, either from hard concurrency limits enforced by the provider or soft concurrency limits observed due to inefficient resource allocation, this leaves little flexibility to dictate how serverless applications should be managed under challenging conditions. Our measurements (Section 2) show current management practices can lead to a variety of issues with serverless performance, including inconsistent and incorrect limitations, inefficient resource allocation, inflated run times, mid-chain function drops, concurrency collapse, and undocumented function prioritization.

To help alleviate these problems, as well as provide a more mature deployment ecosystem, we introduce a Quality-of-Service (QoS) scheduler for serverless functions and chains. Our framework, called Sequoia, allows policies to dictate how or where function chains, or functions within chains, should be prioritized, scheduled, or queued. Our QoS scheduler is implemented as a drop-in frontend so its performance can be analyzed across five different commercial and open-source serverless offerings. Sequoia's design enables flexible policies to be easily defined and realized without changes to the serverless functions themselves. We show how management policies can avoid performance issues and enable rich scheduling techniques such as seamlessly scheduling over a hybrid private-public cloud or managing performance at a chain (*i.e.,* application) level. In short, we aim to make QoS a first-class citizen in serverless deployments. The contributions of our work are as follows:

- A measurement study showing the current state of QoS scheduling for serverless function chains over five major providers. The measurements show how current techniques can adversely affect function chain performance, leading to drops, inflated completion times, and unexpected behavior.
- A new drop-in QoS function chain scheduler that alleviates problems uncovered in the measurement study. The scheduler can accurately realize a variety of flexible policies to make serverless function chain management more effective. Our code is published at: https://github.com/CU-BISON-LAB/sequoia.
- Evaluation of controlled and realistic workloads showing Sequoia eliminates mid-chain drops, reduces queuing times by up to 6.4×, enforces tight chain-level fairness, and improves run-time performance up to 25×.

2 BACKGROUND

This section first details how function chains are supported across serverless platforms and then presents a QoS-related measurement study.

Serverless Function Chains Serverless function chains, consisting of one or more serverless functions, can be realized via three main invocation mechanisms. First, with *synchronous* function calls, developers call a serverless function from within the current function directly. Examples include an HTTP request or an output from a load balancer. Second, in *asynchronous* function calls, a serverless function will output some event, which then triggers a subsequent function call. Examples include adding elements to a storage service or using a pub/sub system. Last, a special case of synchronous chains exists with *composition* frameworks, such as AWS Step Functions [2], Azure Durable Functions [6], or IBM Composer [14]. In composition frameworks developers specify a call graph, and the provider ensures functions are called accordingly.

2.1 QoS in Serverless Offerings

While many FaaS offerings exist [3, 4, 10, 12, 15, 17, 18, 20], relatively basic techniques to manage serverless function invocations are provided today. As function requests are received, the cloud provider schedules functions, mostly in a first-in-first-out manner. Opportunities to invoke a more informed scheduling policy are missed, however, in challenging scenarios such as when incoming demands cannot be satisfied by currently available resources. Such scenarios occur when providers cannot accommodate a rise of invocations due to cold starts or inefficient resource allocation or alternatively when *function invocation limits* are imposed. Function invocation limits bound the number of functions running either instantaneously or over a time period. Because serverless technologies automatically scale to meet demands, function invocation limits ensure a bug or misconfiguration in tenant workloads does not inappropriately scale. In addition, limits help developers manage costs and better understand expected workload characteristics. Changing limits requires out-of-band approval from support centers [5, 7]. To better understand issues with serverless QoS, five major serverless providers are detailed below.

AWS Lambda AWS Lambda provides users with a total concurrency threshold shared by all serverless functions. Individual serverless functions can further be configured to use a dedicated concurrency share which is deducted from the total concurrency threshold. For synchronous traffic, AWS Lambda does not provide any queuing mechanism and therefore any demand or invocations above the concurrency limit gets dropped or returns with an error. For asynchronous workloads, AWS Lambda can queue when concurrency limits are exceeded, running queued functions when current concurrency levels drop below the threshold. Every function is run in isolation (its own micro-VM [9, 65]), although the same VM can later be reused for another instance of the same function. Existing VMs are destroyed automatically after a timeout period of up to a few hours [30].

IBM Cloud Functions IBM Cloud Functions follows a total concurrency threshold model similar to AWS. According to official documentation [13], a 1,000 concurrency limit is enforced across all running functions. As with AWS, IBM enables queuing of asynchronous functions when concurrency limits cross the threshold.

Apache OpenWhisk Apache OpenWhisk is an open-source serverless platform very similar to IBM Cloud Functions as both share a similar design. OpenWhisk follows a total concurrency pool model.

Google Cloud Functions (GCF) GCF divides serverless functions into HTTP functions and background (*i.e.,* asynchronous) functions [11]. GCF enforces concurrency limits on individual functions as opposed to a total concurrency pool. For HTTP functions, there is no mentioned limit on the concurrency however, in practice we observe a varying concurrency limit between 1,000 & 2,000 (Section 2.2). All requests beyond this are queued and run in-turn. For background functions, a strict concurrency limit is enforced per function. Unlike the previous providers, GCF provides various configuration options for its users to limit resource usage, with limits available on total CPU or memory usage over all functions. GCF uses a 100 second interval for assessing and enforcing resource limits. GCF handles synchronous workloads in best effort fashion: it seems to perform queuing but doesn't ensure zero drops (Section 2.2). For asynchronous workloads, GCF provides queuing just as other platforms do.

Azure Function Apps Azure Function Apps group functions into "Function Apps" which automatically add VMs, or "instances," to match the current load on all of the functions within the app. A single function app may have up to 200 VMs allocated at once, and each VM can host multiple functions running in parallel based on the resource demand of each function [8]. Users have the option to configure various other quotas as well, such as HTTP function concurrency, outstanding function queue sizes, and timeouts for long-running functions. If the Function App does not have enough instances allocated to support a sudden burst of function invocations, we found some of the invocations will be enqueued or dropped.

2.2 Measurement Study

This section presents a measurement study to better understand the current state of QoS in today's serverless platforms. Our study consists of results from November 2019 to May 2020. Figure 1 shows the tested workloads. Functions in the workload allocate 256 MB memory and sleep for 15 seconds. The workloads are as follows: *Single:* This is the simplest workload, consisting of individual independent requests. *Linear-N:* A serverless chain where every serverless function invokes up to one new serverless function. *Fan-N:* Another chain where multiple tasks depend on a previous function's completion. Single, Linear-N and Fan-N are important to study because they serve as building blocks for more complex applications. *Combo:* This chain includes combinations of Single, Linear-N, and Fan-N (unless otherwise stated, Combo refers to the chain in Figure 1). *MixedChain:* A workload in which the chains in Figure 1 are run simultaneously. Note the chains share similar functions, *e.g.,* λ_1 is run in all three chains and λ_3 is run twice in Combo.

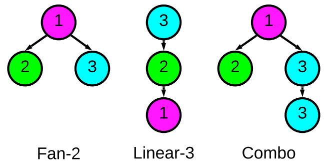
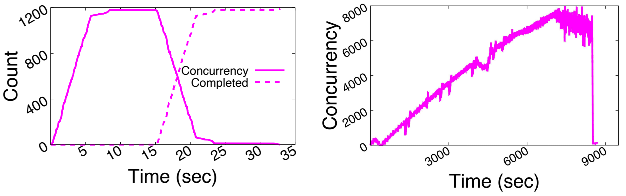


Figure 1: Example function chains in study

The above workloads are run under different demands. *Burst-N* sends a burst of N simultaneous requests at once. Some initial studies have shown burst workloads to be common in serverless applications [40, 52, 56]. *Continuous-N* sends constant N requests per second. An open-loop *Poisson* process, which has been extensively used in serverless evaluations [25, 63, 70, 87, 91] and approximates large-scale, web-driven workloads [86]. Cold start issues are mitigated by running all results multiple times in succession and verifying trends hold.

We conduct a series of experiments in which functions and chains are assigned identifiers and function start and end times are logged to enable reverse engineering of provider queuing policies. Although results are omitted due to space, we find (i) scheduling across frameworks follows a simple FIFO queuing model and (ii) scheduling is performed on a per-function basis (instead of other policies like per-chain).

2.2.1 Limitations. This section shows limitations in existing serverless offerings and how these impact QoS for incoming requests and overall performance. Specifically, it is shown that inconsistent and incorrect concurrency limits are prevalent, mid-chain function drops occur, workloads such as bursts are not easily supported, HTTP functions are prioritized without documentation, inefficient resource allocation is common, and concurrency collapses under certain conditions.



(a) IBM Cloud Functions **(b) Azure Functions**

Figure 2: Incorrect concurrency limits

Inconsistent and incorrect concurrency limits We find numerous issues with concurrency limits on serverless platforms. IBM suffers from a simple issue: default concurrency limits are documented to be 1,000, but up to 1,200 concurrent functions are run in parallel. Figure 2a shows a burst of 1,200 Single functions. The x-axis is time, the y-axis is number of concurrently running functions, the dotted line tracks completions, and the solid line shows up to 1,200 functions running simultaneously.

In the worst case, no enforcement can occur in Azure. A workload is created in which demand is slowly ramped up over time. Azure does not limit the number of concurrent HTTP functions, which was configured to 1,000, or the number of instances, which is 200 by default. During the test, the Function App's Live Metrics Stream reported up to 440 instances allocated to the Function App with up to 8,000 concurrent requests run at a time, as shown in Figure 2b.

Last, GCF does not limit total CPU consumption in a tight manner. GCF caps total CPU usage over all functions to a specified threshold over a 100 second period. CPU consumption is tracked during the period, and when the threshold is reached, no new functions are invoked. We find two issues, however. First, any outstanding functions are able to complete after the limit is reached, violating CPU limits. Second, a slow trickle of invocations still occurs after the CPU limit is reached. Figure 3 shows CPU usage is more than doubled in the MixedChain workload: CPU limits were set to 40M MHz/s, but over 90M MHz/s consumption was encountered (dotted red line). Concurrency for each λ and total concurrency, the sum of all λ concurrencies, are also shown.

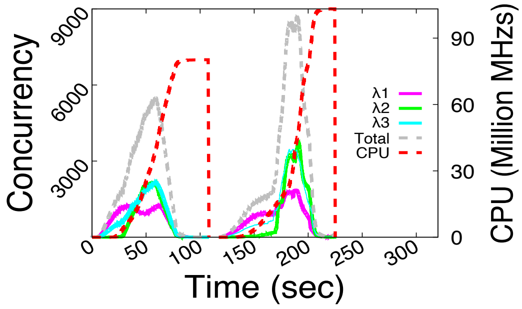


Figure 3: GCF: MixedChain workload CPU usage

The above findings indicate concurrency limits are often inconsistent or incorrect, placing additional burden on serverless developers. When limits are under intended values, workloads may unexpectedly encounter poor performance or increased drops. Dealing with such issues increases serverless application complexity. When limits are over intended values, developers may incur higher costs than budgeted for. And when limits are inconsistent, developers can have difficulty managing and reasoning about serverless performance.

Mid-chain drops Some serverless platforms provide a hard concurrency limit (AWS and IBM) beyond which all subsequent requests are dropped. When demand rises above a specified function invocation limit, functions can be queued (up to 4 days in the case of AWS [81]), silently dropped [82], or returned with an error (in the synchronous case only). This is problematic for several reasons. First, developers may rely on function chain completion, and when function chains drop mid-chain, incorrectness may arise. Alternatively, developers can solve the problem at the application layer, but this increases complexity and developer burden, two problems serverless aims to solve. Third, drops mid-chain result in inefficiency because the resources spent running functions before the drop are wasted and could have been better used to finish some other outstanding function chain. And last, if providers queue requests mid-chain, then the total function chain running time variance can be significantly increased, impacting SLAs or otherwise negatively affecting performance.

To assess the impact of mid-chain drops, a Fan-2 Burst workload is run on AWS Step Functions and IBM Cloud Functions, where the burst is the size of the concurrency limit. Note the "fan" portion of Fan-2 invokes twice as many functions after λ_1 completion, meaning a burst of 1,000 Fan-2's will ultimately result in 2,000 concurrent functions (*i.e.,* λ_2 & λ_3) and a violation of concurrency limits. Figure 4 shows a timeseries of the number of completed chains. The figure shows significant loss, with only 48-54% chains successfully completing.

Burst intolerance Bursts are difficult workloads to support because cold start issues and other inefficiencies with quickly scaling infrastructure can cause delays and even loss. GCF documentation indicates there is no concurrency limit on functions invoked via HTTP requests. In practice, however, concurrency ranges from approximately 1,000-2,000 when a large burst of 6,000 HTTP requests is invoked (not shown). Even if bursts are repeated over five iterations (to avoid cold starts in the subsequent four burst runs), there still exists inconsistent achievable concurrency, which results in 3%-65% HTTP requests returning with error codes, as shown in Figure 5a.

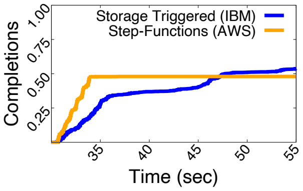
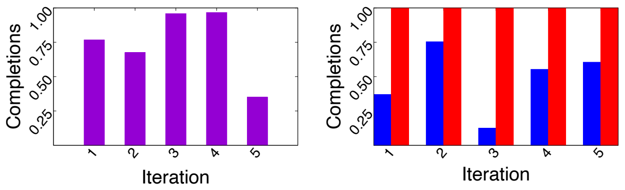


Figure 4: Mid-chain drops



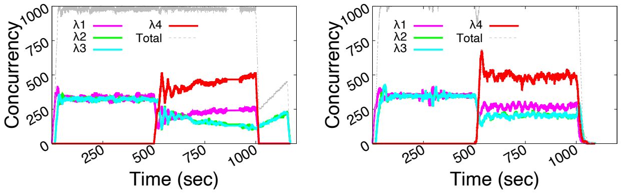
(a) GCF

(b) Azure Functions

Figure 5: Workload burst intolerance; in Figure 5b blue bars are cold starts and red bars are warm starts

Similar measurement results on Azure indicate burst workloads suffer from extraneous loss and low concurrency limits, as shown in Figure 5b. Significant losses occur when Burst-1000 workloads are repeated over five iterations, shown as cold starts in the blue bars. However, if invocation frequency gradually increases over time (instead of a sending a burst), then achievable concurrency can grow and loss can be eliminated, as shown in the warmed bar plots, colored red. Burst intolerance limits achievable concurrency, which in turn creates loss or queuing when demands spike.

HTTP prioritization As serverless providers often have a queue for HTTP invocations and a queue for background invocations, a simple test is designed to understand interference between HTTP and background invocations. First, a Continuous-23 Fan-2 workload is started (23 Fan-2 requests per second translates to a concurrency of 1035 functions in the steady state, just above the invocation limit). From $t = 500 - 1000$, a Continuous-46 Single workload (λ_4) is also run, increasing the total demand to overrun concurrency limits. Figure 6a shows function concurrency in AWS. Ideally, all functions within Fan-2 should have equal concurrency, as demonstrated in the first 500 seconds. Instead, when demands rise at $t = 500$ the first function in the chain, λ_1, which is invoked via HTTP, has larger concurrency than the background functions (λ_2 and λ_3). In addition, λ_4 (the Single workload, invoked by HTTP) and λ_1 see their concurrency *increasing* over time relative to λ_2 and λ_3. Hence, AWS prioritizes HTTP invocations over background invocations.



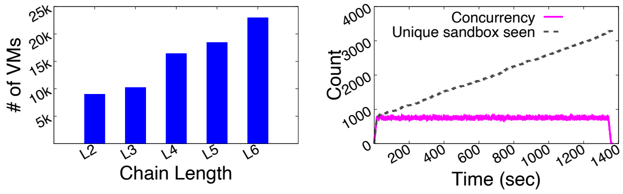
(a) AWS Lambda

(b) IBM Cloud Functions

Figure 6: HTTP prioritization

The HTTP prioritization problem also incurs on IBM. The same test is repeated and the results are shown in Figure 6b. Again, λ_1, which is invoked via HTTP, has larger concurrency than the background functions. Unlike AWS, the difference converges instead of increases over time. In both AWS and IBM, we ran different workloads and found the trends to hold. We were unable to find documentation detailing this prioritization, even though prioritization can significantly impact the behavior of serverless workloads.

Inefficient resource allocation For serverless platforms, using a much higher VM/container pool than the concurrency limit ensures faster scale-up and less cold-starts. This can, however, lead to inefficient resource allocation within the serverless platform. Using the methodology from [99], the number of unique VMs created over five iterations of a Burst-1000 workload are tracked for Linear-N, with N ranging from 2-6. The resulting graph in Figure 7a shows the total number of unique VMs used over all iterations for each chain. Instead of reusing existing in-memory VMs, inefficient VM usage is common. For example, in a perfect reuse scenario, Linear-2 would utilize a total of 2,000 VMs (1,000 for λ_1 and 1,000 for λ_2), but instead almost 10,000 VMs are used.



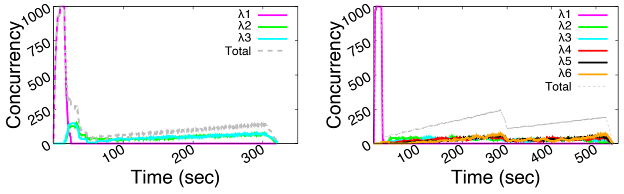
(a) AWS Lambda

(b) IBM Cloud Functions

Figure 7: Inefficient resource allocation

IBM also showed inefficient container usage, as shown in Figure 7b. A Continuous-67 Single function workload (which corresponds to the concurrency limit) sees the number of containers increase over time even though reuse is possible after a function completes. Note IBM reuse seemed inconsistent: a repeated Burst workload (a burst, followed by a timeout, followed by a burst) did indicate high levels of container reuse. Inefficient VM/container reuse can increase overheads such as cold start and also inefficiently utilize memory.

Concurrency collapse On AWS, a phenomena we denote *concurrency collapse*, where concurrency reaches the limit but then drops and does not immediately recover, can occur. A Burst-1000 workload for Fan-2 and Fan-5 are shown in Figure 8a and Figure 8b, respectively. The concurrency significantly drops after the λ_1's complete, but the demand from the invoked λ_2 and λ_3 should saturate concurrency. While we were unable to debug the reason for such collapse on AWS, we suspect the issue arises from inefficient resource allocation. We reproduced the problem in OpenWhisk by limiting the number of available containers to service functions. Here, λ_1's burst consumes all available containers, resulting in the collapse because no containers are readily available to service λ_2 and λ_3. Regardless, we show collapse can prevented in Section 5.



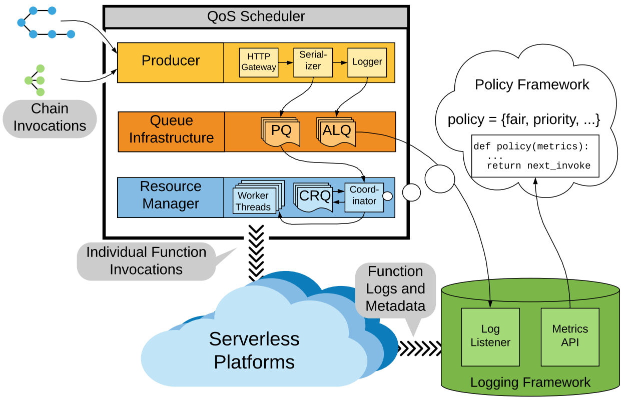
(a) Fan-2 **(b) Fan-5**

Figure 8: AWS Lambda concurrency collapse

Issue	AWS	IBM	GCF	Azure	OW
Concurrency limit issues	N	Y	Y	Y	Y
Inefficient allocation	Y	Y	N	N	N
Burst intolerance	N	N	Y	Y	N
Mid-chain drops	Y	Y	N	N	Y
Concurrency drops	Y	Y	Y	N	Y
HTTP prioritization	Y	Y	N	N	N

Table 1: Summary of issues discovered on providers

Summary Table 1 summarizes the issues discovered over each cloud provider. Note that due to space, graphs for all problematic instances cannot be shown. The findings in this section indicate QoS offerings in modern serverless ecosystems fall short and motivate the need for a new design. In the next section, we detail our QoS framework to better enable QoS.



```
def policy(metrics):
  ...
  return next_invoke
```

Figure 9: Sequoia architecture

3 ARCHITECTURE

This section details the architecture of Sequoia. Sequoia is a standalone scheduling framework that can be deployed as a proxy to existing cloud services or directly integrated into platforms such as OpenWhisk. The framework consists of three main logical entities (Figure 9): a *QoS Scheduler*, a *Logging Framework*, and a *Policy Framework*. The QoS Scheduler decides where, when, and how to run specific functions or function chains. The QoS Scheduler integrates tightly with the Logging Framework, whose responsibility is to store real-time metrics that describe the current and historical state of the serverless environment. Both the QoS Scheduler and Logging Framework interface with the Policy Framework to make scheduling decisions. All three components are highlighted below.

QoS Scheduler The QoS Scheduler, which is composed of a *Producer*, a *Queue Infrastructure*, and a *Resource Manager*, follows a producer-consumer model. The Producer is responsible for enqueuing new function chain invocations into the QoS Scheduler's Pending Queue (PQ), and measuring the volume of incoming traffic for the Policy Framework. In the Producer, function chains are defined as directed acyclic graphs (DAGs) where each node contains a serverless function to invoke and a unique ID to distinguish the node to the Policy Framework. Whenever the Producer receives a function chain invocation, it initializes a *ChainState* object for tracking the invocation with a unique invocation ID and a pointer into the function chain's DAG. These ChainState objects are enqueued into the PQ and read by the Resource Manager (described below) to track each chain's current position in the DAG.

The *Resource Manager* acts as the consumer in the infrastructure, pulling function requests from the queues to schedule functions on the server pool. *Worker Threads* in the thread pool are the main driver within the infrastructure and are responsible for dequeuing functions and scheduling them on the serverless infrastructure. When a thread schedules a function it blocks until its function completes and afterwards makes itself available to the thread pool. A simple example follows. Assume an initialized state with only one worker in the thread pool (concurrency limit = 1) and two just-arrived Fan-2 function chains in the PQ. The thread pulls one of the Fan-2 function chains from the PQ and runs λ_1 to completion. When λ_1 completes, the thread enqueues λ_2 and λ_3 into the *Chain Running Queue* (CRQ). The CRQ's job is to hold subsequent functions within the chain. The thread then exits and goes back into the thread pool. Next, the thread is re-activated and can schedule another function. In this case, λ_1 can be scheduled from the second enqueued function chain or λ_2 or λ_3 can be scheduled from the first chain. Such scheduling decisions are left to policy, so the Resource Manager integrates tightly with the Policy Framework.

Logging Framework In order to make effective scheduling decisions, a serverless function-chain management infrastructure needs to have a clear, live overview of the state of the serverless environment. The Logging Framework ingests state information from live metric streams such as serverless provider logs, function wrappers, and the QoS Scheduler itself. This gives the framework access to live and historical information such as function performance, error reporting, and details about the underlying containers and VMs hosting the functions [99], which it then provides to the Policy Framework in a platform-agnostic API. Note provider-based logging frameworks are generally not real-time and hence inadequate for many scheduling policies.

Policy Framework The Policy Framework engine integrates tightly with the QoS Scheduler and Logging Framework. The Policy Framework serves as an entry point to add, remove, or alter policies in the system. Individual threads in the Resource Manager can consult the Policy Framework when making scheduling decisions. The Policy Framework has access to rich state information from the Logging Framework, enabling many different kinds of policies to be expressed and applied regardless of the serverless platform in use.

4 IMPLEMENTATION

We implemented a proof-of-concept to show Sequoia's benefits. The implementation was written in Python, includes an instance of Kafka, and uses simple serverless function wrappers that send information to the Logging Framework. The design is flexible and modular, and hence other packages and services can be used for implementation. A description of each architectural component follows.

QoS Scheduler As mentioned, the QoS scheduler consists of a Producer and a Resource Manager. The Producer converts input function chain invocations into ChainState objects containing an ID for the invocation and a pointer into the DAG of the function chain. These ChainState objects are serialized using Protocol Buffers [22] and published to a Kafka queue, the Pending Queue, which may be read by the Policy Framework when selecting a function to invoke. Separately, arrival times of each invocation are logged and submitted to the *Arrival Logging Queue* (ALQ), which is read by the Logging Framework and is used in realizing policies, such as reactive scheduling.

The Resource Manager is composed of a *Coordinator* and many worker threads. The Coordinator follows a simple loop: the Policy Framework is called to make a scheduling decision, which returns a function to invoke, and the Coordinator schedules this invocation to occur on a worker thread. Each worker thread, on receiving a function URL, invokes the function over HTTP and sends the start time to the Logging Framework. When the function finishes execution and returns an HTTP response, the ending timestamp is sent to the Logging Framework and the function's DAG is checked to see if there are any child functions to invoke next in the chain. These child functions are added to the in-memory CRQ, which the Policy Framework may read when selecting a function to invoke.

Logging Framework The Logging Framework gathers real-time logs from the QoS Scheduler and serverless functions while they run. The QoS Scheduler's Producer sends timestamps for the arrival times of each function chain invocation and the Resource Manager sends the start and end times of each function being invoked. The serverless functions also report their start and end times to the Logging Framework. In practice, it is useful to track functions' start and end times both on the Resource Manager and in the functions themselves: after the Resource Manager invokes a function, the invocation may spend time in a queue on the serverless platform or be dropped entirely. Consequently, the consumer-side logging represents end-to-end function latency, while the function-side logging represents actual function runtime (which is useful for computing the total concurrent functions at any given time or the monetary cost of executing each function).

As the Logging Framework receives logs from each of its sources, it computes running metrics such as total concurrency of specific functions and chains, number of drops observed in the serverless framework, and inter-arrival times of function chains tracked by the Producer. These metrics are available to be read at any time by the Policy Framework and can be exported to long-term storage to be available for post-hoc analysis.

Function Wrapper The Logging Framework requires serverless functions to send reports from within the serverless platform. These per-function reports contain unique function identifiers combined with a timestamp and are sent at the function start and end. These reports can easily be obtained through third-party libraries (which are agnostic to function code and runtimes) or function wrappers. Python3 introduced function annotations [1] that enable adding arbitrary metadata to function parameters which makes adding a wrapper as concise as a single annotation above the native function.

Policy Framework The Policy Framework provides: (1) a set of configurable policies that can be used to exercise more control over function scheduling than current serverless platforms offer; (2) an intuitive programming interface to express new policies when the provided policies are not sufficient for a specific use case. We have implemented eight policies:

Function-level Allocation: Divide the total function concurrency of the serverless platform equally across all active functions. This policy can be configured to be an unequal split (*e.g.,* limit λ_a to half of the concurrency limit of λ_b), and can be used to prevent lower priority functions, such as loggers, from consuming resources needed by other functions.

Chain-level Allocation: Similar to function-level allocation, but track concurrency of each chain rather than the individual functions composing each chain. This policy can also be configured for an unequal split. Chain-level allocation enables an important single chain function to obtain an appropriate share of the concurrency pool when sharing the pool with lower-priority chains consisting of multiple functions. In addition, developers could use chain-level allocations to ensure fairness amongst their customers/users.

Reactive Concurrency Allocation: At initialization, enforce equal chain-level allocation. Using metrics from the Logging Framework, compute the arrival rate of each function chain and adjust the concurrency limit of each function chain proportionally to the rate. A minimum concurrency allocation is assigned to uncommon chains to prevent starvation. This policy facilitates demand-based chain allocation, rather than FCFS-based function policies deployed on serverless platforms today.

Ongoing Chain Prioritization: Prioritize finishing chains that have been started over new chain invocations, dequeuing from the CRQ before the PQ. This can be used to ensure chains are completed with minimal interruption between beginning and ending the chain.

Shortest Job First: Run the chain with the least-remaining run time. Alternatives, such as lottery scheduling to mitigate unfairness or deadline-aware scheduling to maintain SLAs can also be implemented.

Algorithm 1 Ongoing Chain Prioritization pseudocode

```
1: for queue in List(CRQ, PQ) do
2:     if queue.notEmpty() then
3:         return queue.pop()
4:     end if
5: end for
6: return None
```

Explicit Priority Assignment: Assign priorities to functions and chains. At runtime, the highest priority function is chosen first. This policy also supports weighted priority enforcement (*e.g.,* for every n λ_a's invoked, invoke $\frac{n}{2}$ λ_b's). Priority scheduling benefits cases when user-facing, latency-sensitive applications share a framework with long-lived background applications.

Hybrid Scheduler: Hybrid scheduling allows developers to specify where a function should run, for example in an edge/cloud setting. Such policies reduce latencies for certain workloads, ensure privacy or regulatory compliance, or simply to keep costs low in a private cloud.

Resource-aware Scheduler: A resource-aware scheduler can incorporate available containers/VMs or other available resources such as CPU, concurrency, or memory when scheduling. For example, armed with knowledge of burst intolerance, a resource-aware scheduler could slowly ramp up demands. Alternatively, resource-aware schedulers can take concurrency limits into account and schedule chains accordingly (*e.g.,* only let 333 Fan-2 workloads run at a time with a 1000 concurrency limit, since 333×3 functions will run in a steady state).

In addition to each of these policies, users can implement custom policies as they see fit. Each policy is currently expressed as a Python function that is called whenever a worker thread on the QoS Scheduler is ready to invoke another function. The function has access to all of the serverless environment state provided by the Logging Framework, as well as the PQ and CRQ, and it must return a function to invoke from one of these queues.

Sample psuedocode from two policies are shown in Algorithms 1 and 2. In Algorithm 1, the Ongoing Chain Prioritization first dequeues from CRQ when possible, otherwise defaults to PQ (`List(CRQ, PQ)`, line 1). A ChainState object is returned (line 3), which contains the remaining DAG, and allows the correct function to be scheduled. In the Chain-level Allocation code in Algorithm 2, the function `getChainsWithAlreadyUsedQuota()` keeps track of chains that have used their fair share quota, and function `getQueuesSortedByPriority()` simply iterates over a list of queues sorted by priority and pulls the next ChainState object from the highest non-empty queue.

Algorithm 2 Chain-level Allocation pseudocode

```
chainsToAvoid ← getChainsWithAlreadyUsedQuota()
for queue in getQueuesSortedByPriority() do
    for chain in queue do
        if chain not in chainsToAvoid then
            return chain
        end if
    end for
end for
return None
```

5 EVALUATION

This section evaluates Sequoia. First, microbenchmarks are presented to understand proxy overheads. Second, limitations in Section 2 are revisited, with policy-based solutions implemented to fix the problems. Third, our scheme is shown to adhere to policies in production environments. Last, realistic application workloads are evaluated. Unless otherwise stated, all results are on AWS, with Sequoia deployed as a proxy.

5.1 Microbenchmarks

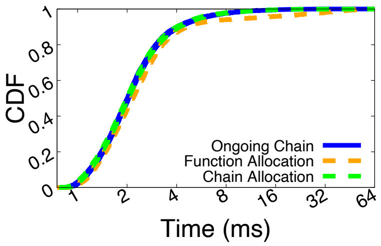


Figure 10: Sequoia overhead

This subsection investigates the overhead of Sequoia. Figure 10 presents the latency of a function request traversing our framework for three different policies. In a Mixed-Chain workload, a burst of 1,000 for each chain is sent to the proxy simultaneously. The plot shows a CDF of the time it takes from the start of the policy framework making a decision to the moment before the function is invoked. Latencies are less than 4.1 ms for the 90^{th}-percentile over all three policies. We believe these overheads can be further reduced with a more optimized implementation.

To measure the overhead of Sequoia logging on serverless functions, we recorded the time spent collecting function logs and sending them to the Logging Framework. Over a burst of 1,000 invocations we observed 99% of the invocations completed logging within 3.34 milliseconds. The median logging time was 2.17 milliseconds. In the case of AWS Lambda, functions are priced per 100 milliseconds [28], so logging overheads minimally effect the cost of most workloads.

5.2 Mitigating previous limitations

This subsection shows Sequoia's policies can mitigate the limitations observed in Section 2.

Inconsistent/incorrect concurrency limits As defined in Section 4, our scheme can limit concurrent invocations. While explicit results are not shown due to brevity, Sequoia ensures concurrency limits are correctly enforced over all providers.

Inefficient resource allocation Next, the performance of linear chains is examined in Figure 11. The x-axis varies the chain length, and the y-axis shows total number of VMs allocated over all five runs.

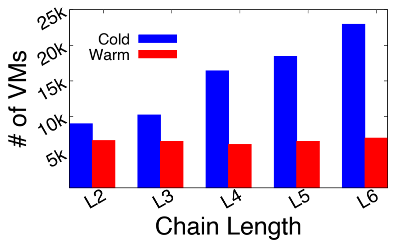


Figure 11: Resource allocation in linear chain analysis

Storage-based chains are analyzed under *normal* (AWS) and QoS (Sequoia) scheduling. For Linear-n, the QoS scheduler admits λ_1's at a rate of $1000/n$. The graph shows a trade-off exists in scheduling: by limiting function chain admission, the QoS scheduler uses significantly less VMs than the normal scheduler. This occurs because limiting function chain admission allows for more opportunities of VM reuse. In the normal case, bursts are allowed for which a burst-worth of VMs must be provisioned to handle the load. Sequoia takes longer to complete because of the limiting (not shown; up to 50% longer in Linear-6 case). This, however, provides a new point in the design space, in which cloud providers can rate limit when under heavy load, or simply rate limit certain customers (*e.g.,* such as free-tier customers) to provision less VMs. Note maintaining idle VMs consumes memory, which is an expensive bottleneck in cloud deployments.

Mid-chain drops Mid-chain drops are caused by exceeding concurrency on platforms with strict concurrency limits. Concurrency overflow can be controlled by rate-limiting chain invocations. We reran the experiment on AWS and IBM with storage triggers in Figure 4 with Resource-aware Scheduling and achieved 97% chain completions on IBM and 100% on AWS without compromising concurrency (figure omitted for space).

Abrupt concurrency drops Figure 8a shows concurrency collapse for Fan-N workloads in AWS. Our unconfirmed suspicion is the burst causes issues with resource allocation.

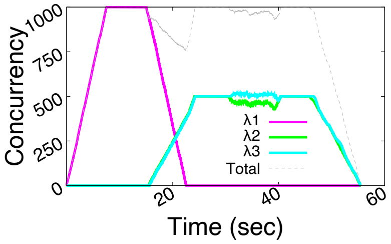


Figure 12: Resource-aware policy prevents concurrency drop

Figure 12 shows the performance of a Resource-aware Ongoing policy for Fan-2. The invocation rates are limited under the concurrency limit with the Resource-aware policy. This simple limiting is very effective: the total time to complete the burst is 55 seconds, whereas the time is over 330 seconds in the collapse case, a 5.5× inflation.

5.3 QoS-based policy realization

This section shows Sequoia accurately realizes various policies in a performant manner. Tests in this section utilize a MixedChain workload.

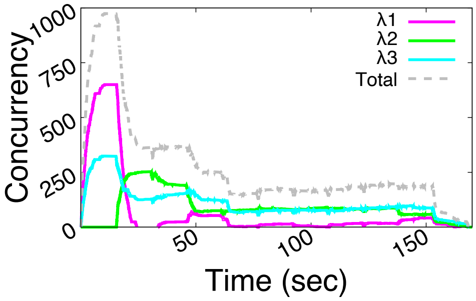
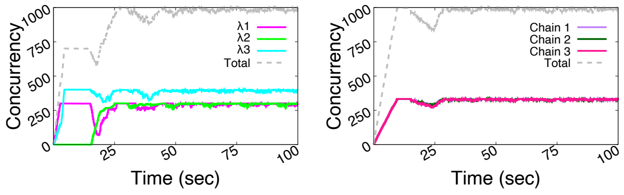


Figure 13: AWS MixedChain baseline

Figure 13 shows the results of Burst-1000, meaning each chain sends a burst of 1,000 requests. The graph is a baseline case with no proxy. The existing platform performs poorly: average concurrency utilization is low (< 30%), which increases runtimes by 3× compared to the ideal case. In addition, nearly 70% of the demand is not completed. Below, Sequoia-implemented policies are applied to the same workload.



(a) Function-level policy **(b) Chain-level policy**

Figure 14: QoS policy validation

Function-level Allocation Figure 14a shows the results of a fair function scheduler. In MixedChain there are 10 total functions across the 3 chains, so each function within a chain should receive one-tenth the concurrency limit. Since λ_1 appears $\frac{3}{10}$, λ_2 appears $\frac{3}{10}$, and λ_3 appears $\frac{4}{10}$, the concurrency should be 300, 300, and 400 respectively. The figure shows convergence to these numbers.

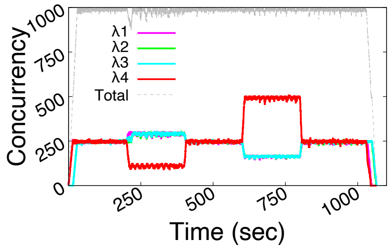


Figure 15: Reactive Concurrency Sharing with adaptive workload

Chain-level Allocation Figure 14b shows the results of a fair chain scheduler. MixedChain has three chains and individual chains should receive equal share even though each chain has a different number of functions. The graph shows how chains with three functions (Fan-2 and Linear-3) consume the same share of concurrency as the Combo chain with 4 functions (y-axis shows the real-time concurrency of all functions within each chain).

Reactive Concurrency Scheduling Figure 15 shows how the Reactive Concurrency policy converges when demands change. First, a Fan-2 and Single function (λ_4) are started with equal demands (17 invocations per second, or IPS). At $t = 200$ Fan-2 becomes 20 IPS and λ_4 becomes 8 IPS (a 2.5:1 ratio). Then, at $t = 400$ IPS ratios again become equal (17 IPS). At $t = 600$ the λ_4 IPS becomes 3× the Fan-2 IPS. Finally, at $t = 800$ ratios are again equal. As shown, the realized concurrency of each chain closely tracks the changes in incoming requests.

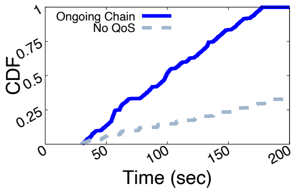


Figure 16: Ongoing Chain Prioritization completion times

This shows our scheme can be reactive to changes in demands. It also shows Sequoia can effectively mitigate the HTTP prioritization problem outlined in Figure 6a by considering resources required for background functions (note λ_1, λ_2, and λ_3 obtain similar concurrency).

Ongoing Chain Prioritization Figure 16 shows a CDF of chain completion times for a Resource-aware Ongoing Chain Prioritization scheme and a baseline with no proxy for a Burst-1000 MixedChain workload. The Ongoing scheme significantly decreases function chain completion times. The baseline scheme only finishes 33% of its chains, whereas Ongoing sustains no chain loss. At the 25^{th}-percentile, the chains with no QoS take 2.5× longer than the QoS policy.

Explicit Priority Assignment Figure 17 shows the results from an explicit priority assignment in Fan-2 where λ_1's are assigned the highest priority, λ_2's the second highest, and λ_3's the lowest. Here, strict scheduling is enforced, with λ's running in appropriate order. When invoking a function, the current implementation checks the next 100 functions currently queued and picks the first function with highest priority.

Hybrid Scheduler Figure 18 shows a hybrid scheduling policy. Here, there exists two deployments that can run serverless functions: a local edge cloud running OpenWhisk and AWS. The edge cloud has a capacity of 200 concurrent functions and a demand of 225 concurrent functions is applied. Our proxy limits concurrency on the edge cloud to 200 functions, while sending the 25 additional concurrent functions to the remote cloud.

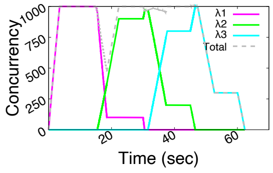


Figure 17: Explicit Priority in Fan-2 Burst-1200

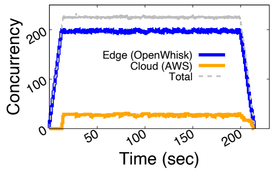


Figure 18: Hybrid in MixedChain

Chain	Ave function run times (s)
NLP	λ_1: 1.6
CV	λ_1: 0.2; λ_2: 1.1 λ_3: 8.7
Video	λ_1: 2.1; λ_2: 1.1; λ_3: 0.9
Compile	λ_1: 0.5; λ_i: 19.6-20.5

Table 2: Summary of realistic chain run times

5.4 Realistic Workloads

This section evaluates the following workloads:

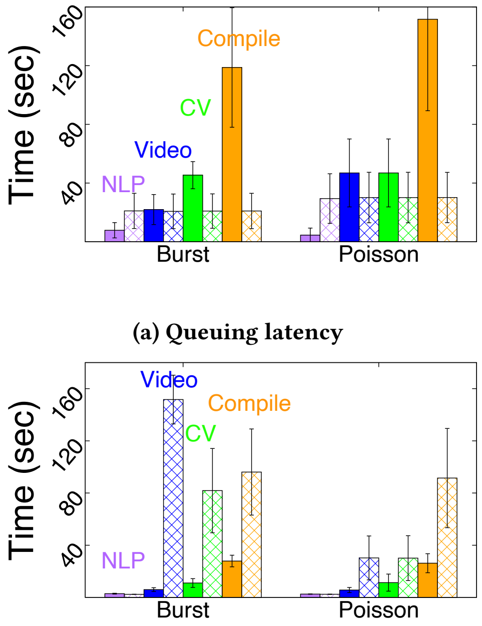
NLP: A single function that builds a trigram bag-of-words model from an input text file.

CV: A Fan-2 chain which performs object classification [49] (λ_2) and detection [79] (λ_3) on an input image.

Video: A Linear-3 chain that generates a preview of a sample video clip, grayscales, and then reverses the clip.

Compilation: A Fan-7 chain that compiles sample source code to seven different architectures.

Functions within the workloads run from 200 ms to over 20 seconds, shown in Table 2. The goal is to have a mix of short and long chains with varying run times.



(a) Queuing latency

(b) Execution latency

Figure 19: Execution latency for real workloads. Bars for SJF are solid and FIFO are dashed.

Runtime analysis The above workloads are run simultaneously for shortest-job first (SJF) and FIFO queuing, with the Resource-aware rate limiting applied to both algorithms for fair comparison. Two workloads are considered: each chain invokes a burst of 1,000 requests and Poisson arrivals where all chains receive 30 request/sec on average. Figure 19a shows the average queuing latency, defined as the time between when the Producer receives the request and the time the consumer schedules the request on AWS, and standard deviation. NLP, the shortest workload, has 2.7-6.4× less queuing times in SJF (solid bars) than FIFO (dashed bars). FIFO does not prioritize, hence queuing is the same for all chains.

Figure 19b shows the average execution latency, defined as the time between the first function in the chain starting and the last function in the chain ending. Again, SJF correctly prioritizes workloads and achieves significant benefits over FIFO: Video ranging from 5.3-25× faster, CV ranging from 2.7-7.3×, and Compile from 3.4-3.5×. Video sees larger benefit because it is a three-function linear chain and SJF reduces queuing between dependent functions in the DAG. NLP, a single function, shows no difference.

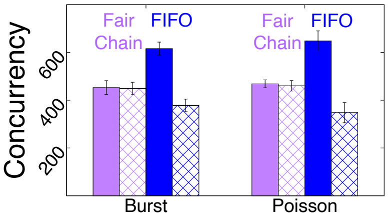


Figure 20: Fairness with realistic workloads: Compile (solid), sCV (dashed)

Fairness analysis We modify the workload to demonstrate fairness. Two computationally expensive applications are chosen since such workloads are likely to consume significant resources and may cause issues if left unchecked. The Compile workload is run with a *single-function* object detection workload (denoted sCV). For Burst, sCV invokes 500 bursts and Compile invokes 3000 bursts. For Poisson, sCV invokes 60 requests per second and Compile invokes 10 requests per second. Demands are saturated in both cases. Figure 20 shows average concurrent invocation rates of each chain, defined as the sum of all functions running concurrently within the chain. The fair chain scheduler is compared to a FIFO scheduler, both with Resource-aware rate limiting. The fair chain scheduler properly ensures large chains do not consume an unfair share of the available concurrency, showing an equal concurrency for sCV (dashed bars) and Compile (solid bars). FIFO, however, schedules on the function-level, and hence Compile, with a large Fan-7, consumes 1.62-1.86× more of the available concurrency than sCV (which has a single function).

6 DISCUSSION

This section discusses assumptions, limitations, and motivations for our work.

Measurement study Measurements in Section 2 were performed November 2019 to May 2020. The results motivate the need for quality-of-service offerings, which is the focus of our paper. We fully expect trends, results, and architectures to change as serverless providers improve their ecosystems. For example, Google deprecated the *CPUMilliSeconds* quota outlined in Section 2 and substituted it with a maximum number of instances quota system on September 8, 2020 [76]. While recent advances continue to improve serverless offerings [88], the need for QoS persists as functions increasingly face spatial and temporal restrictions (such as privacy-based functions running at the edge only, or limited edge resources being prioritized for interactive functions) or require different customers or functions within an application to obtain differential treatment. As such, Sequoia can help developers and providers further manage their serverless workloads.

Known chains We assume known function chains. This holds true over many scenarios. For example, developers may supply providers with a chain's call graph. In the case of composition frameworks (*e.g.,* AWS Step Functions), developers already specify the call graph. Even in non-composition cases, developers upload function source code to providers, as well as configuration files to specify event-triggers. Here, providers can co-opt chain analysis techniques to derive chains [45, 46, 74, 101]. In addition, service mesh frameworks [16, 21, 42, 62, 90] monitor, secure, debug, and connect components within a microservice. Service meshes exploit RPCs to generate dependency graphs between microservice components, which can also aid in serverless call chain identification. Finally, our Logging Framework can be augmented to learn chain behavior over time.

Serverless workloads As serverless proliferates, we believe more complex function chains will quickly emerge. Recent studies indicate this trend may hold. In Azure, 46% of serverless applications contain more than one function [88]. And while one study [91] finds few multi-function serverless applications in the AWS Serverless Application Repository [23], we analyze the IBM Serverless Code Pattern database [24] and find roughly 52% contain more than one function. Regardless of chain size, our QoS framework benefits both single and multi-function chains. As serverless matures, function execution times may also lengthen. Currently, 50% of functions execute for longer than one second in Azure [88], and other studies present use cases for functions that execute for 10-100 seconds [41, 51, 89]. We believe diverse workloads, from chain sizes to function run times, motivates the need for better QoS because all must execute over a common infrastructure.

Increasing concurrency limits One may ask if increasing concurrency limits can solve some of the issues outlined in this paper. Such solution has several limitations. First, developers rely on concurrency limits to manage their workloads–one study found 88% of organizations have a concurrency limit defined for at least one function [80]. Cost can be managed with concurrency limits, and the effect of bugs or misconfigurations can be mitigated. In addition, raising limits may not be immune to an unusual rise in demand. Even in the absence of limits, we find many cloud providers do not scale immediately to meet demand (*e.g.,* with bursts) and hence queuing and QoS scheduling are inevitable.

Drop-in design Studying QoS on black-box actual deployments (*i.e.,* AWS) is difficult because control is limited. As such, our evaluation features a proxy-based solution. Sequoia's drop-in design, however, could be implemented in a variety of ways: via an as-a-Service offering from a third-party, implemented by the developer, or integrated into the cloud provider's framework. Our design highlights the logical components necessary for QoS, and our work studies unique challenges and solutions for serverless-based QoS.

7 RELATED WORK

This section details major classes of related work.

Serverless platform measurements Many studies have provided measurements of serverless platform performance and architectures. Preliminary serverless measurement works study throughput and performance of serverless platforms [58, 61, 66]. Wang et al [99] present a detailed study of three major serverless providers and characterizes their architectures and performances. Shahrad et al [88] study serverless application characteristics and workload patterns. Our study focuses on QoS-related measurements, such as concurrency limits, queuing and scheduling techniques, and reliability.

Serverless scheduling and resource allocation Many works consider scheduling and resource allocation in serverless environments. Archipelago [91] is a latency-sensitive serverless framework that scales scheduling, proactively creates sandboxes, and schedules functions with a shortest-remaining slack first algorithm [47, 85]. Archipelago is complementary to Sequoia, in that we can utilize its architecture to further scale, and Archipelago can implement our policy framework to enable more generalized QoS. Similar to Archipelago, GrandSLAm [53] reorders function requests to prioritize execution of functions with lower remaining slack time. Real-time serverless workloads are supported in [70] via predictive container management and admission control on application requests. FnSched [96] manages invoker resources (VMs/containers that run functions) to ensure unnecessary latency is not incurred, while minimizing cost. Finally, [52] centrally schedules lambdas on CPU cores instead of servers to reduce latency. While some of the above works provide SLAs and function reordering (which Sequoia also provides), none of the works holistically study QoS to the extent our work does. Previous works do not study QoS-related issues presented in Section 2 nor enumerate serverless QoS policies as in Section 4 such as fair chain scheduling.

Other works design new primitives for serverless applications [26] or examine serverless function chains in more detail, such as merging functions to reduce costs [38], optimizing function placement [64], or describing trade-offs in building function chains [32]. These works do not provide a general QoS scheduling framework as we do.

Cold starts Many of the aforementioned measurement studies analyze the cold start problem. In addition, studies show the overheads invoking function chains across platforms [43], measure overheads within cold start and Docker networking [59], and analyze different cold start states [59]. None of these works, however, explicitly examine QoS-based limitations as our study does.

To mitigate cold start latencies, many techniques have been proposed [33, 34, 72, 73, 83]. These techniques are mostly complementary to our study as resource-aware policies can take into consideration cold-start overheads.

Serverless storage and communication Many works examine improvements and issues with serverless storage and communication [25, 54, 56, 57, 77, 84, 94, 104]. Our study does not consider storage and communication, but implementing policy frameworks for such mechanisms is interesting future work.

General cloud schedulers A variety of task and cluster management systems include scheduling subsystems. Many architectures have been designed, from distributed [68, 75, 78, 78, 100] to centralized [48, 97, 98] techniques. Kubernetes can assign QoS to pods [19], but cannot provide function-level QoS. Serverless workloads provide many unique challenges, such as short-lived ephemeral entities that must be packed tightly for efficiency. Techniques used in previous works, such as migration, may not easily be utilized in serverless workloads. As a result, our work designs a serverless-native solution.

8 CONCLUSION

This paper studies quality-of-service (QoS) scheduling frameworks for serverless platforms. A measurement analysis indicates current serverless QoS offerings are relatively nascent and there exist numerous issues with scheduling, limiting, and managing a rich ecosystem of serverless applications today. To address these issues, a system called Sequoia allows administrators or developers to specify and enforce flexible policies. Sequoia is designed as a drop-in framework that improves overall management by enabling QoS in a lightweight and low overhead manner. Evaluations with controlled and realistic workloads show Sequoia effectively mitigates issues uncovered in the measurement study, eliminating mid-chain drops, reducing queuing times by up to 6.4×, enforcing tight chain-level fairness, and improving run-time performance up to 25×.

ACKNOWLEDGEMENTS

We thank Sudeep Galgali and Nithin Veer Reddy Kankanti for their help analyzing and running serverless chains. This work partially funded by NSF-1908910.

REFERENCES

[1] 2006. PEP 3107 – Function Annotations. https://www.python.org/dev/peps/pep-3107/.

[2] 2019. Amazon AWS Step Functions. https://aws.amazon.com/step-functions/.

[3] 2019. Amazon Lambda. https://aws.amazon.com/lambda/.

[4] 2019. Apache Software Foundation. http://openwhisk.incubator.apache.org/.

[5] 2019. AWS Lambda Limits. https://docs.aws.amazon.com/lambda/latest/dg/limits.html.

[6] 2019. Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.

[7] 2019. Azure Functions scale and hosting. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale.

[8] 2019. Azure Functions scale and hosting. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale.

[9] 2019. Firecracker: Secure and fast microVMs for serverless computing. https://firecracker-microvm.github.io.

[10] 2019. Google Cloud Functions. https://cloud.google.com/functions/.

[11] 2019. Google Cloud Functions: Background Functions. https://cloud.google.com/functions/docs/writing/background.

[12] 2019. IBM Cloud Functions. https://www.ibm.com/cloud/functions/.

[13] 2019. IBM Cloud Functions: System details and limits. https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-limits.

[14] 2019. IBM Composer. https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-pkg_composer.

[15] 2019. IronFunctions by iron.io . https://github.com/iron-io/functions.

[16] 2019. Istio. https://istio.io.

[17] 2019. Knative: Kubernetes-based platform to deploy and manage modern serverless workloads. https://knative.dev.

[18] 2019. Kubeless: The Kubernetes Native Serverless Framework. https://kubeless.io.

[19] 2019. KUBERNETES: Configure Quality of Service for Pods. https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/.

[20] 2019. OpenFaaS - Serverless Functions Made Simple . https://github.com/openfaas/faas.

[21] 2019. OpenZipkin: A distributed tracing system. https://zipkin.io.

[22] 2019. Protocol Buffers. https://developers.google.com/protocol-buffers/.

[23] 2020. AWS Serverless Application Repository. https://aws.amazon.com/serverless/serverlessrepo/.

[24] 2020. IBM Serverless Code Patterns. https://developer.ibm.com/technologies/serverless/patterns/.

[25] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[26] Z. Al-Ali, S. Goodarzy, E. Hunter, S. Ha, R. Han, E. Keller, and E. Rozner. 2018. Making Serverless Computing More Serverless. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 456–459. https://doi.org/10.1109/CLOUD.2018.00064

[27] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). *SIGCOMM Comput. Commun. Rev.* 40, 4 (Aug. 2010), 63–74. https://doi.org/10.1145/1851275.1851192

[28] Amazon. [n.d.]. https://aws.amazon.com/lambda/pricing/.

[29] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. ACM, New York, NY, USA, 263–274. https://doi.org/10.1145/3267809.3267815

[30] AWS. 2019. Security Overview of AWS Lambda. https://d1.awsstatic.com/whitepapers/Overview-AWS-Lambda-Security.pdf.

[31] I. Baldini, P. Castro, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, and P. Suter. 2016. Cloud-Native, Event-Based Programming for Mobile Applications. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 287–288. https://doi.org/10.1109/MobileSoft.2016.063

[32] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The Serverless Trilemma: Function Composition for Serverless Computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Vancouver, BC, Canada) *(Onward! 2017)*. ACM, New York, NY, USA, 89–103. https://doi.org/10.1145/3133850.3133855

[33] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2018. Putting the "Micro" Back in Microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 645–650. https://www.usenix.org/conference/atc18/presentation/boucher

[34] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages. https://doi.org/10.1145/3342195.3392698

[35] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A Serverless Framework for End-to-End ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3357223.3362711

[36] Ryan Chard, Tyler J. Skluzacek, Zhuozhao Li, Yadu Babuji, Anna Woodard, Ben Blaiszik, Steven Tuecke, Ian Foster, and Kyle Chard. 2019. Serverless Supercomputing: High Performance Function as a Service for Science. arXiv:1908.04907 [cs.DC]

[37] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[38] Tarek Elgamal, Atul Sandur, Klara Nahrstedt, and Gul Agha. 2018. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. arXiv:1811.09721 [cs.DC]

[39] L. Feng, P. Kudva, D. Da Silva, and J. Hu. 2018. Exploring Serverless Computing for Neural Network Training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 334–341. https://doi.org/10.1109/CLOUD.2018.00049

[40] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi

[41] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. 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)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[42] 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. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud &#38; Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[43] Pedro Garcia Lopez, Marc Sanchez-Artigas, Gerard Paris, Daniel Barcelona Pons, Alvaro Ruiz Ollobarren, and David Arroyo Pinto. 2018. Comparison of FaaS Orchestration Systems. *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)* (Dec 2018). https://doi.org/10.1109/ucc-companion.2018.00049

[44] V. Giménez-Alventosa, Germán Moltó, and Miguel Caballer. 2019. A framework and a performance assessment for serverless MapReduce on AWS Lambda. *Future Generation Computer Systems* 97 (2019), 259 – 274. https://doi.org/10.1016/j.future.2019.02.057

[45] David Grove and Craig Chambers. 2001. A Framework for Call Graph Construction Algorithms. *ACM Trans. Program. Lang. Syst.* 23, 6 (Nov. 2001), 685–746. https://doi.org/10.1145/506315.506316

[46] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call Graph Construction in Object-oriented Languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Atlanta, Georgia, USA) *(OOPSLA '97)*. ACM, New York, NY, USA, 108–124. https://doi.org/10.1145/263698.264352

[47] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. 2003. Size-Based Scheduling to Improve Web Performance. *ACM Trans. Comput. Syst.* 21, 2 (May 2003), 207–233. https://doi.org/10.1145/762483.762486

[48] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 295–308.

[49] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861 [cs.CV]

[50] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2017. Serving deep learning models in a serverless platform. *CoRR* abs/1710.08460 (2017). arXiv:1710.08460 http://arxiv.org/abs/1710.08460

[51] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99Computing (Santa Clara, California) *(SoCC '17)*. ACM, New York, NY, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[52] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized Core-Granular Scheduling for Serverless Functions. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 158–164. https://doi.org/10.1145/3357223.3362709

[53] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. ACM, New York, NY, USA, Article 34, 16 pages. https://doi.org/10.1145/3302424.3303958

[54] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. 2018. Iron: Isolating Network-based CPU in Container Environments. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 313–328. https://www.usenix.org/conference/nsdi18/presentation/khalid

[55] Youngbin Kim and Jimmy Lin. 2018. Serverless Data Analytics with Flint. arXiv:1803.06354 [cs.DC]

[56] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 789–794. https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

[57] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[58] Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, and Dan Li. 2019. Understanding Open Source Serverless Platforms. *Proceedings of the 5th International Workshop on Serverless Computing - WOSC '19* (2019). https://doi.org/10.1145/3366623.3368139

[59] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. 2018. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 159–169. https://doi.org/10.1109/IC2E.2018.00039

[60] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2016. Improving Resource Efficiency at Scale with Heracles. *ACM Trans. Comput. Syst.* 34, 2, Article 6 (May 2016), 33 pages. https://doi.org/10.1145/2882783

[61] Pedro García López, Marc Sánchez Artigas, Gerard París, Daniel Barcelona Pons, Álvaro Ruiz Ollobarren, and David Arroyo Pinto. 2018. Comparison of Production Serverless Function Orchestration Systems. *CoRR* abs/1807.11248 (2018). arXiv:1807.11248 http://arxiv.org/abs/1807.11248

[62] Jonathan Mace and Rodrigo Fonseca. 2018. Universal Context Propagation for Distributed System Instrumentation. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. ACM, New York, NY, USA, Article 8, 18 pages. https://doi.org/10.1145/3190508.3190526

[63] K. Mahajan, D. Figueiredo, V. Misra, and D. Rubenstein. 2019. Optimal Pricing for Serverless Computing. In *2019 IEEE Global Communications Conference (GLOBECOM)*. 1–6.

[64] Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, and Marin Litoiu. 2019. Optimizing Serverless Computing: Introducing an Adaptive Function Placement Algorithm. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering* (Toronto, Ontario, Canada) *(CASCON '19)*. IBM Corp., Riverton, NJ, USA, 203–213. http://dl.acm.org/citation.cfm?id=3370272.3370294

[65] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[66] G. McGrath and P. R. Brenner. 2017. Serverless Computing: Design, Implementation, and Performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 405–410. https://doi.org/10.1109/ICDCSW.2017.36

[67] G. McGrath, J. Short, S. Ennis, B. Judson, and P. Brenner. 2016. Cloud Event Programming Paradigms: Applications and Analysis. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. 400–406. https://doi.org/10.1109/CLOUD.2016.0060

[68] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[69] Stefan Nastic and Schahram Dustdar. 2018. *Towards Deviceless Edge Computing: Challenges, Design Aspects, and Models for Serverless Paradigm at the Edge.* Springer International Publishing, Cham, 121–136. https://doi.org/10.1007/978-3-319-73897-0_8

[70] Hai Duc Nguyen, Chaojie Zhang, Zhujun Xiao, and Andrew A. Chien. 2019. Real-Time Serverless: Enabling Application Performance Guarantees. In *Proceedings of the 5th International Workshop on Serverless Computing* (Davis, CA, USA) *(WOSC '19)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3366623.3368133

[71] Christopher Null. 2019. The state of serverless: 6 trends to watch. https://techbeacon.com/enterprise-it/state-serverless-6-trends-watch.

[72] E. Oakes, L. Yang, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2017. Pipsqueak: Lean Lambdas with Large Libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 395–400. https://doi.org/10.1109/ICDCSW.2017.32

[73] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[74] Matthew Obetz, Stacy Patterson, and Ana Milanova. 2019. Static Call Graph Construction in AWS Lambda Serverless Applications. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/obetz

[75] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 69–84. https://doi.org/10.1145/2517349.2522716

[76] Google Cloud Platform. 2020-07-23. [Action Required] Some Cloud Functions quotas will be removed on September 8, 2020. Electronic Mail communication.

[77] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[78] Hang Qu, Omid Mashayekhi, Chinmayee Shah, and Philip Levis. 2018. Decoupling the Control Plane from Program Control Flow for Flexibility and Performance in Cloud Computing. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/3190508.3190516

[79] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2015. You Only Look Once: Unified, Real-Time Object Detection. arXiv:1506.02640 [cs.CV]

[80] Datadog Research. 2020. The State of Serverless. https://www.datadoghq.com/state-of-serverless/.

[81] Emrah Samdan. 2019. AWS Lambda: Asynchronous Invocation. https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html.

[82] Emrah Samdan. 2019. Configuring Amazon S3 Event Notifications. https://docs.aws.amazon.com/AmazonS3/latest/dev/NotificationHowTo.html.

[83] Emrah Samdan. 2019. Dealing with cold starts in AWS Lambda. https://medium.com/thundra/dealing-with-cold-starts-in-aws-lambda-a5e3aa8f532.

[84] Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. 2017. Data-driven Serverless Functions for Object Storage. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (Las Vegas, Nevada) *(Middleware '17)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/3135974.3135980

[85] Linus Schrage. 1968. A Proof of the Optimality of the Shortest Remaining Processing Time Discipline. *Operations Research* 16, 3 (1968), 687–690. http://www.jstor.org/stable/168596

[86] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open versus Closed: A Cautionary Tale. In *Proceedings of the 3rd Conference on Networked Systems Design Implementation - Volume 3* (San Jose, CA) *(NSDI'06)*. USENIX Association, USA, 18.

[87] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. ACM, New York, NY, USA, 1063–1075. https://doi.org/10.1145/3352460.3358296

[88] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Irfan Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *ATC*. USENIX. https://www.usenix.org/conference/atc20/presentation/shahrad

[89] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: serverless linear algebra. arXiv:1810.09679 [cs.DC]

[90] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

[91] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2019. Archipelago: A Scalable Low-Latency Serverless Platform. arXiv:1911.09849 [cs.DC]

[92] Josef Spillner and Serhii Dorodko. 2017. Java Code Analysis and Transformation into AWS Lambda Functions. arXiv:1702.05510 [cs.DC]

[93] Josef Spillner, Cristian Mateos, and David A. Monge. 2018. FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC. In *High Performance Computing*, Esteban Mocskos and Sergio Nesmachnow (Eds.). Springer International Publishing, Cham, 154–168.

[94] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. 2020. A Fault-Tolerance Shim for Serverless Computing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/3342195.3387535

[95] A. Sriraman and T. F. Wenisch. 2018. μ Suite: A Benchmark Suite for Microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 1–12. https://doi.org/10.1109/IISWC.2018.8573515

[96] Amoghvarsha Suresh and Anshul Gandhi. 2019. FnSched: An Efficient Scheduler for Serverless Functions. In *Proceedings of the 5th International Workshop on Serverless Computing* (Davis, CA, USA) *(WOSC '19)*. Association for Computing Machinery, New York, NY, USA, 19–24. https://doi.org/10.1145/3366623.3368136

[97] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, and et al. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC '13)*. Association for Computing Machinery, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2523616.2523633

[98] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 18, 17 pages. https://doi.org/10.1145/2741948.2741964

[99] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[100] Zhijun Wang, Huiyang Li, Zhongwei Li, Xiaocui Sun, Jia Rao, Hao Che, and Hong Jiang. 2019. Pigeon: An Effective Distributed, Hierarchical Datacenter Job Scheduler. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 246–258. https://doi.org/10.1145/3357223.3362728

[101] Stefan Winzinger and Guido Wirtz. 2019. Model-based Analysis of Serverless Applications. In *Proceedings of the 11th International Workshop on Modelling in Software Engineerings* (Montreal, Quebec, Canada) *(MiSE '19)*. IEEE Press, Piscataway, NJ, USA, 82–88. https://doi.org/10.1109/MiSE.2019.00020

[102] Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. 2019. CSPOT: Portable, Multi-scale Functions-as-a-service for IoT. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing* (Arlington, Virginia) *(SEC '19)*. ACM, New York, NY, USA, 236–249. https://doi.org/10.1145/3318216.3363314

[103] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. 2016. Building a Chatbot with Serverless Computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs* (Trento, Italy) *(MOTA '16)*. ACM, New York, NY, USA, Article 5, 4 pages. https://doi.org/10.1145/3007203.3007217

[104] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the Gap Between Serverless and Its State with Storage Functions. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3357223.3362723