Archipelago: A Scalable Low-Latency Serverless Platform

Arjun Singhvi
University of Wisconsin-Madison

Kevin Houck
University of Wisconsin-Madison

Arjun Balasubramanian
University of Wisconsin-Madison

Mohammed Danish Shaikh
University of Wisconsin-Madison

Shivaram Venkataraman
University of Wisconsin-Madison

Aditya Akella
University of Wisconsin-Madison

Abstract

The increased use of micro-services to build web applications has spurred the rapid growth of Function-as-a-Service (FaaS) or serverless computing platforms. While FaaS simplifies provisioning and scaling for application developers, it introduces new challenges in resource management that need to be handled by the cloud provider. Our analysis of popular serverless workloads indicates that schedulers need to handle functions that are very short-lived, have unpredictable arrival patterns, and require expensive setup of sandboxes. The challenge of running a large number of such functions in a multi-tenant cluster makes existing scheduling frameworks unsuitable.

We present Archipelago, a platform that enables low latency request execution in a multi-tenant serverless setting. Archipelago views each application as a DAG of functions, and every DAG in associated with a latency deadline. Archipelago achieves its per-DAG request latency goals by: (1) partitioning a given cluster into a number of smaller worker pools, and associating each pool with a semi-global scheduler (SGS), (2) using a latency-aware scheduler within each SGS along with proactive sandbox allocation to reduce overheads, and (3) using a load balancing layer to route requests for different DAGs to the appropriate SGS, and automatically scale the number of SGSs per DAG. Our testbed results show that Archipelago meets the latency deadline for more than 99% of realistic application request workloads, and reduces tail latencies by up to ~36*X* compared to state-of-the-art serverless platforms.

1 Introduction

Recent trends in cloud computing point towards increased adoption of micro-services to design and deploy online applications [29]. These micro-services are typically designed to compute a single function with the goal that each microservice can be independently deployed and managed in a cluster, and collectively the microservices implement what used to be realized as large monolithic applications. To meet this demand imposed by independently scalable functions, simplify programming, and relieve programmers from provisioning and elastic scaling responsibilities, cloud computing providers now offer Function-as-a-Service (FaaS) or *serverless* computing [1, 7, 13] offerings, such as, AWS Lambda, Azure Functions, Google Cloud Functions etc.

While serverless computing simplifies a number of aspects of designing and deploying microservice workloads in the cloud, it introduces a number of new challenges with respect to resource management and scheduling for the cloud provider. The specific workload properties that make scheduling challenging, especially in a multi-tenant setting that supports microservices from different applications, include: (i) function execution times are typically short-lived with 90% of functions executing for less than a second, but a few functions execute for 10s of seconds (§2); (ii) as functions are expected to be isolated, they often require setting up appropriate computational units, or "sandboxes", but these sandboxes can be reused to serve future function requests; and, (iii) the arrival patterns of application requests as a whole, and for microservices or functions therein, can vary substantially making it necessary for the scheduler to handle large dynamic variations in the workload.

Existing architectures for scheduling and resource allocation in large clusters are unable to handle the above requirements. Centralized schedulers [14, 27, 47] 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, decentralized approaches (e.g., Sparrow [41] or Ray [38])), where multiple schedulers with a global view carry out scheduling (e.g., by randomly probing machines) are more scalable, but may not find machines that have a sandbox available for reuse leading to additional overheads from sandbox setup. Finally, existing frameworks do not account for the execution time of individual functions and thus are unable to appropriately prioritize DAG requests to ensure that the end-to-end latencies, which may include sandbox provisioning and setup, are as close as possible to the execution time for a vast majority of incoming application requests.

We present Archipelago, a scheduling framework that supports low overhead function execution, and enables tight latencies for application request completions in a multi-tenant serverless setting. Archipelago views each application as a DAG, where nodes are microservices or functions, and edges are I/O dependencies, and allows the programmer to associate a deadline with the DAG. As requests arrive at variables rates for different DAGs, Archipelago schedules the execution of the constituent functions on a given cluster of resources such that a vast majority of incoming requests meet their deadline.

Archipelago achieves the above goal via a combination of techniques. First, Archipelago *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 sent for every DAG it is serving, and proactively allocates sandboxes to minimize the overheads in launching DAGs' functions. 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.

Second, Archipelago uses a scheduling algorithm within an SGS that is aware of the latency requirements for each DAG. This enables us to compute a running *slack*, or the time remaining for a given DAGs' request, and use a variant of the shortest-remaining-time-first algorithm to minimize the possibility of DAGs missing their *deadlines*. Here, we leverage the fact that applications running in a cluster have different slacks, and low-slack applications' resource needs can be met by reallocating resources away 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 Archipelago is to use a sandbox-aware load balancing layer that can route requests while being aware of the number of sandboxes of different DAGs allocated in every SGS. In order to simplify the design of the load balancing layer and make it scalable, every application DAG running in the cluster is assigned to a single SGS to begin with and based on the number of requests, the load balancer can either *scale out* (or scale in) the number of SGS assigned to this DAG. Using an approach that is also aware of sandbox allocation ensures that application performance is minimally affected when scaling across the cluster.

We build Archipelago in Go and evaluate our prototype against the current state-of-the-art serverless scheduler using a collection of applications derived from our analysis of real-world serverless workloads. Our results show that Archipelago is able to meet the latency deadline for more than 99% of requests across various application classes, and reduces tail latencies by more than 36×. We find that sandbox-aware load balancing can reduce tail latencies by up to 24.38×, and that Archipelago's sandbox placement policy is crucial to meeting latency deadlines.

2 Background and Motivation

We start by providing a primer on serverless computing. We then characterize the properties of real world serverless applications available on the repository maintained by AWS [6]. Based on our analysis, we state our requirements and end with why current serverless platforms fall short.

2.1 Serverless Computing Background

In serverless computing or FaaS, the programmer develops an application (or simply an "app") as a directed acyclic graph (DAG) of functions, uploads it to the serverless platform (which stores the code in a datastore) and registers for an event (e.g., incoming HTTP requests, object uploads) to trigger its execution. The platform triggers the DAG execution only when the event arrives, and thus the programmers are billed only when the DAG runs and for the cumulative execution times of the constituent functions. Henceforth, we use event and request interchangeably.

Internally, the platform consists of a load balancing layer, a scheduling layer, and cluster machines. When a request arrives at one of the load balancers, it routes the request to one of the many internal schedulers. The scheduler triggers the execution of the root function(s) of the corresponding DAG by setting up sandboxes (involves launching a new container, setting up the runtime environment, and deploying the function by downloading the code from the datastore) on the machines in the cluster and running the function(s).

Alternatively, the function can directly run on a "warmed up" sandbox as platforms typically do not immediately decommission sandboxes enabling reuse for future executions of the same function. On completion, a notification is sent to the scheduler, which then triggers the execution of the downstream functions. The process repeats until DAG completion. Additionally, the platform elastically scales by launching more sandboxes based on incoming events.

2.2 Characterizing Real World Serverless Apps

We characterize serverless workloads by studying the top 50 deployed apps (as of November 1, 2019) in the AWS Serverless Application Repository (SAR) [9].

SAR consists of diverse apps that run on AWS Lambda [7]. Internally, AWS Lambda uses Firecracker microVMs [12] to run the apps. These apps typically interact with other AWS services (e.g., S3 [5]) as well as third-party services (e.g., Slack [4]). This repository is widely used by the serverless community which is evident from the fact that the top app has been deployed 45K times. All 50 apps have a single function, but many recent serverless proposals have rich DAGs of functions [20, 23, 28, 45]. Out of the 50 functions studied, 23 are in NodeJS, 26 in Python, and 1 in Java.

Benchmarking Methodology. We use the AWS CLI to upload and trigger the execution of the functions under study. The functions were triggered to run in the us-east-1 region via a VM running in the same region. We collect the following statistics: (1) function *code size*; (2) *provisioned memory* - memory available to the function during execution as configured by the programmer while uploading the function to the platform ; (3) *runtime memory* - actual memory consumed during function execution; (4) *sandbox setup overhead* - time taken to setup the function sandbox which includes the steps

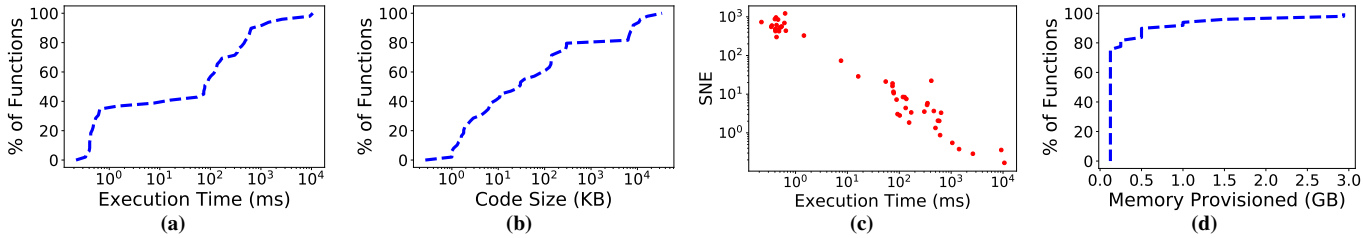
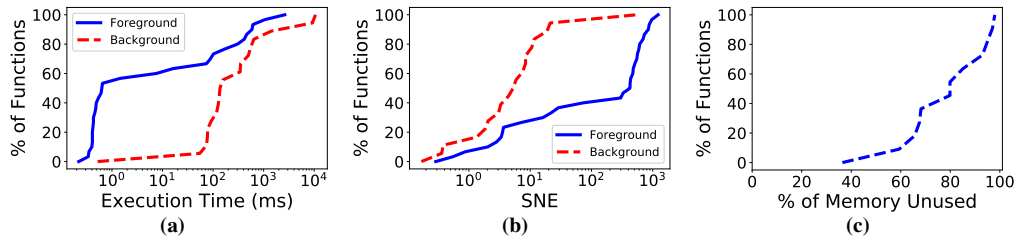


Figure 1. Distribution of (a) execution time, (b) code size, (c) SNE and (d) memory provisioned across the 50 functions



	50%ile	99%ile
FIFO	100ms	101ms
Sparrow	102ms	136ms

(d)

Figure 2. Distribution of (a) execution time, (b) SNE across the foreground and background functions and (c) memory unused across functions that provisioned greater than 128 MB. (d) end-to-end latency comparison between FIFO and Sparrow when the incoming workload leads to a cluster CPU utilization of ~70%.

discussed above; and (5) *execution time* - time taken to execute the core function logic (without including the sandbox setup overhead). Finally, we also classify functions as *foreground* (typically serving user-facing apps) or *background* based on what they are intended for.

We next discuss the key takeaways from our analysis:

[T1] Functions have a wide range of execution times. As seen in Figure 1a, 57% of functions have an execution time of less than 100ms. These typically corresponds to user-facing functions (e.g., ALEXA-SKILLS-KIT-NODEJS-FACTSKILL). Also, ~10% functions have an execution time > 1 second (e.g., NYC-PARKS-EVENTS-CRAWLER takes ~10s). Additionally, recent works in academia have shown that serverless platforms are attractive for embarrassingly parallel tasks that can last for even longer durations [23, 28, 45] (~100s). Fig.2a further shows the split of execution times based on whether they are foreground and background. As expected, we see that majority (~65%) of the foreground functions have execution times < 100ms whereas background functions typically run longer with fewer than ~5% having execution times < 100ms.

[T2] Functions have a wide range of code sizes. Allocating sandboxes also involves downloading the code from the datastore and setting up the runtime. Prior works have shown that these steps can take significant amount of time (upto 10s of seconds) depending on the code [40]. In our analysis (Fig. 1b), we notice that code sizes can be as large as 34MB.

[T3] Sandbox setup overheads dominate execution times. We measure the ratio of the sandbox setup overhead to the execution time of the apps 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. 1c indicates that sandbox setup overheads dominate for > 88% of the functions with the overhead being > 100X in 37% of them. Our observations are consistent with data from prior work [39, 40, 49]. Fig. 2b shows that high sandbox setup overheads impact foreground functions much more severely.

[T4] Functions typically have small memory footprints. Fig. 1d shows the maximum memory provisioned by the functions. 78% of the them require only 128MB. Fig. 2c further shows that most functions requesting more than 128MB of provisioned memory typically leave a significant fraction of provisioned memory unused.

[T5] Majority of apps have a single function. All of the top 50 deployed apps have only a single function. Out all the apps on SAR, we found only two instances of DAGs which were a linear chain of 2 functions (e.g.,CW-LOGS-TO-SLACK). However, as noted earlier, many emerging applications induce richer DAGs. Our work aims for generality, and thus our work also encompasses applications that are DAG-structured, as opposed to focusing on single function ones.

2.3 Serverless Platform Requirements

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

[R1] Minimize the impact of sandbox setup overheads on end-to-end request latencies: Given that these overheads dominate execution times (T3), we wish to eliminate them from end-to-end request execution critical paths.

[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 of the platform to make decisions in sub-millisecond at scale.

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

Overall Goal. Given that many applications may run simultaneously on the platform, our high-level goal is to support tight

performance bounds for application requests. Specifically, we wish to ensure that, per application, end-to-end latencies are "close" to native application execution times for a vast majority of requests We allow developers to define how "close" to native execution they wish to be, by allowing them to specify a deadline.

2.4 Issues with Serverless Platforms Today

Existing platforms and mechanisms cannot meet the above goal due to:

1. Reactive, Fixed, and Workload-Unaware Sandbox Management Policy. Most of today's serverless platforms [1–3, 7] only reactively setup sandboxes, 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 additional latency. Also, given the overheads associated, to amortize the overheads across future requests, platforms adopt a static and workload-unaware policy - a sandbox is kept loaded in memory for a fixed amount of time (since its last invocation). While the above policy is simple to implement, it does not work well in practice as - (a) it does take into account workload characteristics while making decisions which can 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 suddenly); and (b) is easy to game for external users (e.g., frequently send dummy requests to ensure that the sandbox is not evicted [18]).

2. Sub-Optimal Scheduler Architectures. While centralized schedulers can make optimal scheduling decisions, when incoming workload grows arbitrarily, a centralized approach can easily become a scalability bottleneck. Decentralized approaches are promising, but they trade-off scheduling quality or low predictable scheduling latencies for achieving scalability, which lead to higher end-to-end latencies.

For instance, *parallel global scheduling* approaches (e.g., Sparrow [41]), where multiple schedulers with a global view carry out scheduling by randomly probing two machines, may not find the best-fit for the function under load as it randomly probes machines and does not make an optimal scheduling decision (Fig. 2d). Similarly, *bottom-up hierarchical scheduling* (e.g., Ray [38]), where functions are first submitted to a per-node local scheduler and are sent to a randomly chosen global scheduler only when it is not possible to schedule locally (say due to overload), may experience unpredictable scheduling latencies as the function may bounce back and forth between node and global schedulers due to conflicts between multiple global schedulers.

3. Homogeneous Request Handling. In serverless platforms today, every incoming request is handled in the same manner, which limits them from making intelligent scheduling decisions. In practice, functions have varying latency requirements; e.g., foreground functions are typically latency sensitive and can tolerate limited additional delay, whereas background functions normally have higher slack and can tolerate higher delay. And, not all functions with tight latency requirements are likely to impose high load at the same time. An ideal platform can leverage these aspects to carefully multiplex and schedule requests to maximize the number of requests that get their responses before their available slack runs out.

3 Key Ideas and Architecture

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

1. Decoupling sandbox allocation from request scheduling: Archipelago removes 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. Additionally, Archipelago uses a novel *even placement* approach to spread sandboxes across the cluster so as to maximize the probability of future requests benefitting from these provisioned sandboxes (§4.3).

2. Autonomous schedulers and SLA aware scheduling: To scale scheduling, we introduce *semi-global schedulers* (SGSs). 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 ensures that schedulers make optimal decisions within the worker pool. We also develop a *deadline-aware* scheduling strategy (§4.2) that leverages the flexibility of the different slack requirements amongst requests and multiplexing among apps' requests (§2) to ensure that deadlines are met.

3. Co-designing the load balancing and scheduling layers: Partitioning the cluster into a number of SGSs introduces the challenge of determining which DAGs are assigned to which SGS. We use the load balancer to address this challenge and codesign the load balancing and scheduling layers so that the load balancing layer has the required visibility to (a) do *sandbox-aware* request routing and (b) prevent individual SGSs from becoming hotspots. Doing so maximizes future requests that benefit from proactive allocation. Additionally, we develop a low-overhead *gradual scaling* mechanism that allows logically scaling up/down the schedulers associated with a DAG to prevent hotspots (§5.2) without unduly impacting request processing.

We next present an end-to-end example that highlights the various features of Archipelago.

Initial DAG Upload. The user develops the functions that make up the computation DAG and uploads them to our platform. During the initial upload, as done today, the user also specifies the resource requirements of the functions along

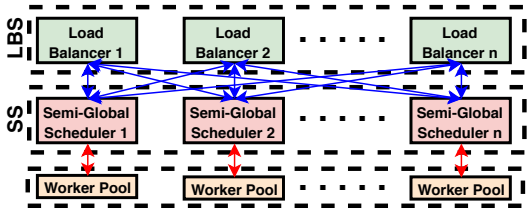


Figure 3. Archipelago Architecture. Core services include load balancing service and a scheduling service consisting of semi-global schedulers that manage their own worker pool.

with the DAG structure using a JSON-based language. Crucially, we also require the user to specify the maximum execution time for the DAG given a new input trigger. This can be derived from the 99% percentile latency that is acceptable for an application. Archipelago aims to maximize the number of requests that are completed within this deadline.

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 available workers in the worker pool in a work-conserving fashion.

In the background we perform two main actions: first, the SS monitors the memory available and 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 each of the above mentioned components in subsequent sections.

4 Scheduling Service (SS)

SS is responsible for managing sandboxes and scheduling incoming DAG requests. We first describe the architecture that makes it scalable (§4.1) and then discuss the deadline-aware strategy used to minimize deadlines missed (§4.2). Finally we explain the approach used to proactively allocate sandboxes so as to minimize the impact of sandbox setup overheads(§4.3).

4.1 Semi-Global Schedulers (SGS)

To handle the low latency requirements and make optimal scheduling decisions, Archipelago divides the cluster into a number of *worker pools*, where each worker pool consists of a subset of machines in the cluster. Every worker pool is then assigned to a semi-global scheduler (SGS) and these semi-global schedulers form a part of the scheduling service.

Given the nature of our workload, where we have a small number of independent, latency-sensitive DAGs , we partition the DAGs such that each SGS is only responsible for a subset of DAGs. This assignment can change at a coarse-time granularity and is managed by the load balancing service.

Sizing Worker Pools. While deploying Archipelago, the platform admin is responsible for determining the size of each worker pool. The trade-off here is that using too large of a worker pool would lead to increased scheduling delays (as discussed in §2). On the other hand using too small a worker pool could result in load imbalance across various SGS and necessitate frequent load balancing (§7.5). As an extreme, if we choose a worker pool with just a single machine then the load balancer would need to perform all the scheduling of requests. A simple approach we espouse is to organize each rack as a worker pool with one of the machines running the SGS.

4.2 Deadline Aware Scheduling

We next present the strategy we use to schedule requests in an SGS, first in the context of individual functions and the generalize it to requests traversing a DAG. Requests are routed to an SGS from the load balancer and incoming requests are placed in a scheduling queue. Given our goal of meeting latency deadlines, we would like to adopt a scheduling policy that minimizes the number of missed deadlines. Additionally, given the short execution times, we assume that functions cannot be pre-emptied during execution.

Following classic scheduling approaches to minimize the execution time [25, 43], 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* for the filtered requests. Slack here is defined as the time a function request can be queued without violating its deadline.

The SGS prioritizes and picks the function request that has the least remaining slack. In case of ties, the SGS picks the function which has the least remaining work. Doing so ensures that we quickly get another opportunity to schedule, which further minimizes deadlines missed. Additionally, scheduling based on remaining slack also avoids starvation for requests with large amount of slack. Finally, the SGS schedules requests on available workers in a work-conserving manner. The SGS spreads out sandboxes for a function across its workers to maximize the chances that a proactively allocated sandbox will be available at the worker (§4.3.2).

DAG Awareness. We now extend the scheduling strategy to handle a DAG. Given the user-specified DAG deadline, the key question that needs to be answered is, how is the remaining slack calculated for a DAG? After a function is processed, the remaining slack for each function of a DAG is calculated by subtracting the critical path execution time [32, 33] from the time remaining to the DAG's deadline. As an SGS is DAG aware, it schedules functions once their dependencies are met by calculating the RS in the manner stated.

4.3 Proactive Sandbox Allocation

Given that typical serverless workloads have their execution time in the same order of magnitude as that of setting up sandboxes (§2.2), we need to ensure that requests are not exposed

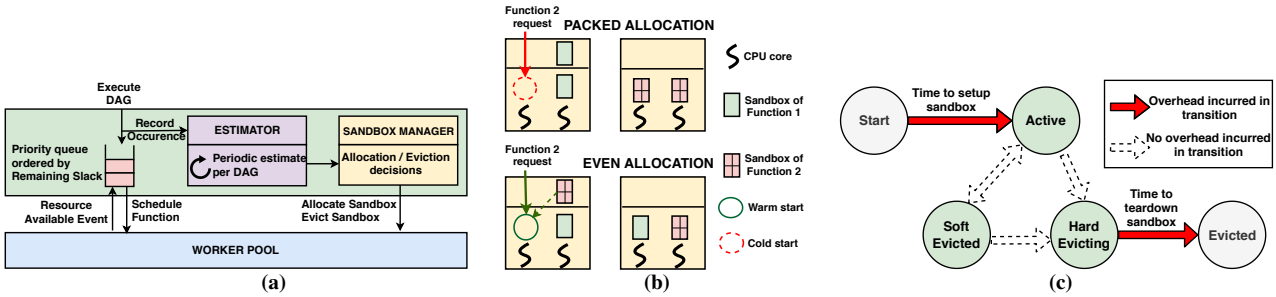


Figure 4. (a) Zoomed-in view of a semi-global scheduler consisting of a priority queue, an estimator, and a sandbox manager (b) Comparison of multiplexing in packed and even allocation policies. With packed allocation, the execution of function 2 incurs a cold start (marked in dashes) due to the unavailability 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) State diagram showing transitions between different stages of the sandbox lifecycle along with overheads incurred

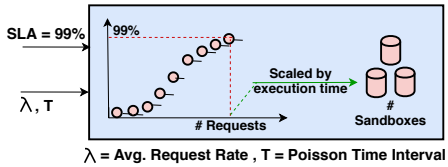


Figure 5. Estimating number of sandboxes to proactively allocate

to this overhead. To achieve this, Archipelago decouples sandbox allocation from scheduling of incoming requests and this allows each SGS to proactively setup sandboxes across its worker pool based on the future expected load. This is in contrast to today's platforms [3] that are not workload-aware and reactively setup sandboxes when a request arrives. By decoupling sandbox allocation from scheduling, Archipelago promotes the pipelining of sandbox allocation with scheduling decisions 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 footprint (§2), 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 administrator 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. Finally, we note that proactively allocated sandboxes are a form of *soft state* [24] 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 DAG that is being handled by the SGS, our goal is to determine the minimum number of sandboxes that need to be allocated for each of its constituent functions, so as to meet the agreed upon SLA. Given the execution time of a function and the SLA, we model how requests of the function arrive to determine the minimum number of sandboxes needed.

In Archipelago, we make an assumption that request inter-arrival times follow an exponential distribution and model the number of requests expected in a given time interval T as a Poisson distribution. Specifically, given the SLA (e.g., 99%), we use the inverse distribution function to find the maximum number of requests that can arrive in T (Fig. 5). However, given that execution time of a function 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.

The SGS requires an estimate of the arrival rate of a function, so as to construct the Poisson distribution, which can then be used to determine the number of sandboxes using the above approach. In the background, the SGS (via its estimator module, Fig. 4a) continuously records the arrival rate of the function (over a 100 ms interval in our prototype) and uses an exponentially weighted moving average (EWMA) over the current interval's measured rate and the previous estimate to get the new estimate. The SGS measures and estimates this for all the functions that it is handling.

4.3.2 Sandbox Placement

Now, given the number of sandboxes that need to be setup proactively 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 number of future requests that will use them.

Given recent efforts [40] 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. 4b). In

Pseudocode 1 Archipelago Sandbox Management

```
1: ▷ Given a DAG D, either allocate or evict sandboxes
2: procedure SANDBOXMANAGEMENT(DAG D)
3:     𝕄                                          ▷ Mapping between DAG and demand
4:     oldDemand = 𝕄[D.id]
5:     newDemand = D.demand
6:     if newDemand > oldDemand then
7:         ▷ Allocate sandboxes as demand increased
8:         for all F ∈ D.functions do
9:             ALLOCATESANDBOXES(F, NEWDEMAND - OLDDEMAND)
10:        end for
11:    else if newDemand < oldDemand then
12:        ▷ Soft evict sandboxes as demand decreased
13:        for all F ∈ D.functions do
14:            SOFTEVICTSANDBOXES(F, OLDDEMAND - NEWDEMAND)
15:        end for
16:    end if
17: end procedure

18: ▷ Given a function F and its demand, allocate sandboxes
19: procedure ALLOCATESANDBOXES(Function F, Int allocDemand)
20:    for _ in range(allocDemand) do
21:        ▷ Get worker which has min sandboxes for this function
22:        minW = GETWORKERWITHMINSANDBOXES(F.ID)
23:        sandboxFound, sandbox = minW.getSoftEvictedContainer(F.id)
24:        if sandboxFound then
25:            ▷ Preferentially allocate a soft evicted sandbox
26:            minW.SoftAllocate(sandbox)
27:            continue
28:        end if
29:        if minW.hasEnoughPoolMem(F) then
30:            ▷ Allocate a new sandbox if enough memory available
31:            minW.Allocate(F)
32:        else
33:            ▷ Otherwise evict a sandbox and allocate
34:            minW.HardEvict(F)
35:            minW.Allocate(F)
36:        end if
37:    end for
38: end procedure

39: ▷ Given function F, evict enough sandboxes to launch a sandbox of F
40: procedure HARDEVICT(Function F)
41:    while w.freePoolMem < F.memNeeded do
42:        victimF = w.getVictimF()          ▷ Get function based on fairness metric
43:        w.Evict(victimF)
44:        w.freePoolMem += victimF.memNeeded
45:    end while
46: end procedure
```

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 new sandbox as no compatible sandbox is available on the worker.

Instead, in Archipelago, for a given function, we *evenly* spread its sandboxes across the various workers (lines 18-38 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. 4a): (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. 4b, the request does not incur setting up overhead as a compatible sandbox is available.

4.3.3 Sandbox Eviction

The previous section described how an SGS proactively allocates containers based on estimations. However, when the estimations 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 Archipelago we *lazily* evict containers from the worker pool to avoid unnecessary sandbox allocation overheads.

Pseudocode 2 Archipelago Per DAG SGS Scaling

```
1: ▷ Given a DAG D, determine if scaling is required
2: procedure SCALING(DAG D)
3:     N⃗                                  ▷ per associated SGS sandbox count for DAG D
4:     ⟶qDelay                            ▷ per associated SGS observed queuing delay for DAG D
5:     weightedQDelay = Σ_i (N⃗_i * ⟶qDelay_i) / Σ_i N⃗_i
6:     scalingMetric = weightedDelay / D.slack
7:     if scalingMetric > ScaleOutThreshold then
8:         SCALEOUT(D)
9:     else if scalingMetric < ScaleInThreshold then
10:        SCALEIN(D)
11:    end if
12: end procedure
```

In Archipelago, a sandbox goes through two stages of eviction - *soft eviction* and *hard eviction* (Fig. 4c). When the estimates fall below what was previously estimated, 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 it takes, 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 is repeated until the required number of sandboxes are soft evicted(lines 11-15 in Pseudocode 1). The aforementioned approach balances the sandboxes across workers to the extent possible which improves statistical multiplexing Having soft evicted sandboxes enables Archipelago to deal with temporary load fluctuations in a better manner. 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 proactively allocated (lines 39-46 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. Also, the SGS prefers to hard evict a soft evicted sandbox first before evicting a sandbox 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 then discuss how our service performs the tasks at hand (§5.2).

5.1 Service Responsibilities

The LBS has two key responsibilities : (1) balance load across SGS: given that the underlying SGSs partitions the cluster, the LBS should ensure that the load is spread across the various SGS 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 DAG

Given that the underlying cluster is partitioned and is managed by various SGSs, a key question that needs to be answered is among how many SGSs should the incoming requests of a DAG be spread? A possible solution would be to use all the available SGSs and spread the incoming requests evenly. This would avoid hotspots but naively applying such an approach in our context 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 DAG 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 a DAG. However, to ensure that this dynamic approach is effective and performant, the following questions need to be answered - (1) what should 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 a number of situations under which the current number of SGSs associated with a DAG could be too few, requiring scale out. First, when the incoming workload of a DAG 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 DAGs 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 independent metrics to indicate the occurrence of the above situations, we leverage queuing delay experienced by requests (of the corresponding DAG) at the SGS as the universal metric. Queuing delay covers all the situations and is easily observable. Specifically, each SGS measures the queuing delays per DAG using EWMA (similar to how it estimates the per DAG RPS) over a window. Having a window ensures that our system does not react to transient changes in queuing delays.

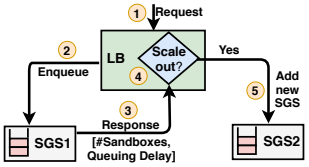


Figure 6. Interaction of load balancer with SGSs during a scale out

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 a particular DAG arrives for the first time at the LBS, we use consistent hashing [31] to determine which SGS to route requests to. Specifically, the LBS maintains a consistent hash ring - with all the underlying SGSs hashed to the ring (by using their ID). Now when the first request arrives, the LBS hashes the DAG ID to the ring and assigns it its initial SGS. Using consistent hashing ensures that no single SGS is overwhelmed by being responsible for a large share of DAGs.

Scale Out (see Fig 6). The LBS receives the queuing delay observed by the requests of this DAG 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 out by associating another SGS (the next one in the ring) with this DAG (lines 7-8 in Pseudocode. 2). Upon scaling out, the LBS updates the mapping in a reliable storage system and notifies each of the SGSs associated with this DAG to reinitialize the queuing delay windows so that we can observe the impact of our decision. The LBS makes the next scaling decision only once the windows are filled up to avoid reacting to transient changes in queuing delay.

Scale In. The LBS follows a similar process as described above to decide if we need to dissociate an SGS from the DAG, with the only difference being that we scale in if the scaling metric falls below the scale-in threshold (lines 9-10 in Pseudocode. 2). We remove the SGS that was added last from the pool of associated SGSs. To avoid oscillations in the scaling process, we keep the scale-in threshold well below the scale-out threshold.

Scaling Metric. Given the per-SGS queuing delay, in order 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 (line 5 in Pseudocode. 2). Next, we normalize this weighted sum by the available slack for the DAG (line 6 in Pseudocode. 2). 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 DAG 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 in comparison to the latter.

5.2.3 How to do transparent scaling?

When the LBS dynamically scales the SGSs associated with a DAG, we also need to ensure that this does not have a negative impact on the requests. Archipelago achieves this by gradually scaling out and in rather than scaling instantly.

When scaling out, we associate an additional SGS with the DAG. However, instantly sending requests to the new SGS will lead to these requests experiencing sandbox allocation overheads. The LBS circumvents this issue by gradually ramping up the newly added SGS in the following manner - (1) uses *lottery scheduling* to perform 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 DAG and (2) notifies the new SGS to proactively allocate the average number of sandboxes 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 also need to scale in gradually. An instant scale in can result in overwhelming the reduced subset of SGSs. We solve this issue by maintaining two lists of SGSs for a DAG - an *active list* and a *removed list*. While scaling in, we remove the SGS from the active list and place it in the removed list. During lottery scheduling, we still consider SGSs in the removed list but scale down the lottery tickets given to such SGSs by a *discount factor*. This ensures that the subset is not overwhelmed and gradually removes the SGS.

6 Implementation

We built our prototype in Go (~15K LOC). All the services are implemented as multi-threaded processes. Our LBS has an HTTP front end to receive events that trigger the execution of the corresponding DAGs. The SGS consists of the three loosely coupled modules - scheduler, estimator and sandbox manager. All workers in the cluster have 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 goroutines as sandbox environments. The external state store is responsible for keeping the SGS and LB state and uses separate goroutines for handling requests. All the communication between the different components happen using protocol buffers [17]. We integrate our prototype with Prometheus [16] and Grafana [15] for timely monitoring. Next, we briefly describe the fault tolerance properties of our implementation.

6.1 Fault Tolerance

We assume the standard fail-stop model in which the Archipelago's services can crash at any point and that there exists a failure detector that can immediately detect the failure.

Worker Failures. When a worker fails, the corresponding SGS updates its cluster view. Additionally, our per-DAG scaling strategy naturally adapts to worker failures and limits the negative impacts on the incoming workload under such situations. Specifically, when workers fail, the cumulative load that an SGS can handle is reduced, and to meet deadlines, we would ideally need to scale out. Since the scaling indicator is the queuing delay, the LBS would observe an increased delay and scale out. Also, given that we evenly spread the proactive sandboxes, on worker failure, incoming requests still benefit from proactive allocation on other workers.

SGS and LB Failures. Archipelago maintains the state required by the SGSs (e.g., proactive sandbox count, estimation state) and LB (per-DAG SGS mapping) in a reliable external store. This ensures that a new instance can recover the state from the store and continue execution.

7 Evaluation

We evaluate the end-to-end benefits of Archipelago on a 74-machine cluster deployed on CloudLab [11] and compare against a baseline that reflects current state-of-the-art serverless platforms [3]. We also carry out several microbenchmarks to delve deeper into Archipelago's benefits.

7.1 Experimental Setup

Our **testbed** has 38 machines with 20 cores and 36 with 28 cores. All machines have 256GB memory and 10Gbps NIC. We partition the cluster to have 8 SGSs, each of which has a worker pool consisting of 8 machines. Each SGS runs on a separate machine. The setup uses a single load balancer to constitute the LBS. We choose the *ScaleOutThreshold* to be 0.3 (§7.5) and model the sandbox setup overheads for different DAGs to be in the range of 125 ms [12] to 400 ms, a conservative estimate given our measurements of overheads in downloading code packages from S3 (§2.2).

Baseline Stack. Our baseline uses a centralized scheduler (similar to [3]) where requests are processed in FIFO order. Also we *reactively* allocate sandboxes and keep them in memory with a fixed inactivity timeout of 15 mins [3, 8, 10].

Workload. We consider four different classes of DAGs: **(i) C1** consists of DAGs that have a single function, short execution times and tight deadlines. These DAGs represent user-facing functions. **(ii) C2** consists of DAGs that have a single function, short execution times, and less strict deadlines. These DAGs represent non-critical user-facing functions (such as updating a metrics dashboard). **(iii) C3** consists of DAGs that have chained functions, medium execution times and relatively strict deadlines compared to their execution times. These

	Avg. RPS	Amplitude	Period	Exec. Time	Slack
C1	[600,1200]	[100,800]	[10,20]s	[50-100]ms	[100,150] ms
C2	[400,800]	[200,400]	[30,40]s	[100-200]ms	[300,500] ms
C3	[500,1000]	[200,600]	[10,20]s	[250-400]ms	[200,300] ms
C4	200	0	∞	[300-600]ms	[500,1000]ms

Table 1. Execution time and slack for various DAG classes (both Workloads). DAGs follow sinusoidal patterns in Workload 2 and we randomly sample the sinusoid pattern parameters from the range stated, depending on the class.

DAGs represent more expensive user-facing functions. **(iv) C4** consists of DAGs that have branched structures, high execution times and loose deadlines. These DAGs represent background jobs that typically perform batch execution [28]. We randomly sample execution time and slack details from the ranges mentioned in Table 1.

We construct 2 workloads to model the arrival rate of requests belonging to different classes. For **Workload 1**, we model the request arrival pattern to follow a Poisson distribution. For the classes C1-C4, we periodically (every second) sample the mean arrival rate from an interval of 800-1200, 600-900, 600-800, 50-150 RPS respectively. For **Workload 2**, we model the request arrival pattern to follow a sinusoidal distribution. The details are captured in Table 1. Both workloads keep the cluster CPU load between ~70% to ~110%.

Metrics. We use a variety of metrics to evaluate different components of the platform - **(i) End-to-end (E2E) latency** - represents the turn around time of a request. **(ii) % Deadlines Met** - the % of requests that complete within their deadline. **(iii) Queuing Delay** - the time spent by a request in the queue before it is scheduled. **(iv) Cold Starts** - the number of requests that experience the overhead of sandbox allocation.

7.2 Macrobenchmarks

Figure 7a shows the end-to-end latencies for Archipelago and the baseline for Workload 1. Archipelago reduces the tail latencies (99.9%-ile) by 20.83× over the baseline. Additionally, in the steady state Archipelago matches the performance of the baseline (50%-ile). Figure 7b shows that these tail latency violations lead to around 33% deadlines being missed by the baseline while Archipelago misses only 0.76% deadlines.

We find that the high tail latencies incurred by the baseline come from requests getting queued up while sandboxes are being reactively allocated. Archipelago minimizes the number of cold starts by proactively allocating sandboxes and being deadline-aware (§7.2.1). Similar results are observed for Workload 2 - Archipelago reduces tail latencies by 35.97× over the baseline (Figure 7c) and misses 0.98% deadlines in comparison to 9.66% missed by the baseline (Figure 7d).

Additionally, in the context of baselines, we see that typically the classes of DAGs that have a slower arrival rate miss more deadlines (C4 misses more than others, C2 misses more than C1). Further analysis indicates that DAGs with lower request rate tend to be stuck behind requests from DAGs with higher request rate in the scheduling queue. Archipelago naturally mitigates this by using a queuing-aware scaling indicator that triggers scale out to another SGS.

7.2.1 Sources of Improvement

We next analyze the sources of improvement for the trends observed for Workload 2. We choose this workload to highlight how Archipelago behaves when the workload does not follow the Poisson arrival process assumed by our estimation logic.

Lower Queuing Delays. Figure 8a shows that Archipelago has lower queuing delays at an SGS. The tail queuing delay for Archipelago is 47.5× lower than the baseline. This is mainly due to - (i) LBS performing sandbox-aware routing and (ii) SGS proactively allocating sandboxes which ensures that requests do not spend additional time in the SGS queue waiting for the allocation to finish.

Fewer Cold Starts. We see that Archipelago overall incurs 24.38× fewer cold starts since sandboxes are proactively allocated in a workload-aware manner. In contrast, the baseline reactively allocates sandboxes leading to more cold starts.

Workload-aware proactive allocation. Figure 8b shows the number of proactively allocated sandboxes for the C2 DAG. We see that the SGSs' estimation is able to closely follow the ideal number of sandboxes required. In the worst case, Archipelago allocates 37.4% more sandboxes. This is primarily because the SGS provisions sandboxes for the worst case load to ensure requests do not incur cold starts (§4.3.1). Additionally, there are instances when an SGS allocates proactive sandboxes anticipating future requests, but then the DAG scales out to another SGS due to contention at the prior one. However, this is not a concern since Archipelago uses an isolated memory pool for proactive sandbox allocation along with a workload-aware eviction policy.

7.3 Microbenchmarks

To further delve into the benefits of Archipelago, we run several microbenchmarks at a smaller scale, with 1 LB, one or more SGSs, and each SGS having 10 workers. We use synthetic workloads that stress specific components of the stack.

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 of evenly spreading sandboxes across the worker pool to an alternative where the SGS packs sandboxes on the same worker. We choose a workload with a single DAG where the request arrival follows a sinusoidal distribution with an average RPS of 1200, amplitude of 600, and a 20s period.

Given that both approaches see the same workload, the number of proactive sandboxes allocated are the same. However, we observe (see Figure 9) that the packing approach leads to ~70% deadlines not being met during intervals of

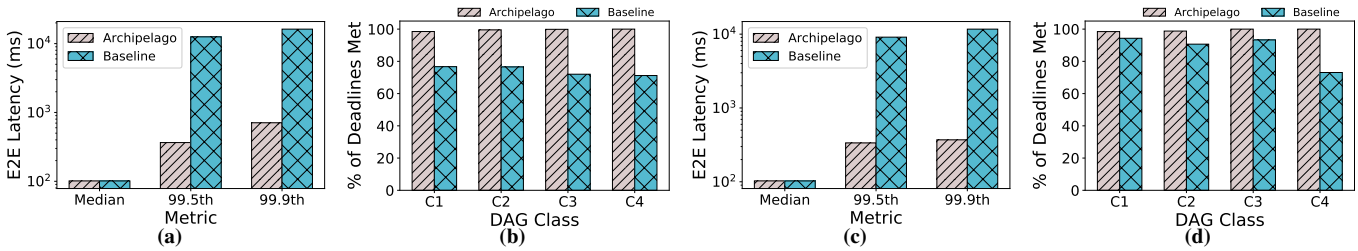


Figure 7. Archipelago vs. baselines. (a) E2E Latency - Workload 1 (b) % Deadlines Met - Workload 1 (c) E2E Latency - Workload 2 (d) % Deadlines Met - Workload 2

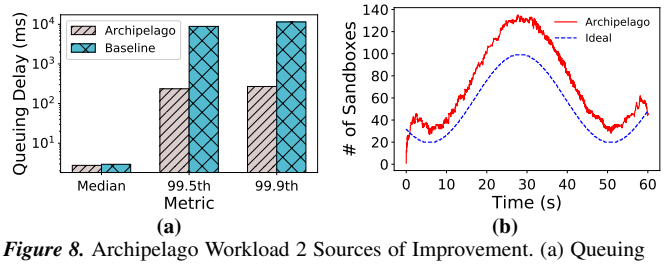


Figure 8. Archipelago Workload 2 Sources of Improvement. (a) Queuing Delay (b) Proactive Allocation Vs Ideal Allocation

increased load (intervals 3-4, 8-9). This does not happen when sandboxes are evenly spread. This is primarily because in case of packing, the sandboxes are available on a smaller fraction of workers, and at increased load, requests gets scheduled on workers that do not have proactively allocated sandboxes available, leading to missed deadlines. In contrast, even placement of sandboxes offers better statistical multiplexing resulting in better handling of bursts.

Benefits of workload-aware hard eviction. We compare our approach of fair eviction with LRU (§4.3.3). We choose a workload that consists of 2 DAGs - one that has constant request rate of 200 RPS and another one that has an on/off pattern with 100 RPS. We have configured the proactively memory pool to be low so that it causes hard eviction. We observe that LRU has a higher tail latency by 4.62X in comparison to fair eviction. This is primarily due to LRU optimizing for the short-term without taking into account the sandbox demand, which Archipelago does. Specifically, we observe that during the off-period, using LRU causes all sandboxes of the second DAG to be hard evicted leading to additional sandbox setup overheads during the next on period.

7.3.2 LBS Scaling Strategy

We now evaluate the various aspects of the scaling strategy adopted by the LBS using 5 SGSs with 10 workers each.

Benefits of gradual scale-out. Archipelago gradually scales-out the number of SGSs for a given DAG using lottery scheduling (§5.2.3). We evaluate the benefit of this against a policy where scale-out happens instantly, which leads to LBS routing requests in a round-robin fashion among the SGSs. We choose a workload with a single DAG wherein the request arrival follows a sinusoidal distribution with an average RPS of 800, amplitude of 600, and a 100s period (elongated period to capture a snapshot of the scale-out benefits).

We observe 1.5× higher tail latencies with instant scale-out. This is because when a new SGS is added for a DAG, the LBS immediately starts routing requests to it, without taking into account the number of available sandboxes.

Deadline-aware per-DAG scale-out. Archipelago's per-DAG scaling metric accounts for the amount of slack in the DAG. To study the effect of this, we consider 2 DAGs, both having an execution time of 100 ms. However, one DAG has a slack of 50 ms while the other has a slack of 200 ms. We assume a workload where requests arrive with the same sinusoidal distribution (see Figure 10 for workload).

From Fig 10, we observe that the DAG with smaller slack scales-up to more SGSs than the DAG with higher slack (e.g., smaller slack DAG scales out to 4 while the larger slack DAG scales out to 3 in the 20-30s interval). This shows the benefits of having a deadline-aware scaling metric which can help latency-sensitive foreground apps over background apps.

Contention-aware per-DAG scale-out. Since DAGs from multiple users are multiplexed across the same cluster, it is important to ensure that one DAG does not suffer due to increased request rates of another DAG. To evaluate this, we consider 2 DAGs - one that is bursty and follows a sinusoidal distribution and another that has a low, constant request rate. The request rate of the second DAG is set such that it requires only a single SGS if it is the only DAG utilizing the cluster (see Figure 11 for workload).

When the second DAG experiences contention for the cluster due to the bursty nature of the first DAG, we observe from Figure 11 that the LBS is able to handle this by scaling-out the second DAG to another SGS (e.g., at ~5s). We also notice that the LBS scales-down once the contention reduces (e.g., at ~17s). This is possible since we co-design the LBS and SS layers, allowing us to observe the contention at each SGS and appropriately scale in a deadline-aware manner.

7.4 System Overheads

Since Archipelago aims to provide low latency scheduling, we present some of the overheads that arise in the critical path of request execution. From our macrobenchmarks, we notice that the median (99%-ile) per request overhead added by the LBS to decide where to route is 190μs (212μs). Scheduling decisions at SGS added an additional median (99%-ile) overhead of 241μs (342μs) per request. We also measure the time

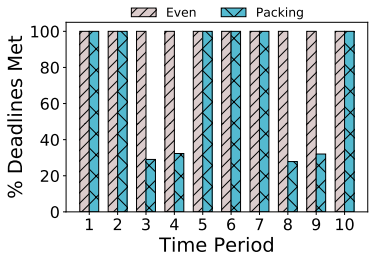


Figure 9. Sandbox Placement - Even Vs. Packing

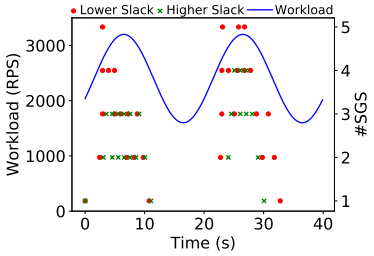


Figure 10. A DAG with lower slack scales-out more than a DAG with higher slack

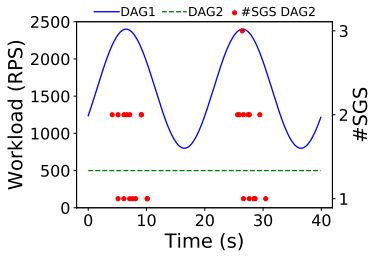


Figure 11. Contention from a bursty DAG (DAG1) causes DAG2 to scale-out

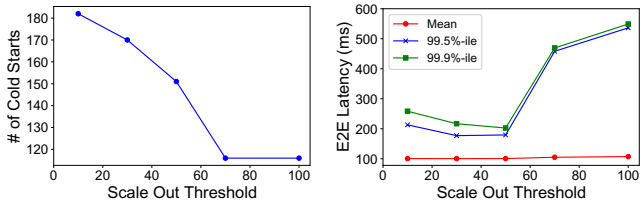


Figure 12. Scale Out Threshold Vs. (a) Cold Starts and (b) E2E Latency

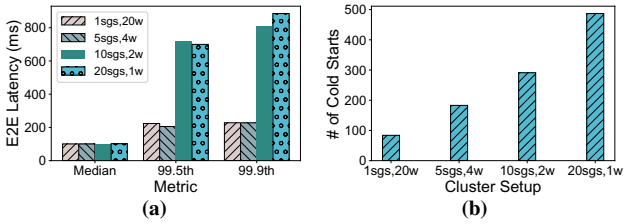


Figure 13. Comparison of (a) E2E latencies and (b) Cold starts for different cluster configurations

taken to scale-out at the LBS as well as time to make an estimation decision. Neither of these happen in the critical path, but help determine the robustness of the system. Scale-out takes a median (99%-ile) time of 128μs (197μs). Estimations at an SGS take a median (99%-ile) time of 879μs (1352μs).

7.5 Sensitivity Analysis

Scale Out Threshold (SOT). Lower values of SOT mean that the LBS scales-out more aggressively. This would result in more frequent scale-outs amounting to a greater number of cold starts as seen in Figure 12a. On the other hand, aggressive scale-out helps keep queuing delays low in comparison to a passive scale-out strategy. Thus, we observe a trade-off between managing queuing delays and the number of cold starts. From Figure 12b, we observe that - (i) At very low *SOT* values, the high number of cold starts negatively impacts the tail latency (ii) At higher *SOT* values, higher queuing delays negatively impacts the tail latency. A cluster operator can thus configure the *SOT* based on knowledge of the workload and the sandbox setup overheads. Based on the above observed values, we choose a *SOT* of 0.3 for our experiments.

SGS Size. Given a fixed number of workers, what should be ideal size of the worker pool under a single SGS? To study this, we consider a setup consisting of 20 workers. We consider 4 ways in which the cluster can be partitioned - (i) 20 SGSs, 1 worker each (ii) 10 SGSs, 2 workers each (iii) 5 SGSs, 4 workers each (iv) 1 SGS, 20 workers each. We choose a workload with a single DAG wherein the request arrival follows a sinusoidal distribution with an average RPS of 600, amplitude of 400, with a period of 20 seconds.

We observe that fine-grained partitioning leads to ~4× higher tail latencies (Figure 13(a)). This is because the LB would need to scale-out more often for each DAG leading to an increased number of cold starts in comparison to when there is no need to scale out as seen in Figure 13(b).

However, having too many workers under an SGS can lead to scheduling overhead becoming a significant contributor to the queuing delay. If this happens, then the LBS would unnecessarily scale out leading to workers under the initial SGS being under-utilized. For functions with 50ms slack, we observed in our testbed, that beyond 64 machines, we were unnecessarily scaling out leading to workers being underutilized.

8 Related Work

Serverless Characterization. [46] looks at how network intensive applications run on serverless platforms whereas [34, 35, 42] characterize the storage requirements of serverless applications. [49] conducted a large measurement study to understand performance, resource management as well isolation in serverless platforms. Similarly, [36] also conducted measurements on the public offerings of serverless frameworks. To the best of our knowledge, no prior works have characterized real world serverless applications.

Sandbox Overhead Reduction. [40] reduces the start up times of functions in OpenLambda [26] through caching Python runtimes and packages, and uses low-latency isolation primitives. [21] advocates for the usage of language-based isolation instead of using traditional virtualization techniques. [19] proposes a two level isolation wherein functions of the same application run within the same container as separate processes. [37] identifies that the container networking setup takes significant time and pre-creates such resources to overcome the overhead, and dynamically binds to a container. All these works are complementary with Archipelago's efforts of reducing the impact of sandbox setup overheads.

Scheduling Architectures. We now discuss scheduling architectures other that those compared to earlier (i.e., [38, 41]). Borg [48] uses random sampling while calculating scores and thus trades off scheduling optimality for scalability. Omega [44]

uses multiple parallel schedulers but trades off scheduling predictability for scalability due to the overheads involved in resolving conflicts which would happen often in our setting due to the resources being held for short durations. While Apollo [22] 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 Archipelago, we can accommodate various kinds of tasks and meet deadlines for all of them. Mercury [30] 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 sub-optimal placement for the shorter tasks.

9 Conclusion

In this paper, we consider the problem of ensuring low latency function execution in serverless settings, an important problem that has not received attention. Our system, Archipelago, meets this goal using the following 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 Archipelago meets the deadlines for more than 99% of realistic application request workloads, and reduces tail latencies by up to ~36X compared to state-of-the-art.

References

[1] 2017. Azure Functions. https://functions.azure.com.

[2] 2017. Google Cloud Functions. https://cloud.google.com/functions.

[3] 2017. IBM Bluemix Openwhisk. https://www.ibm.com/cloud-computing/bluemix/openwhisk.

[4] 2019. Amazon Simple Notification Service. https://slack.com/.

[5] 2019. Amazon Simple Storage Service. http://aws.amazon.com/s3.

[6] 2019. Amazon Web Services. https://aws.amazon.com/.

[7] 2019. AWS Lambda. https://aws.amazon.com/lambda/.

[8] 2019. AWS Lambda Cold Starts. https://mikhail.io/serverless/coldstarts/aws/.

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

[10] 2019. Azure Functions Cold Start. https://mikhail.io/serverless/coldstarts/azure/.

[11] 2019. Cloudlab. https://cloudlab.us.

[12] 2019. Firecracker MicroVM. https://firecracker-microvm.github.io/.

[13] 2019. Google Cloud Functions. https://cloud.google.com/functions/.

[14] 2019. Google Container Engine. http://kubernetes.io.

[15] 2019. Grafana. https://grafana.com/.

[16] 2019. Prometheus. https://prometheus.io/.

[17] 2019. Protocol Buffers. https://bit.ly/1mISy49.

[18] 2019. Serverless WarmUp Plugin. https://github.com/FidelLimited/serverless-plugin-warmup.

[19] 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.

[20] 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*. ACM, 263–274.

[21] 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.

[22] 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*.

[23] 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)*. 363–376.

[24] 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.

[25] 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.

[26] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *HotCloud 16*.

[27] 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*.

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

[29] 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:cs.OS/1902.03383

[30] 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*.

[31] 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*.

[32] James E Kelley. 1961. Critical-path planning and scheduling: Mathematical basis. *Operations Research* 9, 3 (1961), 296–320.

[33] James E Kelley. 1963. The critical-path method: Resources planning and scheduling. *Industrial scheduling* 13 (1963), 347–365.

[34] 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.

[35] 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.

[36] 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.

[37] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile cold starts for

scalable serverless. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[38] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. *CoRR* abs/1712.05889 (2017). http://arxiv.org/abs/1712.05889

[39] 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.

[40] 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*.

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

[42] 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)*. 193–206.

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

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

[45] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: serverless linear algebra. *arXiv preprint arXiv:1810.09679* (2018).

[46] 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.

[47] 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*.

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

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