Atoll: A Scalable Low-Latency Serverless Platform

Arjun Singhvi
University of Wisconsin-Madison

Arjun Balasubramanian*
Amazon Web Services

Kevin Houck*
Amazon Web Services

Mohammed Danish Shaikh*
Google

Shivaram Venkataraman
University of Wisconsin-Madison

Aditya Akella
University of Texas-Austin and Google

Abstract

With user-facing apps adopting serverless computing, good latency performance of serverless platforms has become a strong fundamental requirement. However, it is difficult to achieve this on platforms today due to the design of their underlying control and data planes that are particularly ill-suited to short-lived functions with unpredictable arrival patterns. We present Atoll, a serverless platform, that overcomes the challenges via a ground-up redesign of the control and data planes. In Atoll, each app is associated with a latency deadline. Atoll achieves its per-app request latency goals by: (a) partitioning the cluster into (semi-global scheduler, worker pool) pairs, (b) performing deadline-aware scheduling and proactive sandbox allocation, and (c) using a load balancing layer to do sandbox-aware routing, and automatically scale the semi-global schedulers per app. Our results show that Atoll reduces missed deadlines by ~66× and tail latencies by ~3× compared to state-of-the-art alternatives.

CCS Concepts

• **Distributed architectures → Cloud computing**.

Keywords

Serverless Computing, Low-Latency, Scalable

ACM Reference Format:
Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A Scalable Low-Latency Serverless Platform. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3472883.3486981

*Work done while at University of Wisconsin-Madison

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11…$15.00
https://doi.org/10.1145/3472883.3486981

1 Introduction

Recent trends in cloud computing point towards increased adoption of micro-services to design and deploy online applications [34]. Each micro-service can be independently deployed and managed, and collectively the microservices implement what used to be realized as large monolithic software. To meet this demand imposed by independently scalable micro-services, simplify programming, and relieve programmers from provisioning and elastic scaling responsibilities, cloud providers now offer Function-as-a-Service (FaaS) or *serverless* computing [3, 4, 8].

While serverless platforms simplify deploying such workloads in the cloud, they are unable to provide good latency performance, which is particularly important for user-facing applications [18, 29, 40, 43, 47]. Our analysis of serverless workloads indicate that that the platform needs to handle functions that are very short lived, have unpredictable arrival patterns, and require potentially expensive setup of sandboxes in which functions are executed [55] (§2). The platform's underlying control and data plane designs are a poor fit for these attributes, making it difficult to ensure that serverless request latencies are within the bounds expected by the user.

In a serverless platform, the control plane consists of load balancing and scheduling layers that are responsible for routing incoming requests to the underlying machines, and the data plane corresponds to the sandboxes in which functions are actually executed. To meet latency expectations, various aspects of the control and data planes need to be redesigned, bringing to fore fundamental challenges and trade-offs. Firstly, with data plane/sandbox management, there is an intrinsic trade-off between reducing the number of requests that experience sandbox setup overhead and the memory consumed in keeping sandboxes provisioned (possibly ahead of time). Secondly, the control plane must be scalable yet capable of making optimal scheduling decisions as failing to do so leads to additional setup overheads (e.g., when the control plane fails to find machines that have sandboxes available for reuse) [35]. Finally, existing platforms [3, 4, 8]

do not take into account the native execution time of functions and nor does their function upload workflow allow users to specify request priorities. Thus, it is unclear if or how platforms today prioritize requests to ensure that they are serviced within the expected bounds.

We present Atoll, a serverless platform, designed from the ground-up, that carefully navigates the control and data plane design trade-offs and challenges, and strives to maximize the number of requests that meet their deadline, i.e., finish within the user-specified bounds. The following design choices form the basis of Atoll: (i) a *hierarchical control plane*, in which the different layers are *co-designed*, giving the ability to (ii) *gradually scale* apps across the cluster along with the introduction of (iii) *soft state* in the form of proactively allocated sandboxes and being (iv) *aware of app deadlines*.

Our overall design is scalable at individual component level and performance-awareness permeates the entire architecture. First, Atoll *partitions* the given cluster into a number of smaller worker pools. Each worker pool is managed by a *semi-global* scheduler (SGS); with appropriate sizing of the worker pool, we can ensure that each SGS imposes low scheduling overheads for request execution. To achieve optimal placement and ensure that most incoming requests are served by a ready sandbox, each SGS also tracks the number of requests for every app it is serving, and *proactively* allocates sandboxes to minimize the overheads in critical path of function execution. Crucially, we create these sandboxes as *soft state* where they only use memory resources from a *fixed* sized pool and can be evicted without affecting correctness. While recent efforts [17, 20, 51] proposee techniques to reduce sandbox allocation overheads, they typically come at the cost of isolation and/or code compatibility which precludes their widespread adoption [16] (§8). Atoll's approach of reducing the sandboxes setup in the critical path ensures that the overall request latency is not affected despite any overheads necessary for isolation and/or code compatibility.

Second, Atoll uses a scheduling algorithm within an SGS that is aware of the app latency requirements. This enables us to compute a running *slack*, or the time-to-deadline remaining for a given request, and use a variant of the shortest-remaining-time-first algorithm to minimize the deadlines missed. Here, we leverage that apps running in a cluster have different slacks, and low-slack apps' resource needs can be met by reallocating resources from high-slack ones.

While partitioning a cluster can help lower scheduling overheads, we must determine how requests are routed to each SGS in a cluster. Thus, the third idea in Atoll is to *co-design* the load balancing layer with the scheduling layer. This enables the load balancer layer to use a *sandbox-aware* load balancing policy that can route requests while being aware of the number of sandboxes of different apps proactively allocated in every SGS. In order to simplify the load balancing layer and make it scalable, every app is assigned to a single SGS to begin with and based on the number of requests, the load balancer can scale out (or in) the SGSs assigned to this app. Using an approach that is also aware of sandbox allocation ensures that app performance is minimally affected when scaling across the cluster.

We build Atoll in Go and evaluate our prototype against state-of-the-art alternatives by replaying workloads from a serverless trace published by Microsoft [55] as well as using a collection of apps derived from our analysis of the serverless repository maintained by AWS [7]. Our results show that Atoll reduces the number of deadlines missed by up to ~66× and tail latencies by ~3× over state-of-the-art alternatives while occupying upto 22% additional memory due to worst case load proactive sandbox allocation. We find that the benefits provided by Atoll hold under a spectrum of allocation overheads that reflect various state-of-the-art techniques (§7.4).

2 Background and Motivation

We start with a primer on serverless computing and characterize popular apps in the AWS serverless repository [9]. Based on our analysis, we state our requirements along with why current platforms fall short.

2.1 Serverless Computing Background

In serverless computing, the user writes a function, uploads it to the serverless platform and registers for an event (e.g., HTTP request) to trigger function execution. Henceforth, we use event and request interchangeably. When a request arrives, it may lead to a sandbox being setup (known as "cold start" which involves launching a new container, setting up the runtime environment, and deploying the function by downloading code from the datastore) on the cluster machines and then running the function; alternatively, the request may be sent to an existing "warmed up" sandbox as platforms typically do not immediately decommission a sandbox after execution enabling reuse for future executions of the same function [1].

2.2 Characterizing Real World Serverless Apps

We characterize serverless apps by studying the top 50 deployed functions in the AWS Serverless Application Repository (SAR) [9]. SAR consists of diverse functions that run on AWS Lambda [8]. This repository is widely used by the serverless community which is evident from the fact that the top app has been deployed 94K times. Out of the 50 functions studied, 23 are in NodeJS, 26 in Python, and 1 in Java.

Benchmarking Methodology. We upload functions and initiate their execution in the us-east-1 region via the AWS CLI from a VM. We collect the : (1) function *code size*; (2) *sandbox setup overhead* - time taken to setup the function sandbox; (3) *execution time* - time taken to execute the core

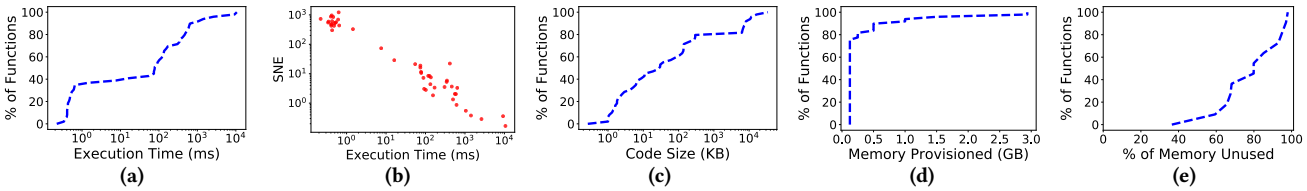


Figure 1: Distribution of (a) execution time, (b) SNE - sandbox setup overhead normalized by execution time, (c) code size, (d) memory provisioned across the 50 functions and (e) unused memory across functions that provision greater than 128 MB memory.

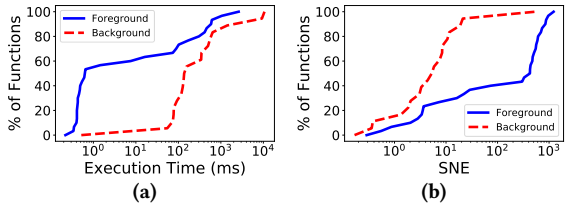


Figure 2: Distribution of (a) execution time and (b) SNE across the foreground and background functions.

function logic (without including sandbox setup overhead); (4) *provisioned memory* - memory available to function as configured by the user; and (5) *runtime memory* - actual memory consumed during execution. We also classify functions as foreground or background based on what they are intended for. We now discuss the key takeaways from our analysis -

[T1] Functions have a wide range of execution times. We find that 57% functions have an execution time <100ms (see Fig. 1a). These typically correspond to user-facing functions. Also, ~10% functions have an execution time >1s. Recent academic works have shown that serverless platforms are attractive for tasks with even longer durations [33, 56]. Fig. 2a shows that the majority (~65%) of the foreground functions have execution times <100ms whereas background ones run longer with <~5% having execution times <100ms.

[T2] Sandbox setup overheads dominate function execution times. We measure the ratio of the sandbox setup overhead to the execution time to investigate the impact of overheads on the end-to-end latencies. We refer to this ratio as SNE (sandbox setup overhead normalized by execution time). Fig. 1b indicates that sandbox setup overheads dominate for >88% functions with the overhead being >100× in 37% of them. Our observations are consistent with prior work [50, 51, 63]. Fig. 2b shows that high sandbox setup overheads impact foreground functions much more severely. These numbers represent the overhead while using microVMs internally used by AWS Lambda [16]. Recently, lighter-weight alternatives have been proposed. However, they offer weak isolation and/or sacrifice code compatibility and thus are not widely adopted [16] (§8).

[T3] Functions have a wide range of code sizes. Setting up sandboxes also involves downloading code from the datastore and loading the runtime, which contribute to the above reported setup overheads. We notice (Fig. 1c) that code sizes can be as large as 34MB. Prior works have shown that these steps can take a significant time (upto 10s of seconds) depending on the code [51].

[T4] Functions typically have small memory footprints. Fig. 1d shows the maximum memory provisioned by the functions. 78% of them require only 128MB. Fig. 1e further shows that most functions requesting more than 128MB of provisioned memory typically leave a significant fraction of provisioned memory unused.

2.3 Serverless Platform Requirements

Based on the above takeaways, the requirements of an ideal serverless platform are as follows:

[R1] Minimize the impact of sandbox setup overheads on end-to-end request latencies. Given that these overheads dominate execution times (T2, T3), we need to eliminate or minimize their impact on the incoming requests.

[R2] Minimize the impact of control plane overheads on end-to-end request latencies. Given that functions with low execution times are the common case (T1), we require the load balancing and scheduling layers to make decisions in sub-millisecond at scale.

[R3] Have a scalable control plane. Given that the platform will service several apps and their request load can grow arbitrarily, we require scalable load balancing and scheduling where neither can become a bottleneck.

Overall Goal. Given that many apps will run simultaneously on the platform, our high-level goal is to support tight performance bounds for the requests. Specifically, per app, we want to maximize the number of requests whose end-to-end latencies are *close* to the native app execution time. We allow developers to define how *close* to native execution they wish to be, by allowing them to specify a deadline. Developers can estimate the app execution time using a few canary requests and specify the deadline based on the requirements of the app (user-facing vs. background).

2.4 Serverless: Current Data + Control Plane Choices

Given the overall goal, we now restate the fundamental tradeoffs and challenges that arise when redesigning data and control planes, and discuss shortcomings of state-of-the-art.

Sandbox Management. To reduce the impact of sandbox setup overheads, platforms do not immediately decommission sandboxes once setup, enabling reuse for future requests

of the same function. However, platforms today [3–5, 8] only *reactively* setup sandboxes [55], i.e., the scheduler waits for a request to arrive and only then sets up a sandbox (if existing ones are busy) leading to requests experiencing a *cold start* - additional latency due to sandbox setup. Additionally, while sandbox reuse amortizes the overhead across numerous requests, platforms today enable reuse by adopting a *static* and *workload-unaware* policy - a sandbox is kept loaded in memory for a fixed time after a function execution. While this policy is simple to implement, it does not take into account workload characteristics and can thus lead to wasteful memory consumption (e.g., when sandboxes are loaded even when the workload does not require them), or additional overheads (e.g, too few sandboxes available and workload increases). There have been recent efforts [17, 20, 51] that reduce sandbox setup overheads. However, such improvements typically come at the cost of isolation and/or code compatibility which precludes their widespread adoption [16].

Scheduling Architectures. While cloud providers do not reveal the architecture adopted by their serverless offerings, we explore the design choices of scheduler architectures and find existing alternatives unsuitable for serverless platforms given the workload requirements. State-of-the-art centralized schedulers [11, 30, 60] can make optimal scheduling decisions but cannot scale to handle the low latency and high requests-per-second throughput requirements, nor are they designed to offer good performance under rapidly-changing request arrival patterns. On the other hand, state-of-the-art decentralized approaches (e.g., Sparrow [52] or Ray [49]), where multiple schedulers without a global view carry out scheduling (e.g., by randomly probing machines) are more scalable, but in many cases do not find machines that have a sandbox available for reuse leading to additional overheads from setup [35]. We further discuss additional scheduler architectures and their shortcomings in §8.

Scheduling and Load Balancing Policies. Existing platforms today [3, 4, 8] do not take into account execution times of functions and nor do their function upload workflow allows users to specify request priorities. Thus it's unclear if/how they prioritize requests to ensure that the latencies are within the expected bounds. Moreover, recent characterizations of popular serverless platforms reveal that their schedulers adopt a simple first-in-first-out policy [58]. The key challenge here is to design scheduling and load balancing policies in concert such that they are light-weight and ensure requests are executed within their latency bounds.

3 Atoll Key Ideas and Request Control Flow

We now describe the key ideas that form the basis of Atoll, a serverless platform designed to meet specified deadlines for latency-sensitive apps running on a fixed-size cluster.

1. Data plane management via decoupling sandbox allocation from request scheduling: Atoll reduces the cold starts by removing sandbox allocation overhead (§2) from the critical path of request execution by *proactively allocating sandboxes* ahead of time based on the expected future load for a function. This approach is viable since functions typically have small memory footprints (T4 in §2.2). Crucially, Atoll allows the platform admin to *bound* the amount of memory that can be used for proactive allocation, and has appropriate allocation and eviction mechanisms in place to act within that bound. Specifically, Atoll uses a novel *even placement* approach to spread sandboxes across the cluster so as to maximize the probability of future requests benefiting from these provisioned sandboxes and *lazily evicts* sandboxes to deal with temporary load fluctuations (§4.3). This approach of reducing cold starts is orthogonal to the recent efforts that reduce the sandbox allocation overheads (§8).

2. Autonomous schedulers and deadline-aware scheduling: To scale scheduling, we introduce *semi-global schedulers* (SGSs) in the control plane. Each SGS is responsible for exclusively managing a partition of the cluster machines known as its *worker pool*. This ensures that a scheduler does not become a scalability bottleneck and makes optimal decisions within the worker pool. We also use a deadline-aware scheduling strategy (§4.2) that leverages the flexibility of the different slack requirements amongst requests and multiplexes among apps' requests to minimize the deadlines missed (§2).

3. Co-designing the load balancing and scheduling layers: Partitioning the cluster introduces the challenge of determining which apps are assigned to which and how many SGSs. To address this challenge, we adopt a hierarchical control plane design but unlike prior works [64], we co-design the load balancing layer with the scheduling layer so that it has the required visibility to (a) do *sandbox-aware* request routing, i.e., take into account proactively allocated available sandboxes, so that maximum future requests benefit from the allocation and (b) prevent an SGS from becoming hotspots by developing a per-app low-overhead gradual scaling mechanism that uses *queuing delay* as the scaling indicator and allows logically scaling out/in the schedulers associated with an app (§5.2) without unduly impacting request processing.

We now present an end-to-end example that highlights the various features of Atoll.

Initial App Upload. The user develops the function and uploads it to our platform. During the initial upload, as done today, the user also specifies the resource requirements as well as the dependencies between functions of the app via a directed acyclic graph (similar to AWS Step Functions [6]). Crucially, we also require the user to specify the app deadline. This can be derived from the 99p latency that is acceptable

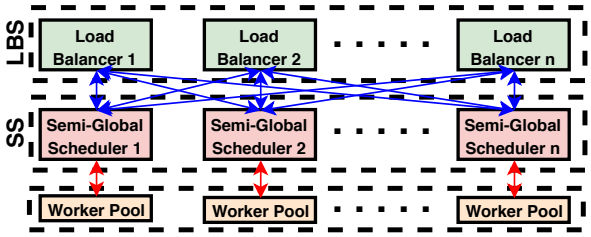


Figure 3: Atoll Architecture. Core services include load balancing service and a scheduling service consisting of semi-global schedulers that manage their own worker pool.

for the app. Using the deadline, Atoll determines the available slack for the requests it services and uses this to ensure minimum deadlines are missed. To prevent developers from always specifying deadlines such that there is zero slack, serverless providers can offer different pricing tiers corresponding to each slack level so as to incentivize developers to specify deadlines that actually reflect app requirements.

Request Control Flow (see Fig. 3). When a request arrives at our platform, it gets routed to one of the many load balancers (LB) that form the load balancing service (LBS). The LB routes it to one of the many SGSs that form the scheduling service (SS) based on its routing policy. At the SGS, the request is enqueued for scheduling. Requests are prioritized by the SGS in a deadline-aware fashion and run on workers available in its worker pool in a work-conserving fashion.

In the background we perform two main actions: first, the SS monitors the incoming traffic to adjust the sandbox allocations and places sandboxes so as to maximize the benefit of proactive allocation. Second, the LBS monitors the load on each SGS and adjusts the routing policy accordingly. We discuss the details of the services in subsequent sections.

4 Scheduling Service (SS)

SS is responsible for managing sandboxes and scheduling incoming app requests. We describe its scalable architecture (§4.1) and then discuss its deadline-aware scheduling strategy (§4.2). Finally, we explain the approach used to proactively allocate sandboxes to reduce cold starts (§4.3).

4.1 Semi-Global Schedulers (SGS)

To cater to the low latency requirements and make optimal scheduling decisions at scale, Atoll divides the cluster into a number of *worker pools*, where each worker pool consists of a subset of machines and is managed exclusively by a semi-global scheduler. The SGSs make up the scheduling service. Each SGS handles a subset of apps. The app to SGSs assignment can change at a coarse-time granularity and is managed by the load balancing service.

Sizing Worker Pools. While deploying Atoll, the platform admin is responsible for determining the worker pool size. The trade-off here is that using too large a worker pool would lead to increased scheduling delays (§2). On the other hand using too small a worker pool could result in load imbalance across various SGS and necessitate frequent load balancing (§7.7). As an extreme, if we choose a worker pool with just a single machine then the load balancer would need to perform scheduling. A simple approach is to organize each rack as a worker pool with one of the machines running the SGS.

4.2 Deadline Aware Scheduling

We now present the scheduling strategy used by an SGS beginning with single function app requests and then generalize it to apps with multiple functions. Given our goal of meeting latency deadlines, we would like to use a policy that minimizes the missed deadlines. Also, given the short executions times, we assume that functions cannot be preempted.

Following classic scheduling approaches to minimize the execution time [28, 53], we propose using the *shortest remaining slack first (SRSF)* algorithm. Whenever a CPU core becomes available, the SGS filters requests to only consider ones whose resource requirements are met by the current available resources and then calculates a *remaining slack* (RS) for the filtered requests. RS here is defined as the remaining time a request can be queued without violating its deadline.

The SGS prioritizes and schedules the request that has the least remaining slack. In case of ties, the SGS picks the request that has the least remaining work. Doing so ensures that we quickly get another opportunity to schedule, which further minimizes deadlines missed. Also, scheduling based on remaining slack avoids starvation for large slack requests.

Multi-Function App. Apps can also consist of multiple functions expressed via a directed acyclic graph (DAG) [33, 56]. We now extend our scheduling strategy to handle such apps. Atoll handles DAGs by calculating the remaining slack for the outstanding functions in the following manner - when a function finishes execution (say the root function), Atoll calculates the slack for the now schedulable functions (functions in the DAG whose dependencies are met) by subtracting the remaining critical path execution time of the app from the time to the app's deadline and places them in the scheduling queue.

4.3 Proactive Sandbox Allocation

Given that serverless workloads have their execution time in the same order of magnitude as that of setting up sandboxes (§2.2) [55], we need to ensure that requests are not exposed to this overhead. To achieve this, Atoll reduces cold starts by decoupling sandbox allocation from scheduling of incoming requests. This allows each SGS to proactively setup sandboxes based on the future expected load which is in contrast to today's platforms [55] that are not workload-aware and reactively setup sandboxes when a request arrives. This ensures that the overhead cost paid for isolation and code compatibility does not get in the way of performance. In contrast, recent proposals to reduce overheads trade-off

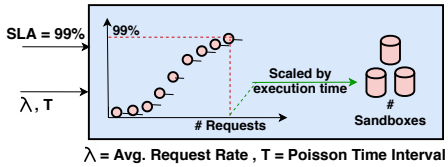


Figure 4: Estimating number of sandboxes to proactively allocate

isolation and/or code compatibility (§8). By decoupling allocation from scheduling, Atoll also enables pipelining of allocation with scheduling, resulting in reduced impacts of cold starts.

Proactively allocated sandboxes occupy memory and do not consume any other resources. With high-memory machines becoming the norm and serverless functions having small memory footprints (§2) [55], we believe it is viable to trade off the memory consumed by the proactively allocated sandboxes to ensure that users are not exposed to sandbox setup overheads. To *limit* the amount of memory used, the platform admin can configure the amount of memory on each machine that can be used to proactively setup sandboxes. We refer to this memory as the *proactive memory pool* from here on. Our evaluations show that Atoll offers benefits over state-of-the-art approaches even under proactive memory pool pressure (§7.5). Finally, we note that proactively allocated sandboxes are a form of *soft state* [25] that can potentially improve performance without affecting correctness.

Each SGS is responsible for proactively setting up sandboxes of functions for which it is receiving requests (as decided by the LBS). In order to do so, the SGS must answer the following questions: (1) how many sandboxes of each function must be setup proactively? (2) how should these sandboxes be placed on its worker pool? (3) when/how should these sandboxes be evicted from the proactive memory pool?

4.3.1 Sandbox Demand Estimation

For each function (part of one or more apps) that is being handled by the SGS, we need to determine the minimum number of sandboxes that need to be allocated to reduce cold starts. To do so, the SGS requires an estimate of the function arrival rate and its request arrival pattern. Atoll's programmable SGS estimator allows the platform admin to use its own logic specifically tailored to its workload (triggered periodically).

Currently, in our prototype we estimate the arrival rate by recording the arrival rate of a function (over a 100ms interval) and using an exponentially weighted moving average over the current interval's measured rate and the previous estimate to get the new estimate. Using this estimated arrival rate, we model request arrivals to follow a Poisson distribution (mimics cloud workloads [36, 48]) and determine the number of requests expected in the Poisson time interval T. Specifically, we use the inverse distribution function to find the *maximum* number of requests that can arrive in T (Fig. 4). However, given that function execution time can be longer than T, we scale up the maximum number of requests to account for requests that overflow from the current time interval to the next one. In our evaluation (§7), we observe that this simple estimator offers reasonable performance in the face of a variety of realistic workload arrival patterns.

4.3.2 Sandbox Placement

Now, given the number of sandboxes that need to be setup for a function, the SGS needs to decide how to place these sandboxes across the various workers in its worker pool. Ideally, we would want to place the sandboxes to maximize the future requests that will use them.

Given recent efforts [51] towards reducing the memory footprint of proactively setting up sandboxes, a tempting approach would be to pack as many sandboxes of the same function on the same worker. While this reduces the memory overhead, it does not increase the probability of future requests benefiting from proactive allocation. For example, consider a scenario where there are two worker machines and the demand estimation of two functions is 2 sandboxes each. Using the above approach, the sandboxes belonging to the same function are setup on the same worker (see Fig. 5b). In such a case, when a core becomes available on worker one and the outstanding request for the second function is to be scheduled, it experiences the overhead of setting up a sandbox as no compatible sandbox is available on the worker.

Instead, in Atoll, for a given function, we *evenly* spread its sandboxes across the various workers (lines 7-19 in Pseudocode 1). Specifically, given the number of sandboxes required, for each sandbox that needs to be setup, the following 2-step process is taken (via the allocator sub-module, Fig. 5a): (1) determine the worker that has the minimum number of sandboxes of this function, and (2) setup sandbox on the worker. This approach improves statistical multiplexing, i.e., makes it easier for future requests to find a proactive sandbox. In Fig. 5b, the request does not incur setting up overhead as a compatible sandbox is already available.

While currently Atoll does sandbox placement on a per-function basis, we leave additional optimizations such as taking DAG dependencies into account as future work.

4.3.3 Sandbox Eviction

The previous section described how an SGS proactively allocates containers based on estimations. However, when the estimators deem that not all the sandboxes previously allocated are required, we need to decide what should be done with these excess sandboxes. A natural approach would be to evict these containers from the underlying worker pool as they consume memory. However, in Atoll we *lazily* evict containers from the worker pool to avoid unnecessary sandbox allocation overheads.

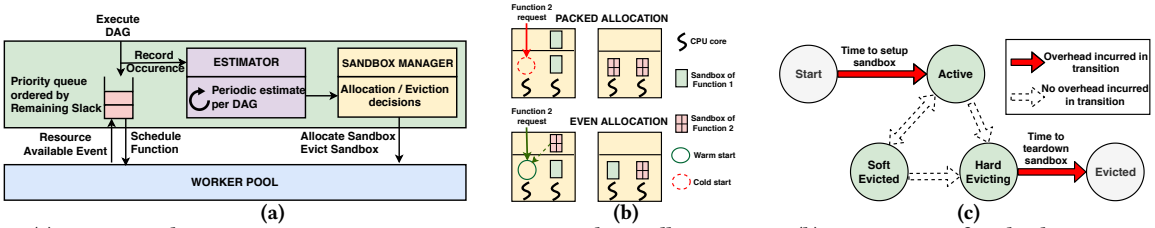


Figure 5: (a) SGS zoomed-in view - a priority queue, an estimator, and a sandbox manager (b) Comparison of multiplexing in packed and even allocation policies. With packed allocation, the execution of function 2 incurs a cold start (marked in dashes) due to the unavailability of a proactively allocated sandbox on that machine. With even allocation, the execution of function 2 does not incur a cold start (marked in solid) since a proactively allocated sandbox is available (c) Sandbox lifecycle in Atoll along with overheads incurred during state transitions.

Pseudocode 1 Atoll Sandbox Management

```
1:  ▷ Given a Function F, either allocate or evict sandboxes
2:  procedure SandboxManagement(Function F)
3:      if F.newDemand > F.oldDemand then
4:          AllocateSandboxes(F, F.newDemand - F.oldDemand)
5:      else if F.newDemand < F.oldDemand then
6:          SoftEvictSandboxes(F, F.oldDemand - F.newDemand)

7:  ▷ Given a function F and its demand, allocate sandboxes
8:  procedure AllocateSandboxes(Function F, Int allocDemand)
9:      for _ in range(allocDemand) do
10:         minW = getWorkerWithMinSandboxes(F.id)
11:         sandboxFound, sandbox = minW.getSoftEvictedContainer(F.id)
12:         if sandboxFound then
13:             minW.SoftAllocate(sandbox)          ▷ Allocate a soft evicted sandbox
14:             continue
15:         if minW.hasEnoughPoolMem(F) then
16:             minW.Allocate(F)                    ▷ Allocate a new sandbox
17:         else
18:             minW.HardEvict(F)           ▷ Hard evict sandboxes to free memory
19:             minW.Allocate(F)

20: ▷ Given function F, evict enough sandboxes to launch a sandbox of F
21: procedure HardEvict(Function F)
22:     while w.freePoolMem < F.memNeeded do
23:         victimF = w.getVictimF()          ▷ Get function based on fairness metric
24:         w.Evict(victimF)                  ▷ w.freePoolMem increases due to eviction
```

In Atoll, a sandbox goes through two eviction stages - *soft eviction* and *hard eviction* (Fig. 5c). When the estimates fall below the previous estimate, the SGS marks the excess sandboxes as soft evicted, i.e., they will not be considered while scheduling requests. Given the excess number of sandboxes of a function that need to be soft evicted, the SGS needs to decide which sandboxes across the various workers need to be soft evicted. For this, the SGS follows a process similar to the placement approach, with the only difference being that it selects the worker(s) that have the maximum sandboxes of this type, and *soft evicts* a sandbox from it. This process repeats until the required number of sandboxes are soft evicted (lines 5-6 Pseudocode 1). This approach balances the sandboxes across workers to the extent possible which improves statistical multiplexing. Having soft evicted sandboxes enables Atoll to deal with temporary load fluctuations. In such scenarios, sandboxes are soft evicted when the load decreases. When the load increases back, soft evicted containers just need to be unmarked and this incurs no overheads.

Finally, a sandbox is *hard evicted* only when the proactive memory pool on a worker is saturated and a new sandbox needs to be setup (lines 20-24 in Pseudocode 1). The SGS hard evicts the sandbox of a function whose current allocation is closest to its estimation. This prevents functions whose allocations are far from their estimation being negatively impacted. The SGS prefers to hard evict a soft evicted sandbox before evicting ones that may be reused for scheduling.

5 Load Balancing Service (LBS)

The LBS is responsible for routing requests to the underlying SGSs. We discuss its responsibilities (§5.1) and how it performs the tasks at hand (§5.2).

5.1 Service Responsibilities

The LBS has two key responsibilities : (1) balance load across SGSs: given that the underlying SGSs partitions the cluster, the LBS should ensure that the load is spread across the various SGSs and a single SGS does not become a bottleneck; (2) perform *sandbox-aware routing*: given that the SGSs proactively allocates sandboxes, the LBS should route requests appropriately with the objective of maximizing the number of requests that benefit from the proactive allocation.

5.2 Scaling SGSs used per App

Given that the cluster is partitioned and is managed by various SGSs, a key question that needs to be answered is among how many SGSs should the requests of an app be spread across? A possible solution would be to use all the SGSs and spread the requests evenly. This would avoid hotspots but naively applying such an approach would lead to degraded performance as more requests would experience the sandbox allocation overhead as each SGS triggers allocations only when it starts receiving requests.

At the other extreme is the option of routing all requests of the app to a single SGS. While this approach does not suffer from the same limitations, a single SGS may not have enough capacity to handle the incoming workload. Thus, we choose a middle ground and dynamically associate the right number of SGSs that are needed to handle an app. However, to ensure that this dynamic approach is effective and performant, the following questions need to be answered - (1) what should

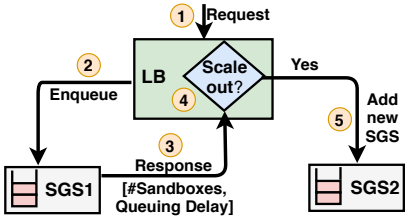


Figure 6: Interaction of load balancer with SGSs during a scale out

be used as the indicator to scale SGSs in and out? (2) what is our scaling mechanism? and (3) how do we ensure that the request latencies do not suffer when we scale out/in?

5.2.1 What is the scaling indicator?

There are numerous situations under which the current SGSs associated with an app could be too few, requiring scale out. First, when the incoming workload of an app cannot be handled by the current SGSs due to resource unavailability. This can happen either due to the incoming load being too high or due to contention with other apps that are handled by the same SGSs. Second, we also need to scale out when there is severe pressure on the cumulative proactive memory pool which can lead to users experiencing sandbox allocation overheads.

Rather than relying on multiple metrics to indicate the above situations, we leverage queuing delay experienced by requests (of the corresponding app) at the SGS as the universal metric. Queuing delay covers the above situations and is easily observable. Each SGS measures the delays per app using EWMA (similar to how it estimates the per app RPS) over a window. The SGS piggybacks this measured queuing delay with each outgoing response to the LBS. The LBS further uses this information to decide if we need to scale out/in.

5.2.2 What is the scaling mechanism?

Initial SGS Selection. When a request for an app arrives for the first time at the LBS, we use consistent hashing [38] to determine which SGS to route requests to. The LBS maintains a consistent hash ring - with all the underlying SGSs hashed to the ring (by using their ID). When the first request arrives, the LBS hashes the app ID to the ring to determine the initial SGS. Using consistent hashing ensures that no single SGS is overwhelmed by being responsible for a large share of apps and enables easy maintenance of the underlying SGSs (an SGS under maintenance is removed from the hash ring and its load is redistributed).

Scale Out (see Fig 6). The LBS receives the queuing delay observed by the requests of this app at the various SGSs. It then computes a scaling metric which is a function of the reported per-SGS queuing delays normalized by the deadline (described below). If the metric is above a *scale-out threshold*, then the LBS scales by associating another SGS (the next one in the ring) with this app (lines 6-7 in Pseudocode 2).

Pseudocode 2 Atoll Per App SGS Scaling

```
1: procedure ScaLing(App A)
2:     N⃗                          ▷ per associated SGS sandbox count for App A
3:     qDelay⃗                     ▷ per associated SGS observed queuing delay for App A
4:     weightedQDelay = Σ_i (N⃗_i * qDelay⃗_i) / Σ_i N⃗_i
5:     scalingMetric = weightedDelay / A.slack
6:     if scalingMetric > ScaleOutThreshold then
7:         ScaleOut(A)
8:     else if scalingMetric < ScaleInThreshold then
9:         ScaleIn(A)
```

Upon scaling, the LBS notifies each of the SGSs associated with this app to reset the queuing delay windows so that it can observe the impact of its decision. The LBS makes the next scaling decision once the windows are filled to avoid reacting to transient changes.

Scale In. The LBS follows a similar process as above to decide if we need to dissociate an SGS from the app, with the difference being that we scale in if the scaling metric falls below the scale-in threshold (lines 8-9 in Pseudocode 2). We remove the SGS that was added last from the pool. To avoid oscillations in the scaling process, we keep the scale-in threshold well below the scale-out threshold.

Scaling Metric (lines 4-5 in Pseudocode 2). Given the per-SGS queuing delay, to calculate the scaling metric, we first compute a weighted sum of queuing delays where we scale per-SGS queuing delay based on the number of proactively allocated sandboxes that exist at the SGS. Next, we normalize this weighted sum by the available slack for the app. Weighing the queuing delays proportional to the number of sandboxes ensures that we give more (less) importance to the SGS that handles more (less) requests of this app as the sandboxes indicate what quantity of requests are handled by an SGS. Normalizing by the available slack makes the scaling deadline-aware as it scales-out more aggressively for latency-sensitive jobs compared to background jobs as the former has less slack and queuing delays can lead to more missed deadlines.

5.2.3 How to do transparent scaling?

When the LBS scales the SGSs associated with an app, we also need to ensure that this does not have a negative impact on the requests. Atoll achieves this by scaling gradually rather than instantly.

When scaling out, we associate an additional SGS with the app. However, instantly sending requests to the new SGS will lead to these requests experiencing sandbox setup overheads. The LBS overcomes this issue by gradually ramping up the new SGS by - (1) using *lottery scheduling* to do sandbox-aware routing among the various SGSs where the number of tickets for each SGS correspond to the number of proactive sandboxes it has setup for this app and (2) notifying the new SGS to proactively allocate the average number of sandboxes

Modes	Description
GFR	Represents state-of-the-art serverless platforms [3, 5, 8]. SGSs have global view (G), schedule requests in FIFO manner (F) [58], allocate sandboxes reactively (R); kept in memory for a fixed 15 mins inactivity timeout [12, 13, 55, 59].
GDR	Replace the FIFO scheduler with deadline-aware scheduler (D) in GFR. Shows the impact of being deadline-aware.
GDPI	Replace reactive allocation with proactive allocation (P) and instant eviction when not required (I) in GDR. Shows the impact of proactive allocation.
Atoll	Replace instant eviction with soft eviction in GDPI. Shows the impact of soft eviction.
D-Atoll	Decentralized version of Atoll. SGS does not have complete view over its worker pool and schedules the incoming request among the two randomly picked workers [52]. Shows the impact of not having a global view.

Table 1: Description of various incremental baselines and Atoll

present across the active SGSs (calculated including the new SGS). We initialize the tickets for the new SGS with a small value (say 1) so that requests go to it and this gets updated as and when sandboxes are setup. Recall that the LBS knows about the number of sandboxes allocated as they are piggy backed on the responses. The system reaches steady-state once the required number of sandboxes have been allocated.

Similarly, we need to scale in gradually as doing so instantly can result in overwhelming the reduced SGSs subset. We do so by maintaining two SGSs lists for an app - an *active list* and a *removed list*. While scaling in, we shift the SGS from the active list to the removed list. During lottery scheduling, we still consider SGSs in the removed list but scale down their lottery tickets by a *discount factor*. This ensures gradual scale in and the reduced subset is not overwhelmed.

6 Implementation

We built our prototype in Go (~15K LOC). All services are implemented as multi-threaded processes. Our LBS has a HTTP front end to receive events that trigger the app execution. The SGS consists of the three loosely coupled modules - scheduler, estimator and sandbox manager. All workers have an execution manager running as a daemon process. This daemon receives scheduling requests from an SGS and places them in the corresponding core queues, and also handles sandbox allocation/eviction requests. Currently, the prototype supports docker containers as well as Go routines as sandbox environments. All communication between the different components happen via protocol buffers [14]. For increased resilience to failure, one may replicate the SGSs and LBS using Apache ZooKeeper [31].

7 Evaluation

7.1 Experimental Setup

Cluster. We evaluate Atoll on a 74 node cluster on CloudLab [23]. All nodes have 256GB memory and 10Gbps NIC. We partition the cluster to have 8 SGSs, each of which has a worker pool of 8 machines, 1 load balancer (uses the logic

Class	Exec. Time (ms)	Slack (ms)
C1	[10-20]	[30-40]
C2	[300-500]	[200-300]
C3	[100-200]	[800-1000]
C4	[100-1500]	[900-4000]

Table 2: [Workloads 1-2] Class-wise execution times and slack

described in §5; unless specified otherwise) and 1 workload generator. We set the scale out threshold SOT= 0.3 (§7.7).

Baselines. We compare Atoll against several baselines (Table 1) with GFR (**G**lobal view, **F**IFO scheduling and **R**eactive sandbox allocation) representing the existing state-of-the-art serverless platforms [3, 5, 8]. We also include incremental baselines that enable us to see the benefits of each aspect of Atoll. To do a fair quantitative comparison, we implement and evaluate various baselines in the same prototype and cluster setup described above. This ensures that all baselines run in the same setting (e.g., using the same network and number of physical machines) and have the same entities in the request serving path. On the other hand, directly using public serverless platforms (e.g., AWS Lambda) leads to challenges in determining the exact number of physical machines used and in breaking down end-to-end latencies (load balancing vs. scheduling vs. other unknown intermediate component overheads). Further, open source serverless frameworks like OpenWhisk [5] involve additional entities (e.g., Kafka [2] and CouchDB [15]) in the critical path making it challenging to perform a fair comparison.

Workloads. We evaluate Atoll using a publicly available trace-driven workload from Microsoft [55] and two synthetic workloads that keep the cluster CPU load between workloads ~70% to 110%. For the two synthetic workloads, we consider four different classes of apps (derived from our AWS SAR study): **(i) C1** consists of single function apps that have very short execution times and tight deadlines. These apps represent user-facing functions. **(ii) C2** consists of apps that have medium execution times and relatively strict deadlines compared to their execution times. These apps represent more expensive user-facing functions. **(iii) C3** consists of apps that have short execution times, and less strict deadlines. These apps represent non-critical user-facing functions (such as updating a metrics dashboard). **(iv) C4** consists of *multi-function* diamond structured apps (similar to [33]) along with single-function apps, high execution times and loose deadlines. These apps represent background jobs. We randomly sample execution times and slack from the ranges in Table 2.

In **workload 1**, for each app we model several request arrival patterns randomly picked from a class of representative serverless arrival patterns such as on/off (on/off duration picked from Poisson distributions), constant and sine patterns [55] (Table 3). In **workload 2**, apart from the patterns present in workload 1 we also introduce Poisson arrivals as they represent human generated events [55] (Table 3). Apart

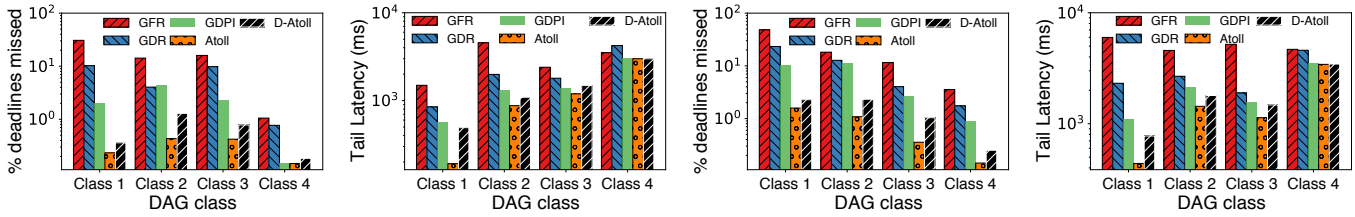


Figure 7: Atoll vs. Baselines. Workload 1 - (a) Deadlines Missed, (b) Tail Latencies; Workload 2 - (c) Deadlines Missed, (d) Tail Latencies (99.9p)

from these two workloads, for **workload 3** we use a publicly available trace of serverless workloads [55]. We pick a couple of popular functions for each of the ten event types and scale the arrival patterns according to our testbed. The function execution times are in the range of 18-124 ms and we set the slack to be 15-25% of execution time as platforms today do not have a notion of deadlines. Given that the trace does not report the allocation overheads, we assign the functions randomly sampled allocation overhead of the latency-sensitive functions (class C1) from our AWS SAR study. For all our experiments, Atoll does not assume that the arrival patterns and rates are known apriori.

Metrics. We use the following metrics to evaluate Atoll - (i) **%deadlines missed** - %requests that do not complete within their deadline; **(ii) number of cold starts** - number of requests that experience sandbox allocation overhead and **(iii) end-to-end latencies** - turn around of a request.

7.2 Macrobenchmarks

Fig. 7a-7b show the deadlines missed and tail latencies for the various baselines and Atoll for workload 1. GFR has the most deadlines missed (66×) and has overall the highest latency (3×) over Atoll as it is deadline-unaware and sets up sandboxes reactively.

Using a deadline-aware scheduler (GDR) vs FIFO leads to 2.73× less missed deadlines. This is due to the scheduler prioritizing latency sensitive requests. However, there is only a slight reduction in cold starts (1.03×) as sandboxes are still being set up reactively. On switching to GDPI, we see that it further leads to 2.99× fewer missed deadlines over GDR due to proactive sandbox allocation which leads to 6.82× fewer cold starts (also 1.45× better tail latencies) over GDR.

Finally, we replace instant eviction with soft eviction which represents Atoll. This leads to a significant reduction in missed deadlines by 8.14× due to 1.29× lesser cold starts over GDPI. The benefits observed are due to Atoll not reacting to transient load changes and as doing so leads to setup overheads once the transient passes. **We observe that the various aspects of Atoll contribute to its significant benefits (**< 1% **missed deadlines) over the state-of-the-art with** 9× **fewer cold starts,** 3× **better tail latencies and** 66× **fewer deadlines missed.** Additionally, we also compare Atoll with a decentralized version of itself - we see that

Pattern	Parameters
Poisson	RPS=[300-1000]
On-Off	Requests=[200-800], Period=[5-20]s
Constant	RPS=[100-200]
Sinusoidal	Avg. RPS=[300-1000], Amplitude=[100-500], Period=[10-30]s

Table 3: [Workloads 1-2] Arrival pattern parameters. We randomly sample multiple patterns for each class

decentralized scheduling leads to more cold starts which results in 1.26× increase in tail latencies and thus 2.14× increase in deadlines missed due to not having a global view.

We observe similar trends (< 1% missed deadlines) for **workload 2 (Fig. 7c-7d) - Atoll leads to** 10.17× **fewer cold starts,** 4.18× **better tail latencies and** 26.39× **fewer missed deadlines over GFR.**

Proactive Sandbox Allocation Analysis. We observe that across both the workloads, the **estimator in Atoll is able to closely estimate the ideal number of sandboxes** that need to be set up for the diverse arrival patterns. Specifically, we observe that in the worst case Atoll allocates ~22% and ~18% more sandboxes for workload 1 and 2 respectively. This is because the SGS provisions for the worst case load to ensure requests do not incur cold starts (§4.3.1). Also, there are times when an SGS allocates sandboxes anticipating future requests, but then the app scales out to another SGS due to contention at the prior one. However, this is not a concern since Atoll uses an isolated memory pool for proactive sandboxes along with a workload-aware eviction policy. Moreover, over-provisioning to this extent is acceptable as it in line with what providers tolerate today [55]. We further evaluate Atoll's performance under memory pressure in §7.5.

Lastly, in the context of the baselines that reactively allocate sandboxes, we see that classes C1-C3 that have relatively less slack tend to miss more deadlines. Our analysis reveal that this is due allocation being coupled to scheduling, i.e., sandboxes can be allocated only when a request is scheduled, which leads to incoming requests to be queued. Atoll mitigates this by proactively allocating sandboxes off the critical path which enables pipelining of allocation and scheduling. This leads to ~13.7× better latencies for such classes.

Workload 3 (Fig. 11a-11b). We see that Atoll offers significant benefits with real world low-latency functions with tight deadlines. Over GFR, we observe that **Atoll leads to**

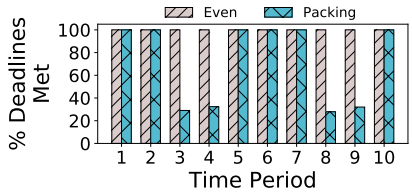


Figure 8: Sandbox Placement - Even Vs. Packing

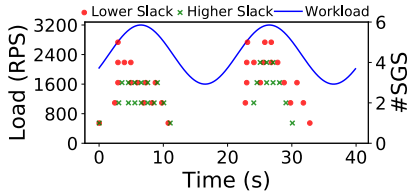


Figure 9: A app with lower slack scales-outmore than a app with higher slack

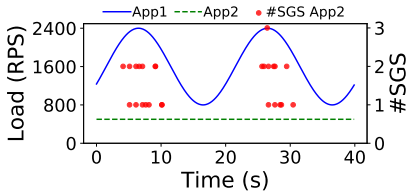
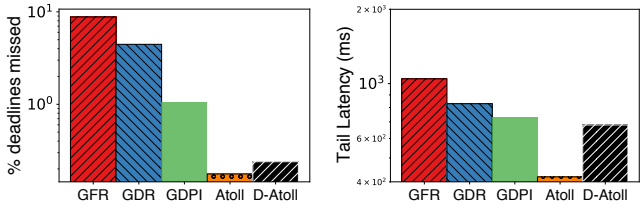


Figure 10: Contention from a bursty app (App1) causes App2 to scale-out



(a) Deadlines Missed **(b) Tail Latencies (99.9p)**

Figure 11: Workload 3 - Atoll vs. Baselines

13.26× **fewer cold starts,** 2.5× **better tail latencies and** 49.91× **fewer missed deadlines due to its various design decisions.** We observe that moving to (a) GDR from GFR leads to 1.97× fewer deadlines missed due to deadline-aware scheduling, (b) GDPI from GDR leads to 5.94× fewer cold starts due to proactive sandbox allocation and (c) soft eviction from instant eviction leads to 5.91× fewer deadlines being missed.

In summary, we notice that the various design choices in Atoll help in reducing the deadlines missed and improving tail latencies, with proactive sandbox allocation and soft evicition resulting in the most gains.

7.3 Microbenchmarks

To delve deeper into benefits of Atoll, we also run small-scale microbenchmarks with 1 LB, one or more SGSs having 10 workers each using simple synthetic workload arrivals to ease explanation of behavior. We find that the behavior holds true for other arrivals as well.

7.3.1 SGS Sandbox Management

We study the effectiveness of the sandbox placement and eviction against alternative strategies using one SGS.

Evenly spreading sandboxes. We compare our approach to when the SGS packs sandboxes on the same worker (to the extent possible) using a single app sinusoidal workload with 1200 avg. RPS, 600 RPS amplitude and 20s period. Both the approaches allocate the same number of sandboxes as they use the same workload. However, we observe (Fig. 8) that packing leads to ~70% deadlines missed during intervals of increased load (3-4, 8-9). This is due to sandboxes being available on a smaller fraction of workers, and at increased load, requests get scheduled on workers that do not have sandboxes available. In contrast, even placement offers better statistical multiplexing resulting in better handling of bursts.

Workload-aware hard eviction. We compare our fair eviction approach with LRU (§4.3.3) using a workload that has two apps: one that has a constant 200 RPS, and another that has an on/off pattern with 100 RPS. The proactive pool is configured low so that it causes hard eviction. We observe that tail latency using LRU is 4.62× worse than fair eviction. This is due to LRU optimizing for the short-term without taking into account future demand, which Atoll does. We observe that during the off-period, it causes *all* of its sandboxes to be hard evicted leading to overheads during the next on-period.

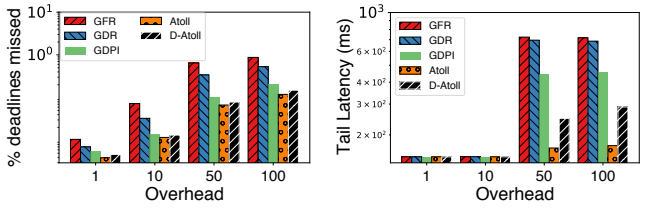
7.3.2 LBS Scaling Strategy

We now evaluate various aspects of the LBS scaling strategy using 5 SGSs.

Gradual Per-App Scaling. We compare our gradual scale-out approach using lottery scheduling (§5.2.3) with an instant scale-out alternative in which the LBS routes requests in a round-robin manner among the SGSs. Using a single-app sinusoidal workload with 800 avg. RPS, 600 RPS amplitude and 100s period (elongated period to capture snapshot of gradual scaling benefits), we observe instant scale-out having 1.5× higher tail latencies over Atoll. This is due to the LBS immediately routing requests to the newly added SGS of an app without taking into account # of available sandboxes.

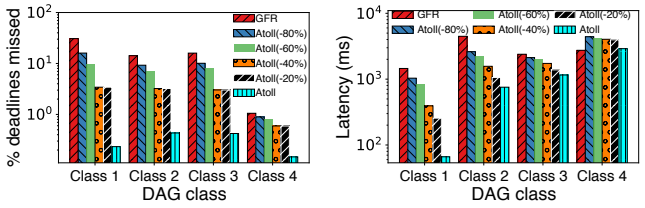
Deadline-aware Per-App Scaling. To study the impact of Atoll's scaling metric being deadline aware, we consider a two app workload in which both follow the same sinusoidal workload (see Fig. 9) and have 100ms execution time but differ in their slacks - one has 50ms while the other has 200ms. We observe (Fig 9) that the smaller slack app scales to more SGSs relative to the other app (e.g., in the 20-30s interval, smaller slack app scales to 4 while the other one scales to 3). This shows the benefits of being deadline aware - the LBS scales up latency sensitive apps more aggressively over background apps that can tolerate delayed responses.

Contention-aware Per-App Scaling. Given that multiple apps are handled by an SGS, we need to ensure that an app does not suffer due to the increased load of another app. To evaluate this, we consider a two app workload - one that has a bursty sinusoidal distribution and the other one having a low, constant rate. The request rate of the second app is set such that it requires only one SGS if it were the only app (see Fig 10). We observe that the LBS is able to scale out the low



(a) Deadlines Missed (b) Tail Latencies (99.9p)

Figure 12: Sandbox Allocation Overhead Impact - Atoll vs. Baselines



(a) Deadlines Missed (b) Tail Latencies (99.9p)

Figure 13: Atoll under Proactive Memory Pool Pressure. The 'Atoll(-X%)" label represents that the cluster was configured with X% lesser proactive memory pool than the incoming workload demand.

rate app to another SGS (e.g., at ∼ 5s) due to contention at the initial SGS and then it scales in once contention reduces (e.g., at ∼ 17s). Atoll is able to act in this manner due to the codesigning of LBS and SS which gives the former visibility into contention at each SGS and scale appropriately.

7.4 Sandbox Allocation Overhead Impact

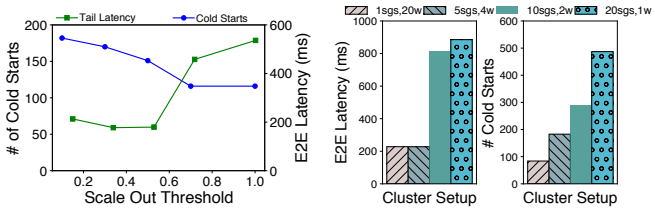
Given the efforts [17, 20, 22, 51] to reduce allocation overheads, we now evaluate Atoll's performance under different overheads - 100ms, 50ms, 10ms and 1ms (emulated using go-routines). Expectedly, we observe (Fig. 12a) that the missed deadlines relative to the GFR reduces from 56.62× to 9.64× to 6.20× to 2.70× as the overhead changes from 100ms to eventually 1ms. This is because lesser overheads leads to more requests in the workload being tolerable to allocation in the critical path as their slack is more than the overhead.

However, workloads with tight slacks (e.g., [42]) would still benefit from Atoll's approach of removing allocation from the critical path. Under allocation overheads of 50ms and 100ms, we observe (Fig. 12b) similar trends as before (§7.2) for tail latencies. However, at the lower end of the spectrum, the latencies remain more or less the same across modes. On further analysis, we observe that this is due to queuing delay at the SGS being the major contributing factor (due to cluster oversubscriptions) to tail latencies and not allocation overheads. These results show that **Atoll's design decisions have an impact across the entire spectrum of sandbox overheads**.

7.5 Atoll Under Proactive Memory Pool Pressure

We now evaluate the performance of Atoll when the proactive memory pool cannot hold adequate sandboxes for all functions during peak load. To do so, we evaluate using workload 1, Atoll's performance under different extents of memory pressure by configuring the proactive pool to be 20%, 40%, 60% and 80% of the peak memory requirement.

We observe (Fig. 13a) that the missed deadlines increases as the extent of under-provisioning increases due to more requests experiencing cold starts. Crucially, even when severely underprovisioned by 80% (unlikely in practice), Atoll misses 1.8× fewer deadlines than GFR. We observe (Fig. 13b) a similar trend for tail latencies as more requests experience cold starts. Interestingly, we observe that class C4, which



(a) (b)

Figure 14: (a) Scale Out Threshold Vs. Cold Starts and Latencies; and (b) SGS Sizing Vs. Cold Starts and Latencies

consists of apps having loose deadlines, has higher tail latencies than GFR under memory pressure. Our analysis reveals that this is because Atoll's deadline-aware scheduling leads to requests belonging to class C4 altruistically yielding to latency sensitive requests belonging to classes C1-C3, while minimizing the number of deadlines missed for class C4.

7.6 System Overheads

We notice that LBS request routing takes 100μs (121μs) and SGS scheduling adds another 241μs (280μs) at the median (99%-ile). We also measure time taken to make a scaling decision at the LBS and estimation at the SGS (these happen off the critical path). Scaling and estimation take 128μs (197μs) and 879μs (1352μs) at the median (99%-ile).

7.7 Sensitivity Analysis

Scale Out Threshold (SOT) (Fig 14a). We observe that there is a trade-off between managing queuing delays and cold starts while choosing the appropriate scale out threshold value.

Setting values either too low or too high has negative impact on tail latencies - (i) at lower values the LBS scales out more frequently leading to higher cold starts and negatively impacting tail latencies; and (ii) at higher values the LBS passively scales out leading to higher queuing delays at the SGS. Based on our observed values, we set the scale out threshold to 0.3 for our experiments.

SGS Size (Fig. 14b). To study the sizing impact we consider 4 ways in which 20 workers can be partitioned - (i) 20 SGSs, 1 worker each; (ii) 10 SGSs, 2 workers each; (iii) 5 SGSs, 4 workers each and (iv) 1 SGS, 20 workers each. We use a single app sinusoidal workload with 600 avg. RPS, 400 RPS amplitude and 20s period.

We observe that fine-grained partitioning leads to ~4× higher tail latencies as the LBS scales-out more often leading to higher cold starts. However, too many workers (beyond 64 workers in our testbed) under SGS can lead to scheduling overhead becoming a significant contributor to queuing delay and this would lead to LBS unnecessarily scaling out even when workers under initial SGS are underutilized.

8 Other Related Work

Serverless Characterization. Many previous papers reverse engineer aspects of serverless offerings by observing the visible metrics [24, 39, 44, 46, 57, 58, 63, 65]. Tariq et al. investigate the QoS offered by public serverless platforms [58]. Shahrad et al. characterize workloads from Azure Functions [55]. Complimentary to, these efforts, we look at popular apps found in AWS SAR [9].

Sandbox Overhead Reduction. SAND [17] and SOCK [51] leverage process-fork based techniques to reduce overheads. Alto [41] and Chromium [10] aim to use a process per trust domain. Boucher et al. advocate for language-based isolation instead of traditional virtualization [18]. SEUSS [20] and LightVM [45] propose using unikernels to reduce overheads. REAP [59] proactively fetches the pages corresponding to a function's working set from disk to further reduce overheads of function snapshotting proposals [22]. Nightcore [32] reduces overheads by trading off isolation as it proposes running multiple invocations of the same function in the same container and optimizes inter-function IO. Our work on reducing the number of cold starts is orthogonal to these improvements. Moreover, it remains to be seen if these efforts see practical adoption as they typically trade-off isolation and/or compatibility for performance [16]. Shahrad et al. make similar observations about today's sandbox management and proposes setting keep-alive windows in a workload-aware manner [55]. Likewise, inspired by traditional caching literature, FaasCache [26] comes up with a workload-aware keep-alive policy that takes into account additional function characteristics such as function size and initialization costs. In contrast, Atoll addresses the issue by proactively launching sandboxes based on the workload and can handle transient load changes better through soft eviction. FaasNet [62] quantifies the overheads involved in pulling sandbox images from the backend datastore and focuses on reducing the overheads involved through a on-demand decentralized fetching mechanism, and is complimentary to Atoll. Ignite [21] reduces overheads due to executing unoptimized code in serverless functions through sharing runtime compiler JIT optimizations across active function sandboxes in the cluster and can be used in concert with Atoll.

Scheduling Architectures. Swayam's [27] architecture is similar to the distributed schedulers that we discussed earlier [52] and thus experiences overheads due to not having complete visibility. Borg [61] uses random sampling while calculating scores and thus trades off scheduling optimality for scalability. Omega [54] uses multiple parallel schedulers but trades off on scheduling predictability for scalability due to the overheads involved in resolving conflicts which would happen often in our setting due to resources being held for short durations.

While Apollo [19] tries to reduce the frequency of conflicts by collecting cluster load periodically and feeding this to individual job schedulers, it does not allow for diverse applications to share the cluster as it makes the assumption that there are either latency sensitive tasks with guarantees or opportunistic tasks with no guarantees. In Atoll, we can accommodate various kinds of tasks and minimize deadlines missed. Mercury [37] is a hybrid scheduler that makes high-quality assignment for long tasks but the short tasks are scheduled in a distributed manner and can be preempted anytime leading to their sub-optimal placement. Pigeon [64] is a two-layer scheduler that partitions workers to handle short and long tasks but the tasks are distributed across the schedulers without taking into account their current state which leads to increased overhead (e.g., when sandbox not available). In Atoll, we avoid such overheads by performing sandbox-aware routing.

Kaffes et al. make similar observations as to why existing scheduling architectures fall short [35]. Their design proposes a core-granular scheduler to improve function placement/isolation but does not address cold start overheads.

9 Conclusion

We consider the problem of ensuring low latency function execution in serverless settings, an important problem that has not received attention. Our system, Atoll, meets this goal using a combination of simple but effective, scalable techniques - (a) partitioning the cluster into (semi-global scheduler, worker pool) pairs, (b) performing deadline-aware scheduling and proactive sandbox allocation, and (c) sandbox-aware routing with automatic scaling. Our evaluation shows that Atoll reduces the number of deadlines missed by upto ~66× and tail latencies by ~3× of realistic workloads compared to state-of-the-art alternatives.

Acknowledgments

We would like to thank our shepherd, Margo Seltzer, the anonymous reviewers of SoCC'21 and the members of WISR Lab for their insightful comments and suggestions. This research was supported by NSF Grants CNS-1565277, CNS-1719336, CNS-1763810, CNS-1838733, by gifts from Google and VMware, a Facebook faculty research award, and by the Office of the Vice Chancellor for Research and Graduate Education at University of Wisconsin-Madison with funding from the Wisconsin Alumni Research Foundation.

References

[1] 2014. Tim Wagner. Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/.

[2] 2017. Apache Kafka. https://kafka.apache.org/.

[3] 2017. Azure Functions. https://functions.azure.com.

[4] 2017. Google Cloud Functions. https://cloud.google.com/functions.

[5] 2017. IBM Bluemix Openwhisk. https://www.ibm.com/cloud-computing/bluemix/openwhisk.

[6] 2018. AWS Step Functions. https://aws.amazon.com/step-functions/.

[7] 2019. Amazon Web Services. https://aws.amazon.com/.

[8] 2019. AWS Lambda. https://aws.amazon.com/lambda/.

[9] 2019. AWS Serverless Application Repository. https://aws.amazon.com/serverless/serverlessrepo/.

[10] 2019. Chromium Projects. https://www.chromium.org/developers/design-documents/site-isolation.

[11] 2019. Google Container Engine. http://kubernetes.io.

[12] 2019. Mikhail Shilkov. AWS Lambda Cold Starts. https://mikhail.io/serverless/coldstarts/aws/.

[13] 2019. Mikhail Shilkov. Azure Functions Cold Start. https://mikhail.io/serverless/coldstarts/azure/.

[14] 2019. Protocol Buffers. https://bit.ly/1mISy49.

[15] 2021. Apache CouchDB. https://couchdb.apache.org/.

[16] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 419–434.

[17] 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)*. 923–935.

[18] 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)*. 645–650.

[19] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI*.

[20] 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*. 1–15.

[21] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. 2021. From Warm to Hot Starts: Leveraging Runtimes for the Serverless Era. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 58–64. https://doi.org/10.1145/3458336.3465305

[22] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. 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*. 467–481.

[23] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[24] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, and Maciej Malawski. 2018. Performance Evaluation of Heterogeneous Cloud Functions. *Concurrency and Computation: Practice and Experience* 30, 23 (2018), e4792.

[25] Armando Fox, Steven D Gribble, Yatin Chawathe, Eric A Brewer, and Paul Gauthier. 1997. Cluster-based scalable network services. In *ACM SIGOPS operating systems review*, Vol. 31. ACM, 78–91.

[26] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 386–400. https://doi.org/10.1145/3445814.3446757

[27] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. 2017. Swayam: distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 109–120.

[28] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. 2003. Size-based scheduling to improve web performance. *ACM Trans. Comput. Syst.* 21 (2003), 207–233.

[29] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651* (2018).

[30] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R. Katz, S. Shenker, and I. Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*.

[31] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 11.

[32] Zhipeng Jia and Emmett Witchel. 2021. 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* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701

[33] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoice, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *SOCC*.

[34] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. arXiv:1902.03383 [cs.OS]

[35] 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*. 158–164.

[36] 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*. 1–16.

[37] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *USENIX ATC*.

[38] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. 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*.

[39] 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)*. 789–794.

[40] 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)*. 427–444.

[41] James Larisch, James Mickens, and Eddie Kohler. 2018. Alto: lightweight vms using virtualization-aware managed runtimes. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. 1–7.

[42] Collin Lee and John Ousterhout. 2019. Granular Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 149–154.

[43] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 363–378. https://www.usenix.org/conference/atc19/presentation/liu-ming

[44] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 159–169.

[45] 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*. 218–233.

[46] Garrett McGrath and Paul R Brenner. 2017. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 405–410.

[47] Garrett McGrath, Jared Short, Stephen Ennis, Brenden Judson, and Paul Brenner. 2016. Cloud event programming paradigms: Applications and Analysis. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 400–406.

[48] David Meisner and Thomas F Wenisch. 2012. DreamWeaver: architectural support for deep sleep. *ACM SIGPLAN Notices* 47, 4 (2012), 313–324.

[49] 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

[50] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Pipsqueak: Lean lambdas with large libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 395–400.

[51] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *ATC 18*.

[52] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, low latency scheduling. In *SOSP*.

[53] Linus Schrage. 1968. A Proof of the Optimality of the Shortest Remaining Processing Time Discipline. *Operations Research* 16, 3 (1968), 687–690.

[54] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, scalable schedulers for large compute clusters. In *EuroSys*.

[55] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar 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 *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association.

[56] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless Linear Algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 281–295.

[57] Arjun Singhvi, Sujata Banerjee, Yotam Harchol, Aditya Akella, Mark Peek, and Pontus Rydin. 2017. Granular computing and network intensive applications: Friends or foes?. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM, 157–163.

[58] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 311–327.

[59] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 559–572. https://doi.org/10.1145/3445814.3446714

[60] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*.

[61] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *EuroSys*.

[62] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 443–457. https://www.usenix.org/conference/atc21/presentation/wang-ao

[63] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *ATC 18*.

[64] 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*. 246–258.

[65] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with Serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 30–44. https://doi.org/10.1145/3419111.3421280