Fifer: Tackling Resource Underutilization in the Serverless Era

Jashwant Raj Gunasekaran
The Pennsylvania State University
jashwant@psu.edu

Prashanth Thinakaran
The Pennsylvania State University
prashanth@psu.edu

Nachiappan C. Nachiappan
The Pennsylvania State University
nachi@alumni.psu.edu

Mahmut Taylan Kandemir
The Pennsylvania State University
mtk2@psu.edu

Chita R. Das
The Pennsylvania State University
cxd12@psu.edu

Abstract

Datacenters are witnessing a rapid surge in the adoption of serverless functions for microservices-based applications. A vast majority of these microservices typically span less than a second, have strict SLO requirements, and are chained together as per the requirements of an application. The aforementioned characteristics introduce a new set of challenges, especially in terms of container provisioning and management, as the state-of-the-art resource management frameworks, employed in serverless platforms, tend to look at microservice-based applications similar to conventional monolithic applications. Hence, these frameworks suffer from microservice agnostic scheduling and colossal container over-provisioning, especially during workload fluctuations, thereby resulting in poor resource utilization.

In this work, we quantify the above shortcomings using a variety of workloads on a multi-node cluster managed by the Kubernetes and Brigade serverless framework. To address them, we propose Fifer — an adaptive resource management framework to efficiently manage function-chains on serverless platforms. The key idea is to make Fifer (i) utilization conscious by efficiently bin packing jobs to fewer containers using function-aware container scaling and intelligent request batching, and (ii) at the same time, SLO-compliant by proactively spawning containers to avoid cold-starts, thus minimizing the overall response latency. Combining these benefits, Fifer improves container utilization and cluster-wide energy consumption by 4× and 31%, respectively, without compromising on SLO’s, when compared to the state-of-the-art schedulers employed by serverless platforms.

CCS Concepts:
- Computer systems organization → Cloud Computing; Resource-Management; Scheduling.

Keywords: serverless, resource-management, scheduling, queuing

ACM Reference Format:

1 Introduction

The advent of public clouds in the last decade has led to the explosion in the use of microservice-based applications [40]. Large cloud-based companies like Amazon [30], Facebook [83], Twitter [60], and Netflix [64] have capitalized on the ease of scalability and development offered by microservices, embracing it as a first-class application model [88]. For instance, a wide range of Machine Learning (ML) applications such as facial recognition [18], virtual systems [47, 75, 98, 100], content recommendation [48], etc., are realized as a series of inter-linked microservices1, also known as microservice-chains [68, 101]. These applications are user-facing [94] and hence, demand a strict service-level objective (SLO), which is usually under 1000 ms [42, 52, 61]. It is, therefore, imperative to mitigate the end-to-end latency of a microservice-chain to provide a satisfactory user experience. The SLOs for such microservices are bounded by two factors – (i) resource provisioning latency, and (ii) application execution time. As a majority of these microservices usually execute within a few milliseconds [46, 47], serverless functions [3, 8, 10] have proven to be an ideal alternative over virtual machines (VM), as they not only have very short resource-provisioning latencies, but also abstract away the need for the users to explicitly manage the resources.

However, adopting serverless functions introduce a new set of challenges in terms of resource management (RM) for the cloud providers [56, 80], especially when deploying 1A microservice is the smallest granularity of an application performing an independent function, a.k.a functions in serverless domain.
large number of millisecond-scale function chains\(^2\). There has been considerable prior work \([2, 16, 39, 70, 82, 84]\) in RM frameworks to leverage the asynchronicity, event-drivenness and scalability of serverless applications. Despite having these sophisticated frameworks, the resource management for thousands of short-lived function-chains still has significant inefficiencies for resource utilization and SLO-compliance. Viewing functions in a function-chain as truly independent entities, further accentuates these inefficiencies. Studying the state-of-the-art RM frameworks, we identify three critical reasons for these inefficiencies.

- Most frameworks are built just to meet each individual function’s SLOs. **Being imperceptive to the total end-to-end SLO of the function-chain leads to sub-optimal uniform scaling of containers at every function stage.** This inherently leads to over-provisioning containers, which in turn results in large number of machines to host idle containers thereby increasing the provider’s operating costs.

- Many frameworks **employ one-to-one mapping of requests to containers** \([92]\). This inherently leads to excessive number of containers being provisioned when handling a sudden burst of requests than that are actually needed to meet the application-level SLOs.

- Lastly, in the quest to reduce the number of provisioned containers, certain frameworks \([10, 12]\) make use of **naive queuing of requests on to a static pool of containers**. Fixing the number of containers in an application agnostic manner results in SLO violations, especially for functions with strict SLO requirements.

These inefficiencies collectively open the door towards stage-aware resource management by exploiting the “leftover slack” in these function chains. Leveraging slack allows individual functions to be queued in batches at existing containers without violating the application-level SLOs. In this paper, we present, \textit{Fifer}, which to the best of our knowledge, is the first work that employs stage-aware container provisioning and management of function chains for serverless platforms. \textit{Fifer} \(^3\) makes use of novel slack estimation techniques for individual functions and leverages it to significantly reduce the number of containers used, thereby leading to increased resource utilization and cluster energy efficiency. While slack-based request queuing can significantly minimize the number of containers spawned, it still leads to SLO violations because of cold-starts, especially during dynamic load fluctuations. \textit{Fifer} makes use of proactive container provisioning using load prediction models to minimize the SLO violations incurred due to cold starts. To this end, the **key contributions** of the paper are the following:

- **We characterize the effect of cold-starts** for various ML inference applications on AWS serverless platforms and show that they have a large disparity in container provisioning times compared to application execution times. Further, we show that for an incoming series of requests, queuing them for batched execution at warm containers can greatly reduce the number of containers being spawned.

- **We introduce the notion of slack**, which is defined as the difference between execution time and overall response latency. We propose \textit{Fifer}, which takes advantage of this slack towards calculating the batch-size (queue length) to determine the optimal number of requests that could be queued at every stage. \textit{Fifer} is inherently **stage aware**, such that it allocates slack to every function stage of an application proportional to its execution time, and independently decides the scale-out threshold for every stage.

- **We quantitatively characterize the benefits of using different load prediction models (ML and non-ML) to enable proactive container provisioning.** Based on our findings, we implement \textit{Fifer} with a novel \textit{LSTM-based} \([51]\) prediction model, which provides fairly accurate request arrival estimations even when there are large dynamic variations in the arrival rate.

- We implement \textit{Fifer} as a part of the Brigade serverless workflow framework \([6]\) in a Kubernetes cluster and extensively evaluate it with different request arrival patterns using both synthetic traces and comprehensive real-world traces to show its advantage over other frameworks. Our results from the experimental analysis on an 80 core cluster and extensive large-scale simulations show that \textit{Fifer} spawns up to 80\% fewer containers on an average, thereby improving container utilization and cluster-wide energy savings by up to 4\times and 31\%, respectively, when compared to state-of-the-art non-queuing reactive schedulers employed in serverless platforms.

2 Background and Motivation

We start with providing a brief overview of serverless function-chains followed by a detailed analysis of their performance to motivate the need for \textit{Fifer}.

2.1 Serverless Functions Chains

The overall framework for deploying microservice-chains in serverless platforms is shown in Figure 1. Multiple functions (with one function per microservice) are stitched together using synchronization services such as AWS Step Functions \([4, 5, 7, 21]\) to form a “function-chain”. Step functions are billed for individual function invocations and memory usage and also for the number of transition across different functions for every invocation. Though the whole function-chain can also be deployed as one monolithic function, splitting them has several known advantages, in terms of ease of deployment and scalability per microservice. The actual transition between each function pair is in the form

\(^2\)We refer to microservice-chains and function chains interchangeably throughout the paper.

\(^3\)A \textit{Fifer} plays a small flute to help soldiers in a brigade (or battalion) to keep their marching pace in coordination with the drummers. In spirit, our framework helps the Brigade system in Kubernetes to adapt to function-chains by being proactive and stage-aware.
of communication events over a centralized event bus. Due to the stateless natures of serverless functions, input data such as pre-trained models, etc., need to be retrieved from ephemeral data storage like AWS S3 [73]. For further details on serverless functions, we refer the reader to prior works [23, 38, 49, 50, 55, 65, 66, 76, 80, 92]. In the context of this paper, we specifically focus on scenarios where tenants choose serverless platforms to host their applications. In the case of multi-tenancy, our proposed ideas can be individually applied to each tenant. Note that, we limit the scope of this paper to container provisioning and management.

2.2 Shortcomings of Current Serverless Platforms

We start by describing the two major implications observed in current serverless platforms, with respect to hosting individual functions and function-chains.

2.2.1 Cold-Start Latency for Single Functions

When a function is invoked as a part of deploying the tenant application in serverless computing, it is launched within a newly-created container which incurs a start-up latency known as cold-start. Though modern virtualization technologies like microVMs [20] reduce container start-up time, the cold-start time is dominated by application and runtime initialization. To avoid cold-starts, public cloud providers like Amazon try to launch every function in warm containers (existing containers) [92] whenever available. However, if all warm containers are occupied, a new container has to be spawned, which usually takes a few seconds. For applications which execute within a few milliseconds, this penalty would be significantly higher, especially when the applications are user-facing where it is crucial to ensure the SLO.

To characterize the cold-start and warm-start latencies, we execute an ML image inference application using the Mxnet [26] framework on AWS lambda [3]. We use 7 different pre-trained ML models with varying execution times proportionate to the model sizes. Figure 2 plots the breakdown of total execution time for both cold and warm start as follows: (i) the time reported by AWS lambda for executing the inference (exec_time), and (ii) the round-trip time (RTT) starting from the query submission at a client to receiving the response from AWS lambda. Cold start latency is measured for the first invocation of a request because there would be no existing containers to serve the request. Warm start is measured as an average latency of 100 subsequent requests, over a 5 minute interval. From Figure 2a, it is evident that the cold start overheads on many occasions are higher than the actual query execution time, especially for larger models like Resnet-200. For warm starts, as shown in Figure 2b, the total time taken is within 1500 ms, except for larger models. We can infer that the cold starts contribute ~2000 to 7500 ms on top of execution time of the function. Note that, the variability in exec_time across each model is due to the pretrained model fetching time from AWS S3 [73].

To avoid cold-starts, certain frameworks [7, 14] employ a pre-warmed pool of idle containers which results in wasted memory consumption, in turn leading to energy inefficiency. This inefficiency can be potentially avoided for millisecond-scale applications (e.g., Squeezenet [54] in Figure 2) by allowing the requests to queue up on existing containers rather than launching them on separate ones. This can be done when the delay incurred from cold-starts is higher than the delay incurred from queuing the requests. Hence, the decision to queue versus starting up a new container depends on the SLO, execution times of the application and the cold-start latencies of the containers. In contrast, RM frameworks used in Azure are known to queue the incoming requests [92] on a fixed number of containers. Fixing the number of containers in an application agnostic manner will result in SLO violations, especially for functions with strict response latencies. Also the schedulers used in existing open-source platforms like Fission [14], Knative [59] use horizontal pod autocalsers which are not aware of application execution times to employ queuing.

**Key takeaway:** Based on SLOs, cold-start latencies and execution times of applications, queuing functions can minimize the number of containers spawned without violating SLOs.
A similar trend of non-uniform execution times is observed. Therefore, the major sources for execution time variability vectors, and exhibit input-independent control flow [42].

In the case of function-chains consisting of a series of serverless functions (as described in Section 2.1), containers would be spawned individually for every stage. In existing serverless platforms, the RM framework would uniformly spawn containers at each stage depending on the request arrival load. However, the execution times of the functions at each stage are not uniform. Figure 3a shows the breakdown of execution times per stage for 4 different microservice-chains. A detailed description of all the microservices used within the applications are given in Table 3. Consider the Detect Fatigue application shown in Figure 3a. It can be seen that 81% of the total execution time is dominated by stage-1, whereas the other stages together take less than 20% of the total time. A similar trend of non-uniform execution times is observed for the other three applications as well. Hence, it would be ideal to employ per-stage intelligent queuing rather than uniformly queuing requests across all stages, because the latter would lead to poor container utilization.

To effectively exploit this per-stage variability described above, there are two assumptions: (i) the execution times for each stage of an application has to be known apriori, and (ii) the execution times should be predictable and not have large variations. The first assumption can be held true for serverless platforms because the applications are hosted as functions prior to the execution. A simple offline profiling can estimate the execution times of the functions. The second assumption also holds especially for ML-based applications because the ML-models use fixed-sized feature vectors, and exhibit input-independent control flow [42]. Therefore, the major sources for execution time variability come from application-independent delays that are induced by (i) scheduling policies or (ii) interference due to request co-location on the same servers. To support this claim, we conduct a characterization of 8 ML-based microservices from Djinn&Tonic Benchmark Suite. As shown in Figure 3b, the standard deviation in execution time measured across 100 consecutive runs of each microservice is within 20ms. In this experiment, the input size (image pixels or speech query) for all the microservices are fixed. Note that execution will vary depending on the input size to each microservice (for e.g., 256x256 vs 64x64 image for IMC application). In our experiments we find a linear relationship between the execution time and the input size for these applications.

**Why does slack arise?** Though user-facing applications can have varied runtimes, the SLO requirement is deterministic because it is based on human perceivable latency. Because these applications are typically short-running, considerable amount of slack will exist. If we know the end-to-end runtime, we can estimate slack as the difference between runtime and response latency. For example, consider the execution times of the four ML based applications shown in Figure 3a: (i) Detect Fatigue, (ii) Intelligent Personal Assistant (IPA), (iii) Image Recognition (IMG), and (iv) Face Security. It can be seen that the maximum execution time among them is well within 220ms. If the end-to-end response latency is fixed at 1000ms, it is evident that all applications have ample amount of slack to queue requests together.

**Key takeaway:** RM frameworks should capitalize on both — variability of execution time across stages, as well as overall application slack, by determining the optimal queue length to queue requests at every stage. This, in turn, can lead to better bin-packing of requests into fewer containers improving overall container utilization.

### 3 Preamble to Fifer

The above set of challenges motivate us to rethink the design of scheduling and resource management framework that can efficiently handle function-chains in serverless platforms. This section introduces high level overview how queuing and batching can be leveraged by RMs. Our baseline is representative of a RM used in existing serverless platforms like AWS lambda [92], which spawns new containers for every request if there are no idle containers (as explained in Section 2.2). On top of these RM frameworks, one can additionally batch the requests by queuing them at every stage of an application, which we name as Request Batching RM (RBRM). The number of requests which can be queued in a container is defined as the batch size ($B_{size}$) of the container. Essentially, $B_{size}$ is the length of the processing queue each container. Contrary to existing RMs [10, 14], in RBRM, instead of statically assuming the $B_{size}$, we calculate it as a function of execution time ($Stage_{Exec\_Time}$) and available slack for each stage ($Stage_{Slack}$) as $B_{size} \Leftarrow Stage_{Slack}/Stage_{Exec\_Time}$. Henceforth by batching we mean, every container sequentially processes $B_{size}$ request from the queue. Based on $B_{size}$, we can queue different number of requests at each stage. Figure 4 shows an example of how the baseline RM compares against RBRM for incoming requests. It can be seen that while the baseline (Figure 4a) spawns a total of 24 containers (with 8 containers per stage), the RBRM exploits slack by consolidating requests and spawns only 10 containers in total. Note that RBRM does not violate SLOs despite batching the requests. In Figure 4b, all requests are queued.
4.1 Estimating execution time and slack

As briefly mentioned in Section 3, by knowing the available slack and execution time at each stage, we can accurately determine the number of requests that can be executed in a batch in one container. We conduct offline profiling to calculate the runtimes of all microservices used in six commonly used ML-based applications from the Djinnn&Tonic [46] benchmark suite (briefly explained in Section 2.2.2). Using the offline values, we build an estimation model using linear regression, which accurately generates a Mean Execution Time (MET) of each service for a given input size (shown in Table 3). We do not use larger inputs which violate our SLO requirements. This model is added as an offline component to Fifer.

**Slack Distribution:** To accurately estimate the slack for every application, we fix the SLO (response latency) as 1000ms, which is the maximum of 5× execution_time [42] of all the applications used in our workloads. By knowing the overall application execution time and response latency, the difference is calculated as slack for the application. To determine the slack for every stage of the application, we distribute the total slack to individual stages. This can be done in two ways, (i) the overall slack can be equally divided (ED) and allocated to each stage or (ii) the overall slack can be proportionally allocated with respect to the ratio of the execution time of each stage. In Fifer, we use proportional slack allocation for each stage, as it is known to give better per-stage utilization compared to ED [57].

4.2 Load Balancing

Fifer utilizes a request queue, which holds all the incoming tasks for each stage. We design a load balancer (LB) along with a load monitor that are integrated to each stage (LM) for efficiently scaling containers for the application. Since we know the execution time and available slack, the LB can calculate the batch size (B-size) for each stage.

**Dynamic Reactive Scaling Policy:** To accurately determine the number of containers needed at every stage which is a function of B_size and queue length, we need to periodically measure the queuing delay due to batching of requests. As shown in Algorithm 1, for a given monitoring interval at every stage, the LM monitors the scheduled requests in the last 10s to determine if there are any SLO violations due to queuing delays. This is because there are
not enough containers to handle all the queued requests. In that case, we estimate the additional containers needed using the Estimate_Containers function. By knowing the B_size and number of pending requests in the Queue (PQ[len]), the function can estimate the number of containers $N_c = PQ[len] / B_size$.

In case the time taken to service the requests by queuing on existing containers is lower than the cold-start delays, spawning a new container would be deemed unnecessary. Therefore, the function takes into account, the delay incurred in terms of queuing the request for a longer time vs cold start ($C_d$). The queuing delay threshold $D_f$ for Stage S is calculated using total number of requests that can be executed without violating SLOs ($L$), and total time required to satisfy all pending requests ($T_d$) as shown below:

$$L = \sum_{i=1}^{N} B_{size_i}, \quad T_d = PQ[len] \times S_r, \quad D_f = \frac{T_d}{L}$$

where $S_r$ is the per stage response latency, N is the number of containers in S. $S_r$ for a stage is defined as the sum of its allocated slack and execution time. If $D_f > C_d$, then LB spawns additional containers ($N_c$) for each stage. We refer to this as dynamic reactive scaling (RScale) policy.

**Stage-aware Container Scaleout:** Since each stage of an application has asymmetrical running times (as shown in Figure 3a), the number of containers needed at every stage would be different. The baseline RM is not aware of this disparity. However, *Fifer* is inherently stage-aware because it employs a proportional slack allocation policy. This, in turn, results in having similar batch sizes for the containers at every stage though they have disproportional execution times. Furthermore, the LM in *Fifer* estimates the queuing delay for every individual stage by continuously monitoring the load. This, in turn, aids in better stage-wise container scaling as opposed to uniformly scaling containers.

### 4.3 Function Scheduling

Apart from dynamically scaling the number of containers needed to host the requests per stage, we also need to design a scheduling policy to select the appropriate request from the queue of each stage. One important concern here is that a single application developer can host multiple types of applications from which some might share common functions (stages)\(^4\). In such cases, the request queue for shared stages will have queries from different applications where the available slack for each application will be different depending on the overall execution time of the application. Therefore, executing the requests in FIFO order will lead to SLO violations. To ensure SLOs of shared functions, we employ a Least Slack First (LSF) scheduling policy (shown in Algorithm 1\(^\circ\)). *Fifer* makes use of LSF such that it executes the application query with the least available slack from the queue at every stage. LSF helps to prioritize requests which have less slack and, at the same time, avoids starvation of requests in the queue. Since sharing microservices is not our primary focus in this work, we do not discuss the trade-offs involved in using other sharing specific scheduling policies.

### 4.4 Bin-Packing to increase Utilization

#### 4.4.1 Greedy Container Scheduling

In order to increase utilization, we need to ensure a minimal number of idle containers at any given point, which depends on the scheduling policy\(^\circ\). In *Fifer*, we design a scheduling policy such that, each stage will submit the request to the container with the least remaining free-slots where the number of free-slots is calculated using the container’s batch-size. In addition, we use a timeout period of 10 minutes to remove idle-containers which have not serviced any requests for that period. Hence, employing a greedy approach of choosing containers with the least-remaining free-slots (shown in Algorithm 1\(^\circ\)) as a scheduling candidate will in turn result in early scale-down of lightly loaded containers.

#### 4.4.2 Greedy Node Selection

The containers used to host functions are themselves hosted on servers, which could be VMs or bare-metal servers. In *Fifer*, similar to the function scheduling policy, we greedily schedule containers on the least-resource-available server. The servers are numbered from 1 to n where n is the number of available servers. We tune the cluster scheduler to assign containers to the lowest numbered server with the least available cores that can satisfy the CPU and memory requirement of the container. As a result, the unused cores will only be consuming idle power, and also the servers with all cores being idle can be turned after some duration of inactivity. Consequently, this can translate to potential savings in cluster energy consumption as a result of bin-packing all active containers on to fewer servers.

### 4.5 Proactive Scaling Policy

It is to be noted that, the queuing delay estimations and scaling based on runtime delay calculations would still lead to sub-optimal container spawning, especially if the future arrival rate is not known. Hence, in *Fifer*, we use a load prediction model\(^\circ\), which can accurately forecast the anticipated load for a given time interval. Using the predicted load, *Fifer* proactively spawns new containers at every stage.

As shown in Algorithm 1\(^\circ\), for every monitoring interval, we forecast the estimated number of requests based on past arrival rates. For each stage, if the current number of containers available is not sufficient to handle the forecast request load, *Fifer* proactively spawns additional containers. This proactive scaling policy is complementary to the dynamic reactive policy at each stage. If the prediction model can accurately predict the future load, then it would not

\(^4\)It is be noted that serverless platforms do not share microservices across tenants. Doing so would violate the security and isolation guarantees.
We have implemented a prototype of the proposed Fifer framework using open-source tools for evaluating the design with synthetic and real-world traces. The details of the implementation are described below.

5 Implementation and Evaluation

We have implemented a prototype of the proposed Fifer framework using open-source tools for evaluating the design with synthetic and real-world traces. The details of the implementation are described below.

5.1 Fifer Prototype Implementation

Fifer is implemented on top of Brigade [6] using 5KLOC of JavaScript. Brigade is an open-sourced serverless workflow management framework developed by Microsoft Azure [13]. Brigade can be deployed on top of a Kubernetes [9] cluster that handles the underlying mechanisms of pod(container) creation, orchestration, and scheduling. Brigade, by default, creates a worker pod for each job, which in turn handles
container creation and scheduling of tasks within the job and destroys the containers after job completion. Henceforth, we refer to a function chain as a “job” and stages within the job as “tasks”. To cater to Fifer’s design, we modify Brigade workers to persist the containers for every task after job completion such that they can be reused for scheduling tasks of other jobs. We implement a global request queue for every stage within the job which holds all the incoming tasks before being scheduled to a container in that stage. Each container has a local queue of length equal to the number of free-slots in the container.

We integrate a mongodb [27] database to maintain job-specific statistics (creationTime, completionTime, scheduleTime, etc) and container-specific metrics (lastUsedTime, batch size, etc), which can be periodically queried by the worker pod and load-balancer. As an offline step, for every function chain the following are added to the database, (a) the response latency, (b) the sequence of stages, (c) estimated execution time, and (d) the slack per stage (calculated as described in Section 4.1). Using these values, each container of a stage can then determine its free-slots.

**Pod Selection:** At runtime, the worker pod queries the database to pick a pod with the least number of free slots to schedule the task. Once a pod is selected, the task is added to its local-queue, and the free-slots of the pod are updated in the database. The same process is applied to every subsequent task of the job.

**Load Balancer:** We designed a Python daemon (1K LOC), which consists of a load monitor (LM) and a load predictor (LP). The LM periodically measures the queuing delay at the global queue of each stage and spawns additional containers if necessary (described in Section 4.2). The LP predicts the request-load using the LSTM model. The model was trained using Keras [28] and Tensorflow [15], over 100 epochs with 2 layers, 32 neurons, and batch size 1. The daemon queries the job_arrival information from the database, which is given as input to the prediction model. Recall that the details of the prediction were described in Section 4.5.

**Node/Server Selection:** In order to efficiently bin-pack containers into fewer nodes, we make modifications to the MostRequestedPriority scheduling policy in Kubernetes such that it always chooses the node with the least-available-resources to satisfy the Pod requirements. For our experiments, each container requires 0.5 CPU-core and memory within 1GB. Hence, we set the CPU limit for all containers to be 0.5. We determine idle cores in a node by calculating the difference between number of cores in a node and the sum of cpu-shares for all allocated pods in that node.

### 5.2 Large Scale Simulation

To evaluate the benefits of Fifer for large scale systems, we built a high-fidelity event-driven simulator using container cold-start latencies, loading times of container images and function transition times from our real-system counterpart.

<table>
<thead>
<tr>
<th>Hardware Configuration</th>
<th>Software Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td><strong>Ubuntu</strong></td>
</tr>
<tr>
<td>Xeon R Gold 6142</td>
<td>16.04</td>
</tr>
<tr>
<td><strong>Sockets</strong></td>
<td><strong>Kubernetes</strong></td>
</tr>
<tr>
<td>2</td>
<td>1.18.3</td>
</tr>
<tr>
<td><strong>Cores*threads</strong></td>
<td><strong>Docker</strong></td>
</tr>
<tr>
<td>16 x 2</td>
<td>19.04</td>
</tr>
<tr>
<td><strong>Clock</strong></td>
<td><strong>MongoDB</strong></td>
</tr>
<tr>
<td>2.8 Ghz</td>
<td>2.6.10</td>
</tr>
<tr>
<td><strong>DRAM</strong></td>
<td><strong>Python</strong></td>
</tr>
<tr>
<td>192 GB</td>
<td>3.6</td>
</tr>
<tr>
<td><strong>Tensorflow</strong></td>
<td>2.0</td>
</tr>
</tbody>
</table>

**Table 1. Hardware config.**

**Table 2. Software config.**

Using synthetic traces in both the simulator and the real-system, we verified the correctness of the simulator by comparing and correlating various metrics of interest.

### 5.3 Evaluation Methodology

We evaluate our prototype implementation on an 80 compute core Kubernetes cluster. We use one dedicated node as the head node. Each node is a Dell PowerEdge R740 server with Intel CascadeLake Xeon CPU host. The details of the single node hardware and software configuration are listed in Table 1 and 2. We use Kubernetes as the resource orchestrator. The mongodb database [27] and Python daemon reside on the head node. For energy measurements, we use an open-source version of Intel Power Gadget [11] measuring the energy consumed by all sockets in a node.

**Load Generator:** We use different traces which are given as input to the load generator. Firstly, we use synthetic Poisson-based request arrival rate with average arrival \( \lambda = 50 \). Secondly, we use real-world request arrival traces from Wiki [86] and WITS [1] (shown in Figure 7). As shown in Figure 7a, the WITS trace has a large variation in peaks (average=300req/s, peak=1200 req/s) when compared to the Wiki trace. The wiki trace (average= 1500 req/s) exhibits the typical characteristics of ML inference workloads, containing recurring patterns (e.g., hour of the day, day of the week), whereas the
WITS trace contains unpredictable load spikes (e.g., black-Friday shopping). Based on the peak request arrival rate, the simulation expands to match up to the capacity of a 2500 core cluster (30x our prototype cluster).

Each request is modeled after a query, which could be one among the four applications (microservice-chains), as shown in Table 4. Each application is compiled as a workflow program in Brigade, which invokes each microservice container in a sequence. The applications consist of well-known microservices derived from the Jinn&Tonic [46] benchmark suite (see Table 3). These include microservices from a diverse range of domains like image recognition, speech recognition, and language processing. All our microservices utilize Kaldi [74], Keras [28] and Tensorflow [15] libraries.

**Workload**: We model three different workload mixes by using a combination of two applications as shown in Table 5. Based on the increasing order of total available slack for each workload (avg. of both application’s slack), we categorize them into Heavy, Medium, and Light. Using the three workload-mix, we comprehensively analyze the scope of the benefits of Fifer for different proportions of available slack. The individual slacks for every application are shown in Table 4.

**Container Configuration**: All the microservices shown in Table 3 are containerized as “pods” in Kubernetes. We set the imagePullPolicy for each pod such that the container image will be pulled from dockerhub by default when starting a new container. This captures the behavior of serverless functions where function instances are loaded from external storage for every new container.

**Metrics and Resource Management Policies**: We evaluate our results by using the following metrics: (i) percentage of SLO violations, (ii) average number of containers spawned, (iii) median and tail latency of requests, (iv) container utilization, and (v) cluster-wide energy savings. The tail latency is measured as the 99th percentile request completion time in the entire workload. We compare these metrics for Fifer against Bline, SBatch and BPred resource-managers (RMs). Bline is the representative scheduler used in platforms like AWS, previously defined in Section 3. In SBatch, we set the batch size by equal-slack-division policy and fix the number of containers based on the average arrival rates of the workload traces. BPred is built on top Bline along with the LSF scheduling policy and the EWMA prediction policy. Note that this is a faithful implementation of scheduling and prediction policy as used in Archipelago [81], which does not support request batching. Further, to effectively compare the combined benefits of the individual components of Fifer, we do a brick-by-brick comparison of Fifer (a) only with dynamic scaling policy (RScale), and (b) combined with RScale and proactive provisioning. Both these variants employ the LSF job scheduling policy, as well as the greedy container/node selection policy. It is also to be noted that Fifer with RScale policy is akin to the dynamic batching policy employed in GrandSLAM [57].

### 6 Results and Analysis

#### 6.1 Real-System Prototype

We explain the results of our real-system prototype using the poisson request arrival trace, in this subsection.

**6.1.1 Minimizing Containers**: Figure 8a and Figure 8b show the percentage of SLO violations and average number of containers spawned for different RMs across all workloads. It is evident that Fifer spawns the least number of containers on average compared to all other schemes except SBatch. This is because SBatch does not scale containers based on changes in request load. However, this results in 15% more SLO violations for SBatch when compared to Fifer. The Bline and BPred RMs inherently over-provision containers due to their non-batching nature, thus minimizing SLO violations. But the BPred RM uses 20% lesser containers on average when compared to Bline due to proactive provisioning. In contrast, both Fifer and RScale batch jobs to reduce the number of containers being spawned. While RScale policy incurs 10% more SLO violations than Bline due to reactive-scaling when trying to minimize number of containers, Fifer does accurate proactive provisioning thus avoiding SLO violations. In short, Fifer achieves the best of both worlds by combining benefits of both batching and proactive scaling.

**6.1.2 Reduction in Latency**: Figure 10a plots the CDF of total response latency up to P95 for heavy workload-mix. The breakdown of P99 tail latency is plotted separately in Figure 9. We separate the response latency into execution time, cold-start induced delay, and batching induced...
delay. The batching induced delay is only for RScale and Fifer policies. It can be seen that the P99 is up to 3x higher for RScale and SBatch when compared to Bline and BPred. This is because aggressive batching and reactive scaling do not handle load variations, which leads to congestion in the request queues. The BPred policy has lesser P99 compared to RScale but in-turn it spawns 60% more containers than RScale. On the other hand, aggressive batching, along with proactive provisioning in Fifer only lead to 2x higher P99 latency than both Bline and BPred. Figure 9. It can be seen that the delay due to cold-starts is much lower for Fifer when compared to RScale. This is because the number of reactively scaled containers are much lesser owing to the accurate load estimation by Fifer’s prediction model.

Since both the RScale and Fifer RM enables batching of requests at each stage, the median latency of the requests is high compared to the Bline (shown in Figure 10a). However, Fifer utilizes the slack for requests at each stage, hence 99% requests complete within the given SLO, despite having increased median latency.

6.1.3 Breakdown of Improvements: The major sources of improvements in Fifer are (i) reduction of queuing delays and (ii) increased container utilization and better energy efficiency. We discuss the reasons for these improvements in detail below. The stage-wise results are plotted for IPA application from heavy workload mix. The results are similar for other applications as well.

Effects of Queuing: Figure 10b plots the queuing time distribution for heavy workload mix. It can be seen that the median queuing latency is high for Fifer (50-400ms), which indicates more requests are getting queued due to exploiting the slack of each stage. For the RScale scheme, the median queuing latency is higher than Fifer (500ms) because it leads to increased waiting times due to reactive spawning of containers with cold-starts. However, for both Bline and BPred, the latency distribution is irregular because the queuing latency will be higher or lower depending on the number of over-provisioned containers.

Stage-aware Batching and Scaleout: Figure 11 plots the stage-wise container distribution for all three stages. The execution time distribution for the stages was previously shown in Figure 3a. It can be seen that both Bline and BPred have more containers allocated for Stage-1 (ASR), which is the longest running stage (bottleneck) in the IPA application. However, the RScale scheme spawns slightly higher containers for Stage-1 (44%) and Stage-3 (35%) when compared to Baseline. This is because the proportionate slack allocation policy will evenly distribute the load across stages. Ideally, the distribution should be very close to SBatch, but reactive scaling of containers leads to many idle containers in each stage.

Fifer, on the other hand, spawns almost equal percentage of containers for Stage-2 (38%) and Stage-3 (36%). This is because, in addition to stage-aware batching, Fifer’s proactive container scaling policy reduces the aggressive reactive scaling of containers. The number of containers is less for Stage-2 (21%) because its a very short running stage (less than 2% of total execution time) and thereby results in early scale-in of idle containers. Though aggressive batching can result in SLO violations (15% and 12% more than Bline for SBatch and RScale respectively), Fifer ensures similar SLO violation as in Bline, because the LSTM model can well adapt to variations in arrival rate.

Container Utilization: Figure 12a plots the average number of tasks (requests) executed by a container in all stages.
We define container utilization as Requests executed per Container (RPC). It is evident that Fifer has the maximum RPC across all stages. Intuitively, for a given total number of requests, higher RPC indicates that a lesser number of containers are being spawned. It can be seen that both Bline and Bpred scheme always spawn a large number of containers due to non-batching nature, which is exacerbated, especially for short running stage-2 (RPC of 8.03\% and 11.67\%). Though both RScale and Fifer employ request-batching, Fifer still has 12.6\% better RPC on average across all stages than RScale. This is because Fifer inherently minimizes over-provisioned containers as a result of proactive container spawning.

To better understand the benefits of proactive provisioning, Figure 12b plots the overall the number of containers spawned measured over intervals of 10s for all four RMs. It can be seen that both RScale and Fifer adapt well to the request rate distribution, and due to batching they spawn up to 60\% and 82\% fewer containers on average when compared to Bline RM. Fifer is still 22\% better than RScale because Fifer can accurately estimate the number of containers required in each stage by using the LSTM prediction.

6.1.4 Cluster Energy Savings: Since in Fifer we effectively bin-pack containers to least-resources-available servers, it results in server consolidation thereby increasing the cluster efficiency. Figure 15 plots the cluster-wide energy as an average of energy consumed across all nodes in the cluster measured over intervals of 10 seconds for the entire workload duration. It can be seen that Fifer is 30.75\% more energy efficient than the Bline (for heavy workload-mix). This is because Fifer can accurately estimate the number of containers at each stage, thereby resulting in all active containers to get consolidated in fewer nodes. The energy savings are a result of non-active nodes only consuming idle power. Fifer is also 17\% more energy efficient than RScale, because proactive provisioning reduces the number of reactively spawned containers. This, in turn, results in increases in the number of idle CPU-cores in the cluster. Fifer is almost as energy efficient as Sbatch (difference of 4\%), but at the same time, it can scale up/down according to request demand, thus minimizing SLO violations when compared to Sbatch.

6.1.5 System Overheads: As meeting the response latency is one of the primary goals of Fifer, we characterize the system-level overheads incurred due to the design choices in Fifer. The mongodb database is a centralized server which resides on the head-node. We measure the overall average latency incurred due to all reads/writes in the database, which is well within 1.25ms. The LSF scheduling policy takes about 0.35ms on average per scheduling decision. The latency of LSTM request prediction which is not in the critical scheduling path and runs as a background process model is 2.5 ms on average. The time taken to spawn new container, including fetching the image remotely takes about 2s to 9s depending on the size of the container image.

6.2 Trace driven Simulation

This is because the Wiki trace follows a diurnal pattern with a highly dynamic load, thus leading to many unprecedented request scale-out and consequently requires more containers to be spawned. Since the Bl ine, Bpred and RScale RMs employ reactive scaling, they experience higher average number of containers spawned (shown in Figure 13b). Especially, the RScale and Bpred RM spawns up to 3.5\times more containers on average compared to Fifer, still leading to 5\% more SLO violations than Fifer (shown in Figure 13a). This is because it cannot predict the variations in the input load. Fifer, on the other hand, is resilient to load fluctuations as it utilizes an LSTM-based load prediction model that can accurately capture the variations and proactively spawn containers. The tail latencies are also high for the RScale RM, due to the congestion of request queues resulting from cold-starts (shown in Figure 14a and 14a). However, the median latencies follow increasing trends, as observed in the real-system.

Figure 14 plots the percentage of SLO violations and average containers spawned normalized to Bl ine, for WITS trace for all three workloads. The WITS trace exhibits sudden peaks due to a higher peak-to-median ratio in arrival rates (the peak (1200 req/s) is 5\times higher than the median (240 req/s)). This sudden surge leads to very high tail latencies (shown in Figure 14c and 14c). Fifer can still reduce tail-latencies by up to 66\% when compared to Sbatch and RScale policies. The amount of SLO violations (shown in Figure 13c) are considerably lower for all policies in comparison to Wiki trace, due to less dynamism in the arrival load. However, Fifer still spawns 7.7\times, 2.7\times fewer containers on average (Figure 13d), when compared to the Bpred and RScale RMs, respectively. The savings of Fifer with respect to RScale are lower when compared to WIKI trace, because the need to spawn additional containers by reactive scaling is considerably reduced when there less frequent variations in arrival rates. Similar to the real-system, Fifer ensures SLO’s to the same degree (up to 98\%) as Bl ine and Bpred RMs.

6.2.1 Effect of Coldstarts Figure 16 plots the number of cold-starts incurred by three different RMs for a 2 hour snapshot of both traces. It can be seen that, Fifer reduces the number of cold-starts by up to 7\times and 3.5\times, when compared to Bpred for the Wiki and Wits trace, respectively. Though RScale also significantly reduces cold-start when
compared to Bline and BPred. Fifer is still 3× better than RScale. This is because Fifer avoids a large number of reactive cold-starts by accurately spawning containers in advance. It should also be pointed out that the number of cold-starts are more for WIKI trace because the average request rate is 5× higher than WITS trace.

7 Related Work

Managing Microservice Chains: The most relevant and recent prior works to ours that have looked into resource management for microservice chains can be summarized as follows: (i) Grandslam [57] proposes a dynamic slack-aware scheduling policy for guaranteeing SLOs in shared microservice execution frameworks. Unlike Grandslam, Fifer focuses on container provisioning combined with container scalability, in the context of RMs used in serverless computing frameworks. As demonstrated by our results, Grandslam (RScale policy) suffers from SLO violations, while scaling containers due to dynamic load variations. (ii) Archipelago [81] aims to provide low latency scheduling for both monolithic and chained microservices. In contrast, Fifer specifically focuses on optimizing microservice-chains by exploiting batching, with the primary objective of increasing resource utilization, without compromising on SLOs. Table 6 provides a comprehensive analysis of all the features of Fifer, comparing it with other relevant works.

Resource Management in cloud: We discuss the resource management policies used in existing cloud platforms for both private and public cloud.

• Private Cloud: A large body of work [33–35, 58, 71] have looked at ensuring QoS guarantees for latency critical applications by developing sub-millisecond scheduling strategies using both distributed and hybrid schedulers. Some works [24, 45, 85] employ prediction-based scheduling in RMs for executing latency-critical tasks that are co-located with batch tasks. However, these techniques are specifically geared for conventional monolithic applications. Prior works [90, 93] propose stage-aware resource management techniques but they cannot be applied to current trends where there are thousands of millisecond scale functions. However, some of the policies proposed w.r.t queuing and reactive scaling can co-exist with Fifer’s policies.

• Public Cloud: There are several research works that optimize for the resource provisioning cost in the public cloud. These works broadly fall into two categories: (i) tuning the auto-scaling policy based on changing needs (e.g., Spot, On-Demand) [17, 29, 42, 44, 45, 78, 91], (ii) predicting peak loads and offering proactive provisioning based auto-scaling policy [42–44, 63, 79, 97]. Fifer uses similar load prediction models and auto-scales containers but with respect to serverless function chains. Swayam [42] is relatively similar to our work such that, it handles container provisioning along with load-balancing. Unlike Fifer which looks at micro-service chains, Swayam is specifically catered for single-function machine learning inference services.

Exploiting Slack: Exploiting slack between tasks is a well-known technique, which has been applied in various domains of scheduling, including SSD controllers [31, 37], memory controllers [41, 67, 77, 89, 96], and network-on-chip [32, 62, 72]. In contrast to exploiting slack, we believe the novelty aspect lies in identifying the slack in relevance to the problem.
Comparing the features of Fifer with other state-of-the-art resource management frameworks.

domain and designing policies to utilize the slack effectively.

Mitigating Cold-starts: Many recent works [16, 19, 69] propose optimizations to reduce container setup overheads. For example, SOCK [69] and SAND [16] explore optimizations to reduce language framework-level overheads and the step function transition overheads, respectively. Some works propose to entirely replace containers with new virtualization techniques like Firecracker [20] and uni-kernels [99]. Complementary to these approaches, Fifer tries to decouple container cold-starts from execution time.

8 Discussion and Future Work

Design Limitations: We set SLO to be within 1000ms, which is the typical user-perceived latency. Note that changing the SLO would result in different slacks for application stages. While providing execution time and SLO information is an offline step in Fifer, for longer running applications where execution time is greater than 50% of SLO, the benefits of batching would be significantly reduced. Our execution time estimates are limited to ML-based applications, but our schemes can be applied to all other applications which have predictable execution times. In addition, based on the type of applications hosted on serverless platforms, the provider can use a combination of tailor-made policies specific to each class of applications. Also, the applications we consider are linearly chained without any dynamic branches. We plan the explore dynamic microservice chains in future work.

Our design policies are implemented on Brigade which is deployed on top of Kubernetes resource orchestrator. The proposed policies can be readily ported to other open-sourced resource management frameworks because of the inherent design choices are that are readily pluggable with minimal modifications of API calls.

All decisions related to container scaling, scheduling and load-prediction are reliant on the centralized database which can become a potential bottleneck in terms of scalability and consistency for a large scale system. This can be mitigated by using fast distributed solutions like Redis [22] and Zookeeper respectively [53]. The LSTM model in Fifer is pre-trained using 60% of the arrival trace. In case different load patterns, the LSTM model parameters can be constantly updated by retraining in the background with new arrival rates.

Cloud Provider Support: The cold-start measurements and characterizations in Fifer are mainly based on AWS. However, our main design can be extended in theory to other major cloud platform as well. We also rely on the platform provider to expose API’s for the tenants to specify their application SLO requirements which are crucial for slack estimation. Such an API would better enable the provider to auto-configure tenants’ execution environments, which would be invaluable in improving resource efficiency [36].

9 Concluding Remarks

There is wide-spread prominence in the adoption of serverless functions for executing microservice-based applications in the cloud. This introduces critical inefficiencies in terms of scheduling and resource management for the cloud provider, especially when deploying a large number millisecond-scale latency-critical functions. In this paper, we propose and evaluate Fifer, a stage-aware adaptive resource management framework for efficiently running function-chains on serverless platforms by ensuring high container utilization and cluster efficiency without compromising on SLOs. Fifer makes use of a novel combination of stage-wise slack awareness along with proactive container allocations using an LSTM-based load prediction model. The proposed technique can intelligently scale-out and balance containers for every individual stage. Our experimental analysis on an 80 compute-core cluster and large scale simulations show that Fifer spawns up to 80% fewer containers on average, thereby improving container utilization by 4x and cluster efficiency by 31%.

Acknowledgments

We are indebted to the anonymous reviewers. Anup Sarma, Cyan Mishra, Sandeepa Bhuyan, and Ram Srivatsa for their insightful comments. We thank our shepherd Dejan Milojicic for his guidance in revising the paper. This research was partially supported by NSF grants #1931531, #1955815, #1629129, #1763681, #1629915, #1908793, #1526750 and we thank NSF Chameleon Cloud project CH-819640 for their generous compute grant. All product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References


