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.

Middleware '20, December 7–11, 2020, Delft, Netherlands
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8153-6/20/12…$15.00
https://doi.org/10.1145/3423211.3425683

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

Keywords: serverless, resource-management, scheduling, queuing

ACM Reference Format:
Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C. Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. 2020. Fifer: Tackling Resource Underutilization in the Serverless Era. In *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3423211.3425683

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 microservices[1], 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

[1] A 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, *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. *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. *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 *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. *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 *Fifer* with a novel *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 *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 *Fifer* spawns up to 80% fewer containers on an average, thereby improving container utilization and cluster-wide energy savings by up to 4× 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 *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 *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 functions-chains by being proactive and stage-aware.

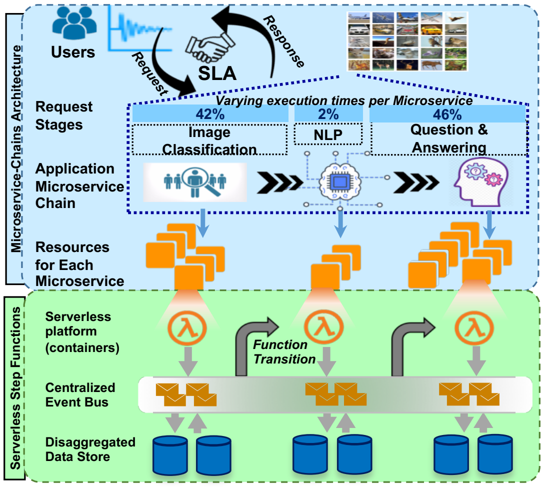

Microservice-Chains Architecture
Users
Request
SLA
Response
Request Stages
Varying execution times per Microservice
42%
Image Classification
2%
NLP
46%
Question & Answering
Application Microservice Chain
Resources for Each Microservice
Serverless Step Functions
Serverless platform (containers)
Function Transition
Centralized Event Bus
Disaggregated Data Store

Figure 1. The blue and green box show the architectures of a typical microservice-chain and a serverless function respectively.

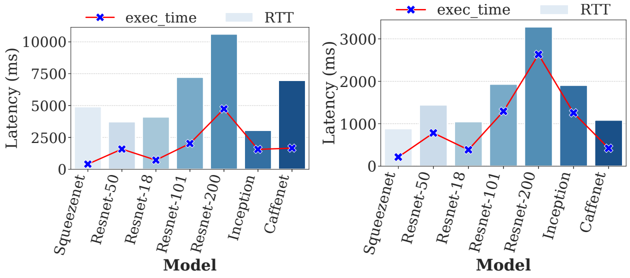
of communication events over a centralized event bus. Due to the stateless natures of serverless functions, input data such as pre-trained models, etc., need to be retrieved from ephemeral data storage like AWS S3 [73]. For further details on serverless functions, we refer the reader to prior works [23, 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].

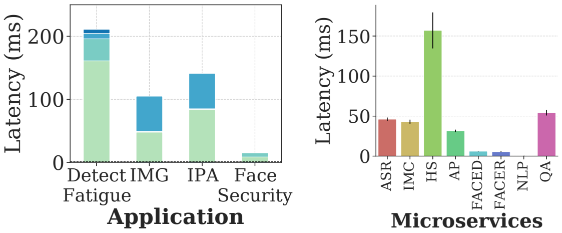


(a) Cold Start Latency. **(b)** Warm Start Latency.

Figure 2. Implications of cold-starts for inference application.

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 autoscaler 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) Per-stage breakdown of overall application execution times.

(b) Variation of execution time for each microservice.

Figure 3. Characterization of Microservices for a fixed input size from Djinn&Tonic Benchmark Suite.

2.2.2 What is different with function-chains?

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 every 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 staged 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.

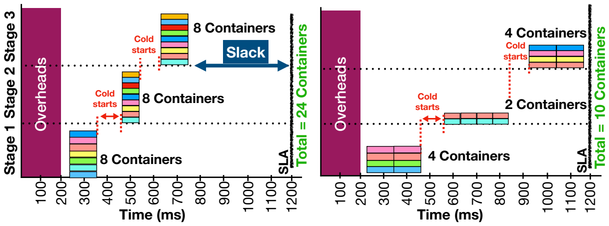
To effectively exploit this per-stage variability described above, there are two assumptions: (i) the execution times for each stage of an application has to be known apriori, and (ii) the execution times should be predictable and not have large variations. The first assumption can be held true for serverless 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 suite [46]. 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 eg, 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



(a) Baseline RM (b) Request-Batching RM

Figure 4. An example representation showing the working of Baseline vs Stage-Aware queuing enabled RM frameworks.

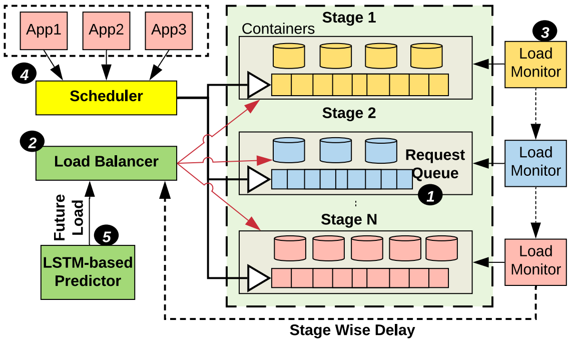


Figure 5. An overview of *Fifer*.

into 10 containers and the number of requests queued per container at every stage depends on the execution time and available slack at each stageAdditional containers would be spawned if the arrival rate increases.

It is important to note that queuing and batching still cannot help in hiding the cold-start latencies encountered (shown in Figure 4) when spawning new containers. The cold-start delays are accentuated especially when there is dynamism in request rate. While cold-start latencies can be reduced by OS-level optimizations [66], the only way to hide them entirely is by proactively spawning containers. Balancing the aggressiveness of proactively spawning new containers and queuing requests at existing ones is crucial for achieving high container utilization and low SLOs, hence this act hinges solely on the prediction model adopted.

Key takeaway: *Request queuing and batching can minimize containers spawned while avoiding SLO violations, but cannot hide cold-starts. SLO violations caused by cold starts can be avoided by provisioning containers in advance by predicting the future load, but reaping the benefits is contingent upon the accuracy of the prediction model used.*

4 Overall Design of Fifer

Motivated by our observations, we design a novel RM framework *Fifer* to manage function-chains on serverless platforms efficiently. Figure 5 depicts the high-level design of *Fifer*. User applications send requests (function triggers) to the system which are handled by the cluster scheduler. The scheduler dispatches the requests to queues of different stages. From there, requests make it to the containers depending on the available free slots. Every stage has a load monitor to keep track of the request load and periodically sends updates to the load balancer. The load balancer decides the scale out factor (#containers) of every stage based on the predicted load and existing queuing delays for each stage. The key components of the design are explained in detail in the below subsections using circled annotations.

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 *Djinn&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 ①. We design a load balancer (LB) ② along with a load monitor that are integrated to each stage (LM) ③ 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:

$$L = \sum_{i=1}^{N} Bsize_i, \qquad T_d = PQ_{len} \times S_r, \qquad D_f = \frac{T_d}{L}$$

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ⓑ). *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 ④. 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ⓓ) 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 ⑤, 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ⓔ, 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

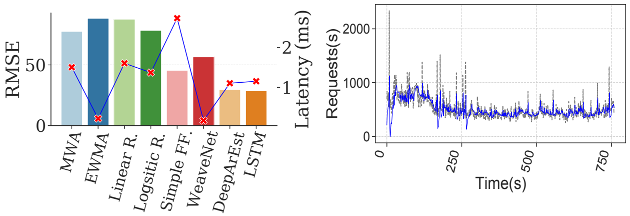
[4] It is be noted that serverless platforms do not share microservices across tenants. Doing so would violate the security and isolation guarantees.

Algorithm 1 Stage_Aware + Slack_Aware + Prediction

```
procedure DYNAMIC_REACTIVE_SCALING(STAGES) ⓐ
    for stage in ∀Stages do
        delay ← Calculate_Delay(last_10s_jobs)
        if delay ≥ stage.slack then
            est_containers ← Estimate_Container()
            stage.containers.append(est_containers)
        end if
    end for
end procedure
procedure ESTIMATE_CONTAINERS(STAGE,PQ_LEN) ⓑ
    total_delay ← PQ_len ∗ stage.resp_latency
    current_req ← len(stage.containers) ∗ batchSize
    delay_factor ← total_delay/current_req
    if delay_factor ≥ cold_start then
        est_containers ← (PQ_LEN - current_req)
        returnest_containers
    end if
end procedure
procedure SCHEDULE_TASK(STAGE)
    Q ← Stage.Queue
    task ← find_min(Q.tasks.slack) ⓒ
    container ← greedy_find_worker(Stage.containers) ⓓ
    execute_task(task,container)
end procedure
procedure PREDICTIVE_STAGE_AWARE(STAGES) ⓔ
    load ← Measure_Load(last_100_jobs)
    for stage in ∀Stages do
        batchSize ← stage.batchSize
        current_req ← len(stage.containers) ∗ batchSize
        Fcast ← LSTM_Predict(load)
        if Forecast ≥ current_requests then
            est_containers ← (Fcast - current_req)
            est_containers ← est_containers / batchSize)
            stage.containers.append(containers_needed)
        end if
    end for
end procedure
```

result in SLO violations as the necessary number of containers would be spawned in advance. However, in the case of mispredictions, the reactive policy would detect delays at the respective stages and spawn additional containers with cold-starts. We next explain in detail the prediction model used in *Fifer.*

To effectively capture the temporal nature of request arrival scenario in date-centers, we make use of a Long Short Term Memory Recurrent Neural Network (LSTM) model [51]. LSTMs are known to provide a state-of-the-art performance for many popular application domains, including Stock Markets forecast and language processing. For a periodic monitoring interval (T) of 10s, *Fifer* samples the arrival rate in adjacent windows of size W_s (5s) over the past 100 seconds. It keeps track of the maximum arrival rate at each window and calculates the global maximum arrival rate. Using this global arrival rate, the model predicts the number of containers as a maximum in the future window of size W_p. The interval (T) is set to 10 seconds since average container start-up latency ranges between 1s and 10s. The prediction window (W_p) is set to 10 minutes since 10 minutes of future trend is sufficient to expose the long term trade-offs. Short-term load fluctuations would still be captured within the 10s interval.



(a) RMSE and Latency(ms).

(b) Prediction accuracy of LSTM model.

Figure 6. Comparing different prediction models.

4.5.1 Prediction Model Design. We also quantitatively justify the choice of using LSTM by a doing a brick-by-brick comparison of the trade-offs of using different non-ML based and ML-based models on a given input arrival trace. We use four non-ML models, namely Moving Window Average (MWA), Exponential Weight Moving Average(EWMA), Linear Regression (Linear R.), and Logistic Regression (Logisitic R.). These models are continuously fitted over requests in last t-100 seconds for every T. In addition, we use four ML models (Simple Feed Forward Network, WeaveNet, DeepAREstimator and LSTM) that are pre-trained with 60% of the WITS [1] arrival trace input as the training set. We employ a time-step based prediction on the ML models as described in the above sub-section. Figure 6a plots the Root Mean Squared Error (RMSE) and latency incurred by 8 different prediction models. It can be seen that LSTM has the least RMSE values. To verify the same, we plot the accuracy of the LSTM model for WITS trace (Figure 6b). It is evident that the model predicts requests accurately (85% from our experiments) for the given test set duration of 800s.

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

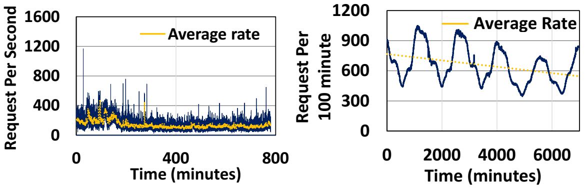
To evaluate the benefits of *Fifer* for large scale systems, we built a high-fidelity event-driven simulator using container cold-start latencies, loading times of container images and function transition times from our real-system counterpart. Using synthetic traces in both the simulator and the real-system, we verified the correctness of the simulator by comparing and correlating various metrics of interest.

Hardware	Configuration
CPU	Xeon R Gold-6242
Sockets	2
Cores(×)threads	16 × 2
Clock	2.8 Ghz
DRAM	192 GB

Table 1. Hardware config.

Software	Version
Ubuntu	16.04
Kubernetes	1.18.3
Docker	19.04
MongoDB	2.6.10
Python	3.6
Tensorflow	2.0

Table 2. Software config.



(a) WITS Trace.

(b) Wiki Trace.

Figure 7. Job Request Arrival Traces.

Domain	ML application	ML Model	Avg. Exec Time (ms)
Images Services	Image Classification (IMC)	Alexnet	43.5
	Human Activity Pose (AP)	DeepPose	30.3
	Human Segmentation (HS)	VGG16	151.2
	Facial Recognition (FACER)	VGGNET	5.5
	Face Detection (FACED)	Xception	6.1
Speech Services	Auto Speech Recognition (ASR)	NNet3	46.1
Natural Language Processing	Parts of Speech Tagging (POS)	SENNA	0.100
	Name Entity Recognition (NER)	SENNA	0.09
	Question Answering (QA)	seq2seq	56.1

Table 3. Description of Microservices (Functions) used in *Fifer*.

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

Application Type	Microservice-chain	Avg Slack(ms)
Face Security	FACED ⇒ FACER	788
IMG	IMC ⇒ NLP ⇒ QA	700
IPA	ASR ⇒ NLP ⇒ QA	697
Detect-Fatigue	HS ⇒ AP ⇒ FACED ⇒ FACER	572

Table 4. Microservice-Chains and their slack.

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 (30× our prototype cluster).*

Each request is modelled 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 *Djinn&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.

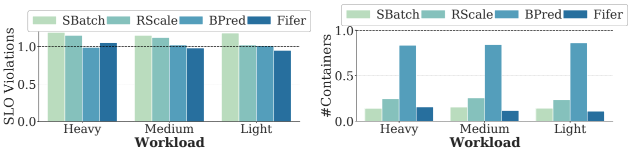
Workload: We model three different workload mixes by using a combination of two applications as shown in Table 5. Based on the increasing order of total available slack for each workload (avg. of both application's slack), we categorize them into *Heavy*, *Medium*, and *Light*. Using the three workload-mix, 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.

Workload	Query Mix
Heavy	IPA, DETECT-Fatigue
Medium	IPA, IMG
Light	IMG, FACE-Security

Table 5. Workload Mix.

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 behaviour 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 99^{th} percentile request completion times 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].



(a) SLO violations norm. to `Bline`. **(b)** Containers norm. to `Bline`.

Figure 8. *Fifer* Prototype: Comparing SLO violations with number of containers spawned.

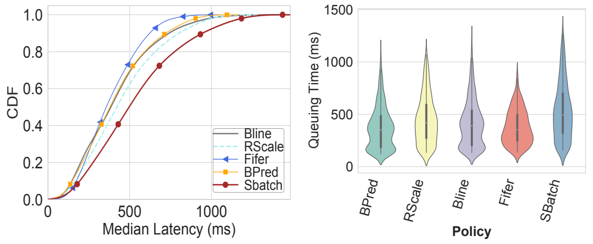
6 Results and Analysis

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



(a) Latency Distribution up to P95. **(b)** Queuing time distribution.

Figure 10. Queuing time and response latency distribution for heavy workload-mix.

delay. The batching induced delay is only for *RScale* and *Fifer* policies. It can be seen that, the P99 is up to 3× 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 2× 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.

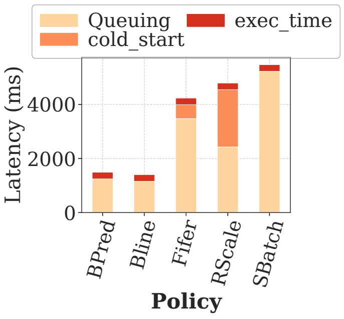


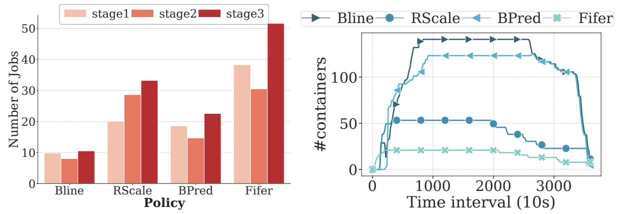
Figure 9. P99 Tail Latency.

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* RM, the latency distribution is irregular because the queuing latency will be higher or lower depending on the number of over-provisioned containers.



(a) Average number of jobs executed per container (JPC) **(b)** Cumulative number of Containers spawned over time.

Figure 12. Sources of Improvement.

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.

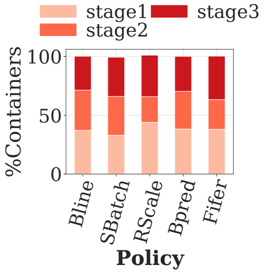


Figure 11. Distribution of Containers across stages of IPA application.

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 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*.

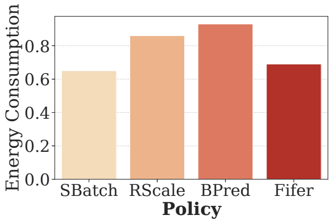


Figure 15. Cluster-wide energy savings normalized to `Bline`.

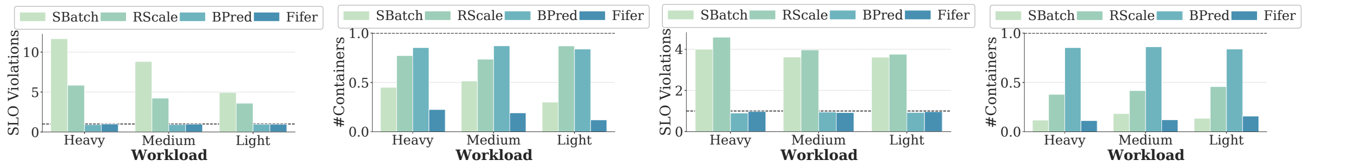
6.1.5 System Overheads: As meeting the response latency is one of the primary goals of *Fifer*, we characterize the system-level overheads incurred due to the design choices in *Fifer*. The *mongodb* database is a centralized server which resides on the head-node. We measure the overall average latency 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 *Bline*, *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× 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.

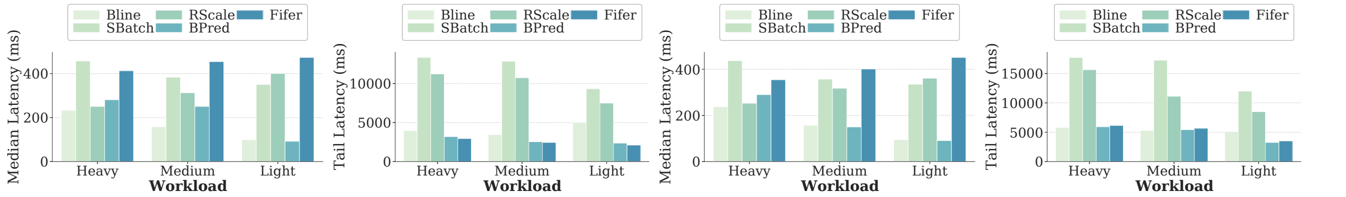
Figure 14 plots the percentage of SLO violations and average containers spawned normalized to *Bline*, for WITS trace for all three workloads. The WITS trace exhibits sudden peaks due to a higher peak-to-median ratio in arrival rates (the peak (1200 req/s) is 5× higher than the median (240 req/s)). This 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×, 2.7× 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 *Bline* 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× and 3.5×, when compared to *Bpred* for the Wiki and Wits trace, respectively. Though *RScale* also significantly reduces cold-start when



(a) WIKI: SLO violations. **(b)** WIKI: Average #Containers. **(c)** WITS: SLO violations. **(d)** WITS: Average #Containers.

Figure 13. SLO violations and average number of containers for Wikipedia and WITS request arrival trace. Results normalized to `Bline`.



(a) WIKI: Median Latency. **(b)** WIKI: Tail Latency. **(c)** WITS: Median Latency. **(d)** WITS: Tail Latency.

Figure 14. Median and Tail Latency for Wikipedia and WITS request arrival trace.

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.

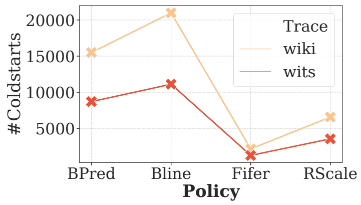


Figure 16. Number of Coldstarts.

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

Features	Grandslam [57]	Power-chief [95]	Time-Trader [87]	Parties [25]	MArk [97]	Archipelago [81]	Swayam [42]	*Fifer*
Server consolidation	✓	✓	✓	✗	✓	✗	✓	✓
SLO Guarantees	✓	✗	✓	✓	✓	✓	✓	✓
Function Chains	✓	✓	✗	✗	✗	✓	✗	✓
Slack based scheduling	✓	✓	✓	✓	✗	✓	✗	✓
Slack aware batching	✓	✗	✗	✗	✗	✗	✗	✓
Energy Efficient	✗	✓	✓	✗	✗	✗	✓	✓
Autoscaling Containers	✗	✓	✗	✗	✓	✓	✓	✓
Request Arrival prediction	✗	✗	✗	✗	✗	✓	✓	✓

Table 6. 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 Zookeper 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 4× 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

[1] 2013. WITS: Waikato Internet Traffic Storage. https://wand.net.nz/wits/index.php.

[2] 2016. Apache Openwhisk. https://openwhisk.apache.org/.

[3] 2020. AWS Lambda. Serverless Functions. https://aws.amazon.com/lambda/.

[4] 2020. Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable.

[5] 2020. Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable.

[6] 2020. Brigade-workflows. https://brigade.sh/.

[7] 2020. IBM-Composer. https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-pkg_composer.

[8] 2020. IBM Serverless Functions. https://www.ibm.com/cloud/functions.

[9] 2020. Kubernetes. https://kubernetes.io/.

[10] 2020. Microsoft Azure Serverless Functions. https://azure.microsoft.com/en-us/services/functions/.

[11] Feb 24, 2020. Intel Power Gadget. https://github.com/sosy-lab/cpu-energy-meter.

[12] February 2018. Google Cloud Functions. https://cloud.google.com/functions/docs/.

[13] March 28,2019. Brigade-azure. https://cloudblogs.microsoft.com/opensource/2019/03/28/announcing-brigade-1-0-new-kind-of-distributed-application/.

[14] May 11,2020. Fission-workflows. https://docs.fission.io/docs/workflows/.

[15] Martín Abadi. 2016. TensorFlow: learning functions at scale. In *Acm Sigplan Notices*. ACM.

[16] Istemi Ekin Akkus et al. 2018. SAND: Towards High-Performance Serverless Computing. In *ATC*.

[17] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. 2019. BurScale: Using Burstable Instances for Cost-Effective Autoscaling in the Public Cloud. In *Proceedings of the ACM Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, USA.

[18] Marian Stewart Bartlett, Gwen Littlewort, Mark Frank, Claudia Lainscsek, Ian Fasel, and Javier Movellan. 2005. Recognizing facial expression: machine learning and application to spontaneous behavior. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, Vol. 2. IEEE, 568–573.

[19] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2018. Putting the "Micro" Back in Microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 645–650.

[20] Marc Brooker, Andreea Florescu, Diana-Maria Popa, Rolf Neugebauer, Alexandru Agache, Alexandra Iordache, Anthony Liguori, and Phil Piwonka. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*.

[21] Jyothi Prasad Buddha and Reshma Beesetty. 2019. Step Functions. In *The Definitive Guide to AWS Application Integration*. Springer.

[22] Josiah L Carlson. 2013. *Redis in action*. Manning Publications Co.

[23] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. 2017. Serverless Programming (Function as a Service). In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2658–2659.

[24] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. *SIGARCH Computer Architecture News* (2016).

[25] Shuang Chen, Christina Delimitrou, and Jose F. Martinez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *ASPLOS*.

[26] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* (2015).

[27] Kristina Chodorow. 2013. *MongoDB: the definitive guide: powerful and scalable data storage.* " O'Reilly Media, Inc.".

[28] Francois Chollet. 2018. *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. MITP-Verlags GmbH & Co. KG.

[29] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. 2018. Stratus: Cost-aware Container Scheduling in the Public Cloud. In *SoCC*.

[30] Amazon Elastic Compute Cloud. 2011. Amazon web services. *Retrieved November* (2011).

[31] J. Cui, Y. Zhang, W. Wu, J. Yang, Y. Wang, and J. Huang. 2018. DLV: Exploiting Device Level Latency Variations for Performance Improvement on Flash Memory Storage Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 8 (2018), 1546–1559.

[32] Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita Das. 2010. Aergia: A network-on-chip exploiting packet latency slack. *IEEE micro* 31, 1 (2010), 29–41.

[33] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2016. Job-aware Scheduling in Eagle: Divide and Stick to Your Probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*.

[34] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. 2015. Hawk: Hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 499–510.

[35] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. 2015. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (Kohala Coast, Hawaii) *(SoCC '15)*. ACM, New York, NY, USA.

[36] Vojislav Dukic and Ankit Singla. 2019. Happiness index: Right-sizing the cloud's tenant-provider interface. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA.

[37] Nima Elyasi, Mohammad Arjomand, Anand Sivasubramaniam, Mahmut T. Kandemir, Chita R. Das, and Myoungsoo Jung. 2017. Exploiting Intra-Request Slack to Improve SSD Performance. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Yunji Chen, Olivier Temam, and John Carter (Eds.). ACM, 375–388.

[38] Lang Feng, Prabhakar Kudva, Dilma Da Silva, and Jiang Hu. 2018. Exploring Serverless Computing for Neural Network Training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*.

[39] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *ATC*.

[40] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[41] Mrinmoy Ghosh and Hsien-Hsin S Lee. 2007. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked DRAMs. In *40th Annual IEEE/ACM international symposium on microarchitecture (MICRO 2007)*. IEEE, 134–145.

[42] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. 2017. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. In *USENIX Middleware Conference*.

[43] Rui Han, Moustafa M. Ghanem, Li Guo, Yike Guo, and Michelle Osmond. 2014. Enabling Cost-Aware and Adaptive Elasticity of Multi-Tier Cloud Applications. *Future Gener. Comput. Syst.* 32, C (March 2014), 82–98.

[44] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R. Ganger, and Phillip B. Gibbons. 2018. Tributary: spot-dancing for elastic services with latency SLOs. In *ATC*.

[45] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. 2017. Proteus: Agile ML Elasticity Through Tiered Reliability in Dynamic Resource Markets. In *Eurosys*.

[46] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. 2015. DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers. In *ISCA*.

[47] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G. Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. 2015. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers. In *ASPLOS*.

[48] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 620–629.

[49] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. *arXiv preprint arXiv:1812.03651* (2018).

[50] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association.

[51] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* (1997).

[52] John A. Hoxmeier and Chris Dicesare. 2000. System Response Time and User Satisfaction: An Experimental Study of Browser-based Applications. In *AMCIS*.

[53] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8.

[54] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

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

[56] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized Core-granular Scheduling for Serverless Functions. In *SoCC*.

[57] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *EuroSys*.

[58] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 485–497.

[59] Nima Kaviani, Dmitriy Kalinin, and Michael Maximilien. 2019. Towards Serverless as Commodity: a case of Knative. In *Proceedings of the 5th International Workshop on Serverless Computing*. 13–18.

[60] Abeer Abdel Khaleq and Ilkyeun Ra. 2018. Cloud-Based Disaster Management as a Service: A Microservice Approach for Hurricane Twitter Data Analysis. In *GHTC*.

[61] Ron Kohavi and Roger Longbotham. 2007. Online Experiments: Lessons Learned. *IEEE Computer* (2007).

[62] A. Kostrzewa, S. Saidi, and R. Ernst. 2016. Slack-based resource arbitration for real-time Networks-on-Chip. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1012–1017.

[63] Adithya Kumar, Iyswarya Narayanan, Timothy Zhu, and Anand Sivasubramaniam. 2020. The Fast and The Frugal: Tail Latency Aware Provisioning for Coping with Load Variations. In *Proceedings of The Web Conference 2020 (WWW '20)*. Association for Computing Machinery, New York, NY, USA.

[64] Tony Mauro. 2015. Adopting microservices at netflix: Lessons for architectural design. *Recuperado de https://www. nginx. com/blog/microservices-at-netflix-architectural-best-practices* (2015).

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

[66] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. In *HotCloud 19*. USENIX.

[67] Nachiappan Chidambaram Nachiappan, Haibo Zhang, Jihyun Ryoo, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut T Kandemir, Ravi Iyer, and Chita R Das. 2015. VIP: virtualizing IP chains on handheld platforms. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 655–667.

[68] Y. Niu, F. Liu, and Z. Li. 2018. Load Balancing Across Microservices. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 198–206.

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

[70] Matthew Obetz, Stacy Patterson, and Ana Milanova. 2019. Static Call Graph Construction in AWS Lambda Serverless Applications. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA.

[71] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 69–84.

[72] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. 2019. Opportunistic computing in gpu architectures. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 210–223.

[73] V. Persico, A. Montieri, and A. Pescape. 2016. On the Network Performance of Amazon S3 Cloud-Storage Service. In *Cloudnet*.

[74] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, et al. 2011. The Kaldi Speech Recognition Toolkit. In *ASRU*.

[75] P. V. Rengasamy, H. Zhang, S. Zhao, N. C. Nachiappan, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das. 2018. CritICs Critiquing Criticality in Mobile Apps. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 867–880.

[76] Research and Markets. 2017. Function-as-a-Service Market by User Type (Developer-Centric and Operator-Centric), Application (Web and Mobile Based, Research and Academic), Service Type, Deployment Model, Organization Size, Industry Vertical, and Region - Global Forecast to 2021. In *Research and Markets*.

[77] Anup Sarma, Huaipan Jiang, Ashutosh Pattnaik, Jagadish Kotra, Mahmut Taylan Kandemir, and Chita R Das. 2019. CASH: compiler assisted hardware design for improving DRAM energy efficiency in CNN inference. In *Proceedings of the International Symposium on Memory Systems*. 396–407.

[78] Prateek Sharma, David Irwin, and Prashant Shenoy. 2017. Portfolio-Driven Resource Management for Transient Cloud Servers. *Proc. ACM Meas. Anal. Comput. Syst.* 1, 1, Article 5 (June 2017), 23 pages.

[79] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. 2015. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer*

Systems. 1–15.

[80] Eismann Simon and Scheuner Joel. May 29, 2020. A Review of Serverless Use Cases and their Characteristics. *SPEC-RG-2020-5* (May 29, 2020). https://research.spec.org/news/single-view/article/technical-report-on-a-review-of-serverless-use-cases-and-their-characteristics-published.html.

[81] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2019. Archipelago: A Scalable Low-Latency Serverless Platform. *arXiv preprint arXiv:1911.09849* (2019).

[82] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. arXiv:2001.04592 [cs.DC]

[83] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. 2019. Softsku: Optimizing server architectures for microservice diversity@ scale. In *ISCA*.

[84] Amoghvarsha Suresh and Anshul Gandhi. 2019. FnSched: An Efficient Scheduler for Serverless Functions. In *Workshop on Serverless Computing*.

[85] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das. 2019. Kube-Knots: Resource Harvesting through Dynamic Container Orchestration in GPU-based Datacenters. In *CLUSTER*.

[86] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. 2009. Wikipedia workload analysis for decentralized hosting. *Computer Networks* (2009).

[87] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T. N. Vijaykumar. [n.d.]. TimeTrader: exploiting latency tail to save datacenter energy for online search. In *MICRO 2015*.

[88] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang. 2016. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In *CCGrid*.

[89] Vivek Pandey, W. Jiang, Y. Zhou, and R. Bianchini. 2006. DMA-aware memory energy management. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.* 133–144.

[90] Rob Von Behren, Jeremy Condit, Feng Zhou, George C Necula, and Eric Brewer. 2003. Capriccio: scalable threads for internet services. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 268–281.

[91] Cheng Wang, Bhuvan Urgaonkar, Neda Nasiriani, and George Kesidis. 2017. Using Burstable Instances in the Public Cloud: Why, When and How? *SIGMETRICS* (June 2017).

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

[93] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada) *(SOSP '01)*. Association for Computing Machinery, New York, NY, USA, 230–243. https://doi.org/10.1145/502034.502057

[94] C. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 331–344.

[95] Hailong Yang, Quan Chen, Moeiz Riaz, Zhongzhi Luan, Lingjia Tang, and Jason Mars. 2017. PowerChief: Intelligent power allocation for multi-stage applications to improve responsiveness on power constrained CMP. In *Computer Architecture News*.

[96] P. Yedlapalli, N. C. Nachiappan, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das. 2014. Short-Circuiting Memory Traffic in Handheld Platforms. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 166–177.

[97] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *ATC*.

[98] Haibo Zhang, Prasanna Venkatesh Rengasamy, Shulin Zhao, Nachiappan Chidambaram Nachiappan, Anand Sivasubramaniam, Mahmut T. Kandemir, Ravi Iyer, and Chita R. Das. 2017. Race-to-sleep + Content Caching + Display Caching: A Recipe for Energy-efficient Video Streaming on Handhelds. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts). ACM, New York, NY, USA, 15.

[99] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: a dynamic library operating system for simplified and efficient cloud virtualization. In *2018 USENIX Annual Technical Conference*. 173–186.

[100] S. Zhao, H. Zhang, S. Bhuyan, C. S. Mishra, Z. Ying, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das. 2020. Déjà View: Spatio-Temporal Compute Reuse for' Energy-Efficient 360° VR Video Streaming. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 241–253.

[101] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 149–161.