Harmonizing Efficiency and Practicability: Optimizing Resource Utilization in Serverless Computing with JIAGU

Qingyuan Liu, Yanning Yang, Dong Du, and Yubin Xia, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education;* Ping Zhang and Jia Feng, *Huawei Cloud;* James R. Larus, *EPFL;* Haibo Chen, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education; Key Laboratory of System Software (Chinese Academy of Science)*

https://www.usenix.org/conference/atc24/presentation/liu-qingyuan

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

Harmonizing Efficiency and Practicability: Optimizing Resource Utilization in Serverless Computing with JIAGU

Qingyuan Liu[1,2], Yanning Yang[1,2], Dong Du[1,2], Yubin Xia[1,2], Ping Zhang[4], Jia Feng[4], James R. Larus[5], and Haibo Chen[1,2,3]

[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
[2]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education
[3]Key Laboratory of System Software (Chinese Academy of Science)
[4]Huawei Cloud
[5]EPFL

Abstract

Current serverless platforms struggle to optimize resource utilization due to their dynamic and fine-grained nature. Conventional techniques like overcommitment and autoscaling fall short, often sacrificing utilization for practicability or incurring performance trade-offs. Overcommitment requires predicting performance to prevent QoS violation, introducing trade-off between prediction accuracy and overheads. Autoscaling requires scaling instances in response to load fluctuations quickly to reduce resource wastage, but more frequent scaling also leads to more cold start overheads. This paper introduces JIAGU to harmonize efficiency with practicability through two novel techniques. First, *pre-decision scheduling* achieves accurate prediction while eliminating overheads by decoupling prediction and scheduling. Second, *dual-staged scaling* achieves frequent adjustment of instances with minimum overhead. We have implemented a prototype and evaluated it using real-world applications and traces from the public cloud platform. Our evaluation shows a 54.8% improvement in deployment density over commercial clouds (with Kubernetes) while maintaining QoS, and 81.0%–93.7% lower scheduling costs and a 57.4%–69.3% reduction in cold start latency compared to existing QoS-aware schedulers.

1 Introduction

Serverless computing [34] simplifies the management of cloud applications with features like on-demand execution and autoscaling. Through autoscaling, cloud providers can dynamically deploy numerous instances to run a specific function in response to real-time demands, optimizing performance and cost efficiency. Critical to this process are two main components: the autoscaler and the scheduler. The autoscaler utilizes a predefined threshold for each function, called *saturated value* (e.g., the average requests per second, or RPS, per instance), and triggers the creation of new instances as needed when the live load approaches this threshold [4, 7][1]. By doing so, the autoscaler prevents instances from becoming overloaded and guarantees performance, which is generally measured in terms of Quality-of-Service (QoS). When the autoscaler triggers the creation of a new instance, the scheduler is responsible for scheduling the instance to a suitable node. In making these assignments, the scheduler accounts for various aspects, including whether a node can meet a function's resource requirements which developers manually define.

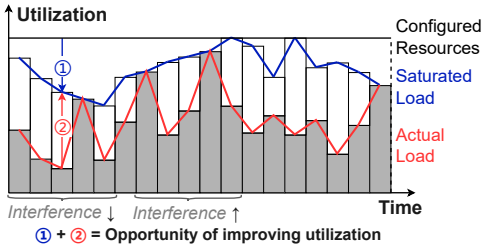


Figure 1: **Opportunities of improving resource utilization.** "Configured resources" are the fully utilized allocated resources. The blue line demonstrates the resource utilization of instances that are serving a saturated value of load, which fluctuates due to the variation in resource interference.

However, current serverless systems still fall short of efficiently utilizing resources. According to traces from our public cloud (Huawei Cloud) and AliCloud [68], most serverless functions can only utilize a small portion of allocated CPU and memory resources. Figure 1 is a schematic illustration of resource wastage within an instance. It shows that the issue of underutilized resources mainly results from two factors. First, to guarantee performance, users usually consider the worst case, and thus specify excessive resources. Even if instances are processing a saturated load of requests, they cannot fully utilize the allocated resources. This causes the resource wastage of part ① in Figure 1. Second, in practice, user loads served by each instance continuously fluctuate. Under-

[1]This approach typically ensures individual instance loads do not exceed the saturated value, as doing so would lead to the instantiation of additional instances, thereby redistributing the load.

loaded instances usually require fewer resources and are less able to utilize the resources, resulting in resource wastage of part ② in Figure 1.

To mitigate these two issues, serverless systems can leverage *overcommitment* [41, 68] and *autoscaling* [56]. First, for part ① in Figure 1, overcommitment is an intuitive solution, i.e., deploying more instances on a server. This approach could be implemented by the scheduler, as the scheduler is responsible for deciding how instances are deployed. Second, for part ②, it could be helpful to reduce the load fluctuations within instances, trying to keep the load close to the saturated load for each instance. This can be achieved by the autoscaler, which is responsible for observing load fluctuations and adjusting the number of instances accordingly.

Nevertheless, it is challenging to apply the two approaches while achieving effectiveness with practical cost. First, since overcommitment may lead to performance degradation and QoS violation, it is a common practice to predict performance before scheduling an instance to avoid QoS violation. However, it is challenging to *accurately predict instances' performance at a practical cost.* Although prior predictor-based schedulers have made great progress [22, 48–50, 68, 73, 81], they fall short in achieving both high accuracy and low runtime cost, since the two goals can be a trade-off: making accurate prediction requires considering complex *interference* on multiple resources imposed by highly heterogeneous colocated functions, but the complex computation inevitably introduces high overheads. For example, some prior works predict with specially designed machine learning models [81], which is costly for scheduling. Since cold start latency mainly consists of scheduling latency and instance initialization latency, with initialization having been optimized to $<$1ms, scheduling with inference ($\sim$20ms or higher) introduces a new bottleneck [43]. To compromise, public clouds still use heuristic policies for scheduling, thus failing to accurately predict performance and providing limited improvement in resource utilization.

Second, to efficiently utilize resources under load fluctuation, the autoscaler needs to quickly respond to the load variation, to evict instances and free the resources when the load drops. Intuitively, the faster autoscaling can react to load fluctuations, the more efficiently resources can be used. However, *more sensitive and frequent scaling also means more additional cold starts, resulting in the trade-off between resource utilization and cold start overheads.* It is challenging to achieve both goals simultaneously.

We present two key insights to tackle the aforementioned challenges. First, the trade-off between scheduling costs and efficiency is due to the fact that these complex decisions need to be made at the time of scheduling, using costly predictions. Therefore, it can be beneficial if we *decouple prediction and decision making*. Specifically, we can predict in advance the performance of possible incoming instances and save the results. If the incoming instance matches an earlier predicted scenario, the scheduling can directly check the prepared decisions without model inference. This provides a scheduling fast path.

Second, the trade-off between resource utilization and cold starts occurs because instances do not release resources until they are evicted. Instead, we can *decouple resource releasing and instance eviction.* Specifically, sending requests to fewer instances could achieve a similar effect to releasing resources by eviction. Releasing resources without instance eviction can be less costly, and thus can be applied with higher sensitivity to achieve better resource utilization.

Inspired by the two insights, we propose JIAGU, a serverless system that harmonizes efficiency and practicability to improve resource utilization, tackling the challenges with two techniques. First, JIAGU achieves accurate and low-cost performance prediction with *pre-decision scheduling*, which decouples prediction and decision making, providing a scheduling *fast path*. For every deployed function, JIAGU predicts in advance its *capacities* on a server using a model. A function's capacity means the maximum number of its instances that can be deployed on a server under the interference of current neighbors without violating everyone's QoS. To schedule a new instance of a deployed function, JIAGU only needs to compare the function's capacity with the number of its instances to determine whether the scheduling can be successful or not. Moreover, JIAGU designs an *asynchronous update* approach, which keeps the capacities up-to-date without incurring model inference on the scheduling critical path and applies a *concurrency-aware scheduling* to batch the scheduling of concurrent incoming instances upon load spikes. Results with real-world patterns show that >80% of scheduling goes through the fast path.

Second, JIAGU designs a *dual-staged scaling* method. When the load drops, before instances are evicted, JIAGU first adjusts the routing with higher sensitivity, sending requests to fewer instances, thus releasing some of the instances' resources. If the load rises again before idle instances are evicted, JIAGU can get the instances working again by rerouting with minimum overhead. Third, to cope with the case where a node is full and idle instances on it cannot be converted to the saturated state, JIAGU migrates idle instances to other nodes in advance to hide the overhead of the required cold starts.

We present a prototype of JIAGU based on an open-source serverless system, OpenFaaS [6]. We evaluate it using ServerlessBench [75], FunctionBench [37], and applications with real-world traces from Huawei Cloud. The results show that on real-world traces, compared to a state-of-the-art model-based serverless scheduler [81], JIAGU incurs 81.0%–93.7% lower scheduling costs and 57.4%–69.3% lower cold start latency with *cfork* [25]. It also achieves the highest resource utilization compared to all baseline schedulers (54.8% higher instance density than Kubernetes).

2 Motivation

2.1 Background and Motivation

Notations and terms. The term function denotes the basic logical unit in user programming within a serverless platform. A function might comprise multiple homogeneous instances to serve requests. For example, consider an image resizing function. Each time an image is uploaded, a new instance of the function is created to resize that specific file. Each instance operates independently, processing its assigned image file using the allocated resources. For example, the user may specify that each image resizing instance requires 2 CPU cores and 4 GB of memory to ensure efficient processing. These resources can be predefined by the user or automatically determined by the platform. Quality-of-Service (QoS) describes performance targets, primarily characterized by tail latency but extendable to other metrics.

Instance and resource schedulers. When a new instance is created, an instance scheduler is responsible for assigning a specific node to it for deployment. The scheduler takes several factors into account when selecting a node. For example, a scheduling policy would ensure that if the user specifies that each instance requires 2 CPU cores and 4 GB of memory, the scheduler will select nodes that can provide these resources. Moreover, some platforms also apply a resource scheduler to manage resource allocation and ensure efficient utilization, which we will discuss more in §8. The term "scheduler" as used hereinafter refers mainly to the "instance scheduler".

Components for load management in serverless platforms. First, a router dispatches user requests to available instances of a function, typically employing a *load balancing* strategy to ensure equal distribution of requests among instances. Second, an autoscaler [4, 7] is responsible for determining the number of instances. It uses a pre-determined RPS threshold, referred to as the saturated load, to determine whether it is necessary to create or evict instances, to avoid instance overloaded or resource waste. Different from creating instances, the autoscaler evicts instances with a keep-alive approach, which does not immediately evict instances when the expected number of instances drops, but waits for a "keep-alive duration". This duration can reflect the sensitivity of the autoscaler — shorter duration means higher sensitivity and more frequent evictions.

Low resource utilization in serverless platforms. Optimizing resource usage is one of the top cloud initiatives [2], as every 1% increase in utilization could mean millions of dollars in cost savings. However, production traces show that the resources of serverless platforms are still severely underutilized (Figure 3). In general, the wastage mainly arises from two parts: part ① and ② in Figure 1.

Part ① is because the resources demanded by a user are usually conservative, considering the worst cases that suffer the most severe resource interference. Therefore, the allocated resources could be even higher than the actual demand of instances that are processing saturated loads. Part ② is because the load served by each instance constantly fluctuates, and under-loaded instances are less able to fully utilize the resources. For example, Figure 2 shows the average RPS served by an instance for one of the most popular functions in the trace of Huawei Cloud. If the instance is always considered saturated, **51%** of resources could possibly be wasted.

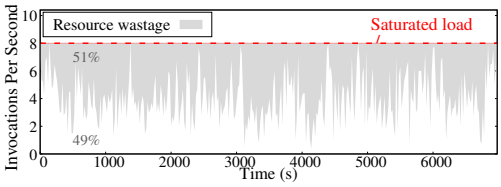


Figure 2: **Fluctuation of user load per instance.**

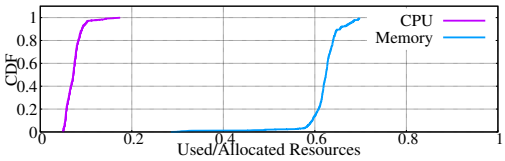


Figure 3: **Statistics on the ratio of actual resource usage to allocated resources.** The data was collected on Huawei Cloud over one month from a region. About 80% of servers only use 8% CPU and 64% memory resources.

Opportunities for improving resource utilization. To mitigate the issues, serverless systems could adopt two approaches respectively. First is overcommitment for part ①, i.e., the scheduler could deploy more instances on a server. Second is autoscaling for part ②, i.e., the autoscaler dynamically evicts instances when the load drops, reducing the overestimation of resource demands for under-loaded instances. With these approaches, our objective is to simultaneously achieve *effectiveness* in improving resource utilization at a *practical cost*.

2.2 Challenges

However, achieving both efficiency and practicability can be challenging, since the two goals usually have trade-offs.

2.2.1 Challenges for the Scheduler

Although overcommitment helps improve resource utilization, it could also cause performance degradation, possibly resulting in QoS violation. Therefore, a common approach for the scheduler is to predict instances' performances when scheduling, maximizing instance deployment density while not violating instances' QoS. The goal means that the scheduler should *simultaneously* achieve: accurate QoS violation prediction (with scalable profiling overhead) and low scheduling latency. Although prior schedulers have made great progress for the serverless scenario [68, 81] compared with schedulers [22, 48–50, 73] designed for monolithic services, they fall short of

Table 1: **A comparison between JIAGU and previously developed methods.** "Accurate prediction" cannot be achieved by simple heuristic algorithms or historical information. Scalable profiling requires no more than $O(n)$ complexity, where n represents the number of functions, k denotes to the number of instances that can be deployed on a server. Pythia, due to having a model for each function, incurs $O(n^2)$ complexity. Whare-map requires $O(n^k)$ complexity to profile all colocation combinations where k denotes to the number of co-runners allowed on a server. Owl profiles the combination of two functions with k instances, requiring $O(n^2k)$ complexity. *"Fast scheduling"* implies that the scheduler makes a decision within a few milliseconds (∼1ms). Gsight requires more than 20ms for scheduling (§7.2).

Workload	System	Year	Challenge I: efficient QoS-aware scheduling with low cost				Challenge II: handle load dynamicity with low cost
			Prediction model	Accurate prediction	Overhead		
					Profiling cost	Fast scheduling	
Serverless	**Jiagu**		**Global statistical model**	✓	$O(n)$	✓	**Dual-staged scaling**
	Owl [68]	2022	Historical information	Limited	$O(n^2k)$	✓	Autoscaling
	Gsight [81]	2021	Global statistical model	✓	$O(n)$	✗	Autoscaling
Monolithic Service	Pythia [73]	2018	Per-function Linear model	✓	$O(n^2)$	✓	Autoscaling
	Paragon [22]	2013	Heuristic	N/A	$O(n)$	✓	Autoscaling
	Whare-map [48]	2013	Historical information	Limited	$O(n^k)$	✓	Autoscaling
	Bubble-up [49]	2011	Heuristic	Inaccurate	$O(n)$	✓	Autoscaling
Resource Schedulers	Aquatope [82], Cilantro [15], Sinan [79] FIRM [55], Orion [45], Heracles [44], ...		Dynamically adjust allocated resources to instances. Resilient to runtime variations with limitations (§8).				

achieving both goals. Table 1 summarizes the key differences of prior schedulers against our objectives.

Accurate QoS prediction. The key to accurately predicting the QoS violation is to quantify the resource interference by colocated function instances, which is challenging. First, in the serverless scenario, the number of colocation combinations grows exponentially with the number of functions, and ultimately becomes practically infinite. Therefore, the profiling overhead should be scalable, otherwise it could be impossible to profile such a large number of colocation combinations. Second, resource interference can be complex, since plentiful and heterogeneous functions exert various pressures on multiple resources. The large number of colocation combinations further amplifies the interference complexity.

Some prior schedulers do not meet the goal. For example, some schedulers predict based on historical performance information [48, 68], so that their predictive ability is limited and can only predict the performance of profiled colocated instance combinations. Their profiling cost is also unscalable (Table 1). Schedulers that train a model for each function also have unscalable profiling and training overhead [73]. Schedulers using heuristic algorithms are limited in their ability to accurately predict performance due to overly simplistic models [49], or do not actually predict performance at all [22].

Low scheduling latency. The scheduling process is involved in every instance's cold start. Given the recent advancements in optimizing the instance initialization cost to a few milliseconds or even sub-milliseconds [13, 17, 25, 26, 33, 53, 63, 64, 70, 71], it is imperative to ensure that the scheduling cost remains low as well. Otherwise, the scheduling cost can become the new bottleneck in startup cost [43]. To illustrate, we collect the data from several cold start optimization papers and compare them with scheduling overheads in Table 2. We use the average scheduling costs of Gsight [81] (our ported version) to illustrate the overhead of scheduling over the total cold startup costs in these systems. Although it can accurately predict the performance and consider serverless-specific features like partial interference, it can incur non-trivial overhead even in commercial systems like AWS Lambda with Snapstart [16] (>20%). Scheduling cost can be dominant for cold starts with state-of-the-art optimizations like Catalyzer [26]. In short, schedulers need to make decisions within a few milliseconds to avoid hindering the startup process.

Table 2: **State-of-the-art serverless systems with startup optimizations.** "Container startup" presents the runtime startup latency reported by public reports or papers. "Scheduling overhead" presents the overhead of scheduling in a cold startup with model-based methods. We use 21.78ms as the model-based scheduling costs, the average result of our ported Gsight [81] model.

System	Container startup	Scheduling overhead
AWS Snapstart [16]	∼100ms	∼21.8%
Replayable [71]	54ms	40.3%
Fireworks [64]	∼50ms	∼43.6%
SOCK [53]	20ms	108.9%
Molecule [25]	8.4ms	2.6x
SEUSS [17]	7.5ms	2.9x
Catalyzer [26]	0.97ms	22.5x
Faasm [63]	0.5ms	43.6x

2.2.2 Challenges for Autoscaling

The autoscaling feature commonly helps mitigate the defect. When the load drops, the autoscaler would dynamically evict instances, thus freeing space for deploying new instances, and preventing those under-loaded instances from wasting resources (typically every instance can be under-loaded because of load balancing). Intuitively, a more sensitive autoscaler could better utilize resources in response to load fluctuation. However, more frequent scaling would also result in more additional cold starts. It means the *trade-off between resource*

utilization and cold start costs.

The irregularity of load fluctuations further complicates the problem. Prior works have analyzed production traces and tried to predict the invocation patterns [28, 58, 62, 82], pre-warming instances accordingly to reduce cold starts. However, user loads exhibit only moderate regularity over extended periods of time (e.g., diurnal patterns), but are extremely unpredictable over shorter intervals. For example, the average coefficient of variable (CV) over the number of requests in a minute could be more than 10 [76] in the Azure trace [62]. The more frequent scaling also means a finer-grained pre-warming prediction and a worse prediction accuracy.

2.3 Insights

We observe two insights that help to tackle the two challenges.

Insight-1: Prediction and decision making can be decoupled. The trade-off between prediction accuracy and scheduling cost arises from the coupling of the prediction and decision making, i.e., the costly model inference is usually made during scheduling. We can *decouple prediction and decision making*. Specifically, the scheduler can model the interference environment before a new instance arrives, and make advance predictions based on the assumptions. When a new instance arrives and the interference environment at the time of scheduling matches our previous assumption, the scheduling can be made by directly checking the prepared decisions. This provides a scheduling fast path without inference.

Insight-2: Resource releasing and instance eviction can be decoupled. The trade-off between resource utilization and cold start overheads arises from the coupling of resource releasing and instance eviction. Instead, we can *decouple resource releasing and instance eviction.* Specifically, even if an instance is not evicted, we could adjust the routing and not send requests to it. It consolidates the loads of under-loaded instances to fewer instances to reduce the waste of resources caused by treating under-loaded instances as saturated instances, achieving a similar resource releasing effect as an actual eviction. The overhead of adjusting the routing is much smaller than actual cold start overheads. Therefore, in this way, we could release/reclaim resources with higher sensitivity to cope with load fluctuations, while avoiding excessive additional cold start overheads.

3 Design Overview

We propose JIAGU, an efficient and practical QoS-aware serverless system that tackles the two challenges of improving resource utilization with these insights. The overall design is shown in Figure 4. First, JIAGU designs a pre-decision scheduling that could make accurate predictions with low scheduling latency (§4). When an instance is created, JIAGU predicts its performance after deployment, trying to increase

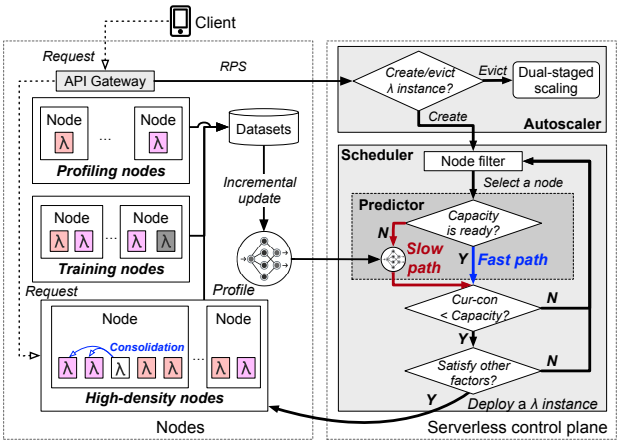


Figure 4: **JIAGU overview.**

the instance deployment density without violating QoS to fully utilize the resources. Second, JIAGU adopts a dual-staged scaling design that efficiently utilizes resources under load fluctuation with minimum overhead (§5).

Cluster setting. To apply JIAGU, our cluster is partitioned into three types of nodes. A small proportion of nodes are profiling nodes and training nodes, deploying instances whose runtime behavior is collected to construct function profiles and the dataset to train the model used by JIAGU's scheduler. JIAGU schedules instances that actually handle user requests to high-density nodes, maximizing the resource utilization of these nodes. All types of nodes are homogeneous.

User configurations and QoS. In our platform, when uploading a function, a user is expected to specify the allocated resources (CPU and memory) and the saturated load of the function (optional with default value). QoS provides basic performance guarantees for functions, which is beneficial for most users. Most jobs on our FaaS platform are latency-sensitive jobs. Hosting latency-insensitive jobs in FaaS is not considered an industry best practice as they have less need for ultra-high instance elasticity. On our platform, the QoS requirement of a function is set by the provider according to the function's previously monitored performance without interference. The QoS is set to be slightly more lenient than the collected performance, e.g., the P95 tail latency should be less than 120% of the solo-run P95 tail latency. Moreover, we can establish specific QoS agreements with top-tier customers, since their workload could be the majority in the cloud.

4 Pre-decision Scheduling

4.1 JIAGU's Prediciton Model

Prediction model. JIAGU's QoS-aware scheduling requires accurate prediction of the functions' performance under resource interference. The prediction model is based on Random Forest Regression (RFR). Specifically, JIAGU's regres-

Table 3: **Profiling metrics.**

Metrics	Description
mCPU	CPU utilization
Instructions	Instructions retired
IPC	Instructions per cycle
Context switches	Switching between privilege modes
MLP	Efficiency of concurrent access indicated by Memory Level Parallelism (MLP)
L1d/L1i/L2/LLC MPKI	Cache locality indicated by misses per thousand instructions (MPKI)
TLB data/inst. MPKI	TLB locality indicated by MPKI
Branch MPKI	Branch predictor locality indicated by MPKI
Memory bandwidth	Memory usage and performance

sion model has the following form:

$$P_{A\cup\{B,C,...\}} = RFR\{P_A, R_A, C_A, R_B, C_B, R_C, C_C, ...\}$$

where, $P_{A\cup\{B,C,...\}}$ is function A's performance[2] under the interference of functions B and C. R_A, R_B and R_C is the profile matrix of function A, B and C respectively. JIAGU uses a single instance's multiple resource's utilization as its profile (Table 3), and adopts a *solo-run* approach to collect the profile metrics of functions (details in §6). C_A, C_B and C_C are concurrency information of the three functions. The concurrency information includes two parts: the number of saturated instances and the number of cached instances (detailed in §5). P_A is the solo-run performance of function A.

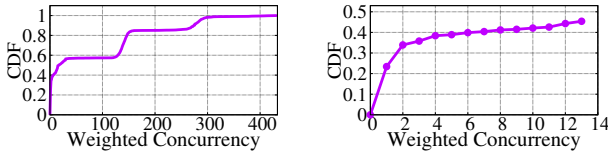
The model is proven effective in our public cloud and also utilized in prior systems [41, 81]. Moreover, compared to prior models that predict the performance at instance granularity [73, 81], since different instances of the same function are homogeneous and perform similarly on the same server, our model predicts performance at function-granularity. We merge the features of the same function's instances and introduce *concurrency* as a new feature for a function. This effectively reduces the input dimensions, resulting in less training overhead and possible mitigation of the "curse of dimensionality" [14]. JIAGU is also flexible to utilize other prediction models.

Inference overhead. In practice, a policy may require predicting different functions' performances under various colocation environments, and the colocation environments can be described by multiple inputs. Notably, learning frameworks (e.g., sklearn[9]) have optimizations to infer multiple simultaneous inputs, which incurs trivial additional cost (§7). Therefore, in the descriptions that follow, if multiple predictions can be merged in the form of multiple inputs, we refer to such inference overhead as "once" inference overhead.

4.2 Capacity and Capacity Table

Inspired by the first insight, to accurately predict instances' performances without incurring excessive costs to scheduling,

[2] Although we use P90 tail latency as our performance metric, it is a general model that can be extended to other metrics like costs.



(a) Weighted instance concurrencies of functions. **(b) Weighted instance concurrencies (<13) of functions.**

Figure 5: **Analysis of highly-replicated characteristic.**

we can decouple prediction and decision making. However, making predictions requires knowledge of what the incoming instance is and what the interference environment on the server is, which is not known until scheduling (decision making). Considering there are an infinite number of functions, it is also impossible to traverse all possible interference environments. Therefore, decoupling can be challenging.

Serverless features that help tackle the challenge. We realize that the *highly-replicated* feature of serverless may help address the issue. The feature can be illustrated with the following statistical analysis on real-world traces collected from Huawei Cloud's production environment. Specifically, we mainly analyze the concurrency of function instances, i.e., the number of concurrently running instances of a function in a certain time window, rather than the invocation concurrencies in previous studies [58, 62, 78, 82]. We weight the ratio of every concurrency value by itself, e.g., if there are 100 functions that have only 1 concurrency but 1 function whose concurrency is 100, the CDF point should be (1, 0.5) and (100, 1). The distribution is shown in Figure 5-a. Each point (e.g., (x,y)) in the figure indicates that (y$\times$100)% instances belong to functions whose concurrencies are $\leq$ x. For example, 56% of instances are from functions whose concurrency is $>$12, which is shown more clearly in Figure 5-b. Instances of functions with only one concurrent instance are just 23% of all instances. In conclusion, such analysis shows serverless's *highly-replicated* feature — **most serverless functions have many replicated instances**.

Introducing capacity for decoupling. The feature means, that when deploying an instance on a server, it is likely that more instances of the same function would come in the future. Moreover, considering locality when scheduling has also been proven beneficial by plentiful prior works [11, 61]. Therefore, to decouple prediction and decision making, we predict in advance whether the next incoming instance of *existing functions* can be deployed. With this idea, for every server, JIAGU calculates *capacity* for each existing function on that server. JIAGU predicts in advance the function's new instance's performance after deployment, and sets the capacity value "true" if the predicted performance meets its QoS. The capacities for all functions on a server form the server's *capacity table*. After that, when the scheduler is making scheduling decisions, it can predict QoS violation by simply checking the table without model inference. This is the scheduling "fast path". Moreover, it is only necessary to make predictions on the critical path if the function to which the incoming instance

belongs is not in the capacity table. This is the scheduling "slow path".

4.3 Asynchronous Update

Although introducing capacity could predict the QoS violation of new instances in advance, the deployment of a new instance could impose resource interference to its neighbors, possibly causing them QoS violations. Therefore, it is necessary to introduce a *validation* process, which predicts and checks whether neighbors will violate their QoS after deploying the new instance. However, such validation introduces costly inference for cold starts if done before deployment, or the risk of QoS violation if done after deployment.

To mitigate the defect, JIAGU applies an *asynchronous update* approach. To avoid the cost, asynchronous update refines the way to calculate the capacity, predicting all colocated functions' performances, and setting the capacity "true" if the predicted performance of each function meets its own QoS. In this way, after deploying the new instance, the predicted performances of either the new instance itself or the neighbors would not violate the QoS. This prevents the validation from affecting the scheduling latency.

Furthermore, when a new instance is scheduled to a server, it triggers updating the capacity table. Now that the validation can be delayed, and this update can be done asynchronously, outside the scheduling critical path. By asynchronous update, the capacity table is always up-to-date with minimum overhead, ensuring that the capacity table always reflects real-time interference on servers when scheduling.

The "asynchronous update" is the way to make the necessary updates to the capacity table when a new instance arrives with low overhead. Besides, JIAGU also allows for additional updates to be conducted on a regular and timely basis, which does not affect the execution of functions. These updates can further help keep the capacity table up-to-date in case the workload pattern changes, etc. More details in §6.

4.4 Concurrency-aware Scheduling

Decoupling prediction efficiently reduces model inference for individual schedulings. Moreover, one of the extreme invocation patterns in the cloud is a load spike, where the load rises rapidly so that multiple instances are created simultaneously. In such case, the asynchronous update of the previous instance could possibly block the scheduling of the next instance, as shown in Figure 7-a. To mitigate, JIAGU further adopts *concurrency-aware scheduling*, which batches the scheduling of concurrent incoming instances at load spikes.

JIAGU further refines the way of calculating capacity, considering not only whether the next instance can be deployed, but also how many next instances can be deployed, as shown in Figure 6. The capacity value of each function is no longer a bool value, but a specific numerical value indicating how

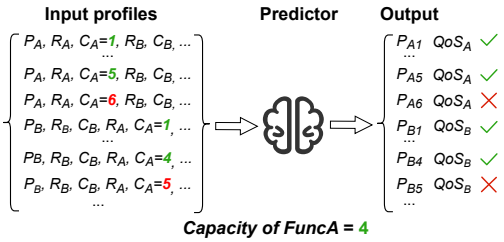
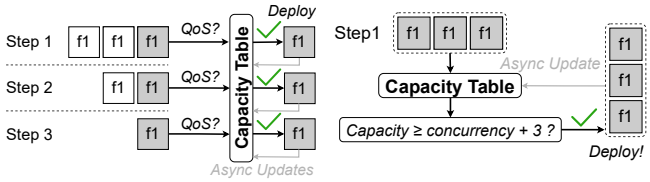


Figure 6: **Calculating capacity.** It finds the maximum concurrency value as the capacity which every colocated function's predicted performance can guarantee its QoS.



(a) Without batching. **(b) With batching.**

Figure 7: **Concurrency-aware scheduling enables batch scheduling for one function's multiple instances.**

many instances of the function can be deployed with current neighbors. For example, for a server where there are 2 f1 and 3 f2 instances, with f1 and f2's capacities are 4 and 6 respectively. It means that 4 f1 instances can be deployed with 3 f2 instances, while 6 f2 instances can be deployed with 2 f1 instances. Then, as shown in Figure 7-b, when multiple instances of the same function arrive, if the function's capacity on the server is sufficient to accommodate those new instances, the scheduling and the asynchronous update can be done once for scheduling multiple instances.

4.5 Put It All Together: Scheduling Example

Now we describe the complete scheduling process and provide an example. As shown in Figure 4, when a function's new instance is created, JIAGU will first select a node for it using a node filter (detailed in §6). Then, based on the node's capacity table built in advance, the scheduler decides whether to schedule the instance via a fast path or a slow path.

Figure 8 demonstrates an example. First, an instance of f3 arrives, and the node filter selects the node in Figure 8 for it. The scheduler checks the capacity table of the node and finds that there is no entry for f3. Therefore, the scheduler calculates f3's capacity using the prediction model (as shown in Figure 6). The result means that 3 f3's instances can be deployed with current numbers of f1 and f2 instances, so the new instance of f3 can be deployed on that server. It is the scheduling slow path involving a costly model inference.

Then, when a load spike occurs, two instances of f3 are created and come to the node. The scheduler checks the capacity table that the number of f3's instances will not exceed its capacity after deployment, so that the scheduling decision is made quickly without model inference. This is the schedul-

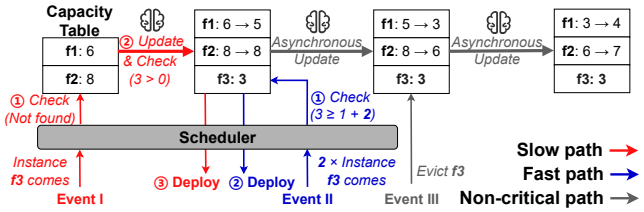
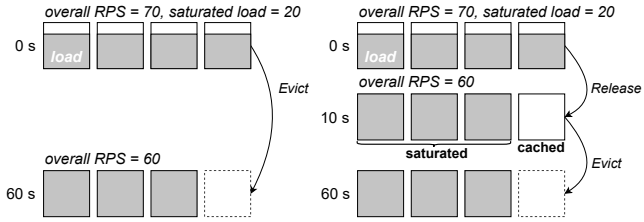


Figure 8: **Scheduling example on a node.**



(a) Existing eviction. **(b) Dual-staged eviction.**

Figure 9: **Dual-staged scaling.** In the example, the release and keep-alive duration are 10 and 60 seconds respectively.

ing fast path. The deployment also triggers an asynchronous update of the capacity table, which can be done outside the scheduling critical path. The schedulings of two instances are batched, triggering only one update of the capacity table.

When the load drops and an instance of f3 is evicted, such an event will also trigger an update to the capacity table. The increased capacities of f1 and f2 mean that the resources can be reutilized by the scheduler to deploy more instances later. Of course, this update would hardly affect the scheduling.

5 Dual-staged Scaling

Motivated by the second insight, JIAGU designs dual-staged scaling to efficiently utilize resources under load fluctuation while minimizing cold start overheads.

Dual-staged eviction. The key technique of the scaling approach is dual-staged eviction. When the load drops and so as the expected number of instances decreases, instead of directly evicting instances after the keep-alive duration as shown in Figure 9-a, JIAGU introduces a "release" duration (Figure 9-b). The "release" duration is (much) shorter than the keep-alive duration, i.e., the "release" operation is more sensitively triggered before eviction. After the expected number of instances drops for the release duration, the autoscaler first triggers the "release" operation, which changes the routing rules, sending requests to fewer instances, but does not actually evict any instances. The instances not actually processing requests are called cached instances, while other instances that are still processing requests are called saturated instances, as shown in Figure 9-b. In the example, the load of four instances is consolidated into three instances. Therefore, when considering resource interference, the scheduler would consider only three saturated instances instead of four.

Logical cold start. If the load rises again and the expected number of instances exceeds the current number of saturated instances, if there are cached instances, JIAGU triggers a "logical cold start". Specifically, JIAGU's scheduler chooses a cached instance, and then the router re-routes requests to the selected instance, letting it return to a saturated instance. This "logical cold start" is fast since the re-routing only takes $< 1ms$, while a real instance initialization is more costly.

Real eviction. If the load does not rise again after the initial drop, after the keep-alive duration, the cached instances are actually evicted. If the load rises again and there are no cached instances, it will trigger the autoscaler to initialize new instances, which involves "real cold starts". It is consistent with the traditional autoscaling approach.

Scheduling with dual-staged scaling. The scheduler is responsible for maintaining and updating the capacity tables. After applying dual-staged scaling, the scheduler would be triggered to update the capacity tables not only in response to actual creation (or eviction) of instances, but also to instance releases (or logical cold starts). Specifically, after the "release" operation, since some saturated instances are released to cached instances, the scheduler will trigger asynchronous updates as if they are evicted, which may increase the capacities of other functions on the server. It means that the resources can be reutilized by the scheduler to deploy more instances, thus reducing resource wastage caused by overestimation of loads. Similarly, after logical cold starts, the scheduler will also trigger asynchronous updates and may decrease other functions' capacities. Therefore, the release (or logical cold starts) could release (or reclaim) resources with greater sensitivity and efficiency than actual instance evictions (or creations), without the need to incur the overheads associated with actual cold starts.

On-demand migration. If the number of saturated instances of a function on a server drops and new instances of other functions are deployed to the server, the function's capacity could be updated and reduced. Thus, after a while, the server could possibly be "full", i.e., the conversion of the function's cached instances to saturated instances on that server could result in the number of saturated instances surpassing its capacity. For example, if f1's capacity is 5 (possibly updated several times as instances of other functions' arrive), and it has 4 saturated instances and 3 cached instances, this means that there is no room for 2 $(= 3 + 4 - 5)$ cached instances to convert back to saturated instances. At this moment, if the load rises again, JIAGU has to re-initialize instances on other feasible nodes, rather than merely trigger "logical cold starts".

To avoid incurring such additional "real cold starts", JIAGU learns in advance how many cached instances can no longer be converted to saturated instances by comparing the capacity with the sum of the numbers of both types of instances (2 in the above example). Then, it migrates these cached instances to other feasible servers in advance before they are required to

be converted. Therefore, such additional cold start overheads can be hidden and will not affect the running of instances.

6 Implementation

We implement JIAGU based on OpenFaaS [6], requiring 4.5k+ LoCs changes. JIAGU is also developed and studied at Huawei Cloud.

OpenFaaS architecture. OpenFaaS is a high-level controller based on Kubernetes (K8s), which schedules instances with the K8s scheduler and deploys instances with K8s resources (e.g., K8s *Deployment* and *Service* [3, 10]). It receives user requests from its gateway, and distributes requests to each instance's queue. OpenFaaS relies on Prometheus [8] to monitor the RPS of user requests and autoscale instances accordingly. The autoscaler calculates the expected number of instances of a function by dividing the overall RPS by the predefined saturated load. By default, it applies a 60-second keep-alive duration, i.e., it evicts instances after the expected number of instances is less than the current number and lasts 60 seconds. If the expected number of instances is larger than the current number, it creates an instance immediately to avoid performance degradation.

JIAGU's scheduling controller. JIAGU's scheduling logic is implemented as a centralized controller, which manages the model, capacity tables,and other information required by the scheduling. It works together with a plugin for the Kubernetes scheduler in the prototype, allowing for interaction between the scheduler and the controller and enabling the scheduler to obtain placement decisions for incoming instances. It also implements the asynchronous update, where the scheduling results are returned first before performing the update. Moreover, it is responsible for choosing instances for "logical scaling" for dual-staged scaling with its scheduling algorithm.

Implementing dual-staged scaling. To implement dual-staged scaling, JIAGU adds new Prometheus rules to monitor RPS, and calculates the expected number of saturated instances according to the monitored values. It applies a 45 (or 30, this sensitivity can be configured) seconds "release" duration, i.e., if the expected number of saturated instances decreases and lasts for 45 (or 30) seconds, it notifies the router to re-route. To re-route, the scheduling controller chooses an instance and labels it as "cached". The "cached" instances are excluded by the function's corresponding K8s *Service* [10] so that new requests will not be routed to them. When the load rises and the expected number of saturated instances increases, it immediately triggers a logical cold start, i.e., the scheduler chooses a cached instance and unlabels it.

Profiling and model training. JIAGU adopts a *solo-run* approach motivated by Gsight [81] to collect the profile metrics of functions. To profile a function, JIAGU additionally deploys an instance to the profiling node exclusively, executes the function by the instance under the saturated load, and profiles the resource utilizations using tools such as *perf* for a duration. Moreover, to construct and maintain a training dataset, JIAGU further collects the performance metrics of various colocation combinations at runtime.

User configurations for functions. How users properly configure their functions [45, 56] (e.g., configure resources and the saturated value) is out of the scope of the paper. Even if a user wisely configures their functions, the traditional approach of scheduling and autoscaling could still suffer from interference and load fluctuations (shown in Figure 1), leading to resource underutilization.

Node filter and management. The node filter described in Figure 4 prioritizes worker nodes in JIAGU. For a new instance, it grants higher priority to nodes that have deployed instances of the same function, as scheduling on those nodes possibly follows a fast path. While the current implementation focuses on this simple priority scheme, future work may explore extending it with additional scheduling policies, such as placing multiple instances of the same function on a single node to benefit from the locality [11, 61]. In cases where no usable nodes are available for an instance, JIAGU will request the addition of a new server to the cluster. Similarly, an empty server will be evicted to optimize costs.

System parameter configuration. The implementation involves configuring several specific parameters, including the profiling time, the definition of model convergence, or the release/keep-alive durations. Setting these parameters requires careful consideration of trade-offs based on the actual scenario. For example, for dual-staged scaling, a shorter release duration (e.g., a 30-second duration) could mean that the system would be more sensitive to load variations (e.g., than a 45-second duration), and therefore more likely to achieve higher resource utilization. However, it could also lead to more logical cold starts, which in turn could lead to more real cold starts or on-demand migrations. Therefore, these parameters should be configured carefully, taking into account the actual requirements of the system and these trade-offs.

Performance predictability in the cloud. QoS violations are mostly predictable, especially when using tail latency as the QoS metric, which considers the overall performance of multiple requests over a period of time. It is consistent with the observations in our production environment and prior works [41, 50, 73, 81]. In Huawei Cloud, JIAGU keeps monitoring the QoS violation and adopts two ways to handle the incidental unpredictability. First, the random forest model is naturally feasible for incremental learning, and we continuously collect runtime performance metrics and retrain the model periodically with the up-to-date training set in case the behavior of functions changes. Second, if the prediction error does not converge after several iterations, JIAGU disables overcommitment and uses traditional conservative QoS-unaware policies to schedule the instances of the unpredictable function on separate nodes. JIAGU can perform well and improve

resource utilization in the real world.

Overheads of (re-)training and inference. With JIAGU's random forest model, the (re-)training and inference only require one core. We also evaluate the temporal cost, and the results are illustrated in Figure 17. The overhead is non-trivial compared to the saved resources.

7 Evaluation

7.1 Methodology

Experimental setup. We use a cluster of 24 machines for evaluation. Each machine is equipped with an Intel Xeon E5-2650 CPU (2.20GHz 48 logical cores in total) and 128GB memory, running Ubuntu LTS 18.04 with Linux 4.15 kernel. One machine is dedicated to OpenFaaS control plane components (gateway, scheduler, JIAGU's controller, etc.). The machine is also responsible for sending requests and collecting QoS and utilization results. The other machines are worker machines responsible for executing or profiling function instances.

Baseline systems. We compare JIAGU with three baseline systems: Kubernetes, Gsight [81] and Owl [68]. Kubernetes scheduler is one of the mostly used serverless scheduler systems in production [1, 5]. Gsight [81] and Owl [68] represent state-of-the-art predictor-based and historical information-based serverless scheduling systems. Since they do not open source the implementation, we implement comparable prototypes of them. Moreover, we evaluate three versions of JIAGU. Jiagu-30 and Jiagu-45 denote prototypes with 30 and 45 seconds "release duration" of dual-staged scaling respectively, while JIAGU-NoDS disables dual-staged scaling. We mainly compare JIAGU with instance schedulers rather than resource schedulers like Orion [45]. We believe that these two types of schedulers are not incompatible and can benefit from each other. Details on the comparisons are in Section 8.

Scheduling effect metrics. To evaluate the scheduling effect, we compare JIAGU with our baseline scheduling algorithms by two metrics. First is the *QoS violation rate*, which is the percentage of requests that violate QoS in all requests to all functions. The QoS constraint is chosen to be 120% of the tail latency when the instance is saturated and suffers no interference, consistent with previous work [20, 68, 81, 82] and in-production practice. In the evaluation, we aim to achieve a QoS violation rate of less than 10% so we predict the p90 tail latency accordingly. Second is the *function density*, higher density means better resource utilization. We normalize the traditional Kubernetes scheduler's function density to one, meaning that instances are deployed with exactly the amount of configured resources. One of the goals of JIAGU is to increase function density ($>$1) as much as possible while achieving an acceptable QoS violation rate ($<$10%).

Workloads and traces. We use six representative functions in ServerlessBench [75] and FunctionBench [37] for evaluation, including model inference (rnn), batch processing applications (image resize, linpack), log processing, chameleon and file processing (gzip). All functions are configured with the same amount of resources. To ensure the robustness of our system to perturbations in the benchmark inputs, we optionally add zero-mean Gaussian noise to the inputs.

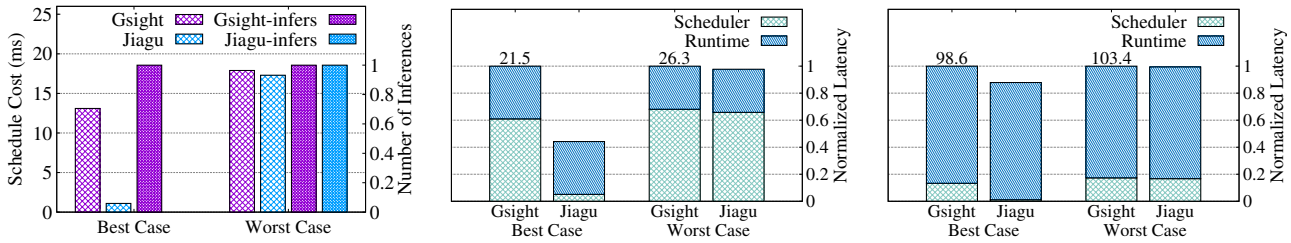
We use various traces for evaluation. We mainly use real-world traces from Huawei Cloud since they possess a similar level of complexity as in the production environment. The traces and the existing open source traces [35] are collected at different times on the same system. We take the following approaches to ensure that our evaluated traces are representative and generalizable. First, to generate a set of traces, we *randomly* select six functions' invocation patterns from real-world Huawei Cloud traces and map each pattern to a function with a similar execution time. The randomness here is to ensure the generality of the traces. Second, we generate four different sets of real-world traces, each with six invocation patterns, to prevent the evaluation results from being influenced by randomness in the traces. Third, the four traces are selected from various geographical regions. Fourth, for each set of real-world traces, we exclude inactive functions with very low request rates. Finally, to adapt sampled traces to our cluster size, we use methods similar to prior works [69, 82], i.e., scale patterns such as request frequency proportionally so that our cluster is not overloaded with peak loads.

In addition, we also use specially constructed traces to evaluate the scheduling performance in extreme scenarios.

7.2 Scheduling Performance Analysis

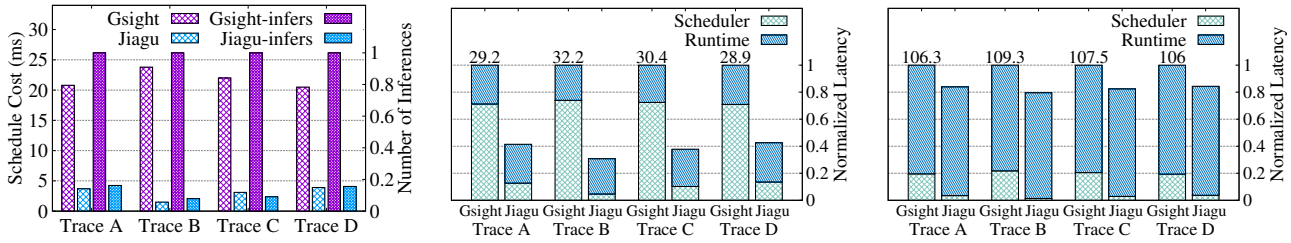
In this section, we evaluate the scheduling costs with different traces in two aspects. First, we evaluate the average scheduling costs of the two algorithms. Second, we simulate how JIAGU can optimize the cold start in conjunction with one of the state-of-the-art instance initialization optimizations, container fork (cfork) [25], which can create an instance in $\sim$ 8.4ms. We also analyze the cold start with Docker ($\sim$85.5ms). In each test, we compare JIAGU's default version (Jiagu-45) with Gsight, whose performance is normalized to 1.

Performance under extreme scenarios. We first analyze the extreme performance of JIAGU using specific traces. For the best case, we use a timer trace with only one function. It assumes that the function is invoked and instances are scaled at a fixed frequency. The results (Figure 10-a) show that Gsight suffers a scheduling overhead 11.9x larger than JIAGU. This is because almost all scheduling decisions of JIAGU go through the fast path in this case. Considering cold start latency (Figure 10-b), Gsight's is 126.3% longer than JIAGU when using cfork because of its costly scheduling. For the worst case, we construct a rare trace where the function concurrencies recurrently change from 0 and 1, so that every scheduling process goes through the slow path. The results show that JIAGU's performance degraded to a similar level as Gsight.



(a) Scheduling cost and model inferences. **(b) Cold start optimization with cfork.** **(c) Cold start optimization with docker.**

Figure 10: **Performance analysis under extreme scenarios.** It includes scheduling cost, number of model inferences per schedule and cold start latency optimizations. We normalize results and label concrete numbers (ms) in the figure.



(a) Scheduling cost and model inferences. **(b) Cold start optimization with cfork.** **(c) Cold start optimization with docker.**

Figure 11: **Performance analysis on real-world traces.** It includes scheduling cost, number of model inferences per schedule and cold start latency optimizations. We normalize results and label concrete numbers (ms) in the figure.

When using Docker in both cases, the cost of instance initialization becomes the bottleneck. We believe that as a growing number of efforts have limited initialization overhead to <10ms [13, 17, 25, 26, 33, 53, 63, 64, 70, 71], the reduction in scheduling costs will be increasingly meaningful.

Performance with real-world traces. We then analyze JIAGU's scheduling costs with four real-world traces. The results (Figure 11) show that for real-world traces, JIAGU achieves 81.0%–93.7% lower scheduling costs than Gsight. This is because JIAGU's scheduling policy can drastically reduce the number of model inferences (83.8%–92.1%, shown in the y2 axis of Figure 11-a), so that the inference overhead is amortized over multiple cold starts. For cold start latency, when applying JIAGU with cfork, the reduced scheduling costs lead to 57.4%–69.3% lower cold start latency than Gsight.

Breakdown of cold start overhead reduction. This paragraph shows how JIAGU's design effectively reduces cold start overhead with a breakdown analysis. As shown in Figure 12, we evaluate the total cold start latencies with cfork/docker, and with different scaling/releasing sensitivities (30 or 45-second keep-alive duration). For each trace, the highest total cold-start overhead is normalized to one, i.e., scheduling without a fast path and scaling instances with a 30-second keep-alive duration. The result shows that dual-staged scaling significantly reduces the total cold-start overhead compared to the baseline. This is achieved by reducing the number of cold starts. In addition, pre-decision scheduling could further reduce the cold-start overhead by reducing the scheduling latency.

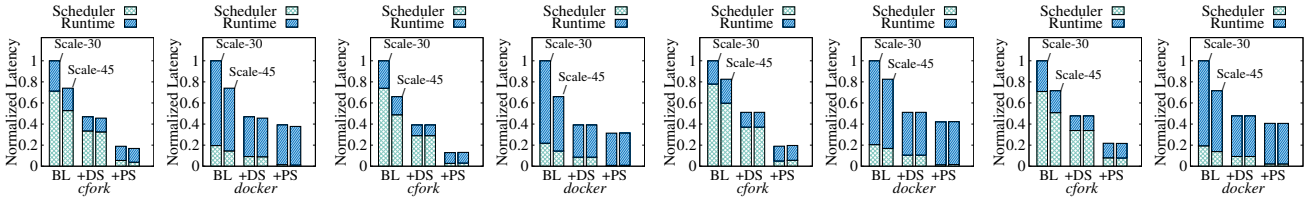
In conclusion, JIAGU's scheduling algorithm can significantly reduce model inference times and result in lower scheduling costs, leading to better cold start latency.

7.3 Scheduling Effect on OpenFaaS

This section describes how JIAGU improves resource utilization while guaranteeing QoS. We analyze JIAGU and the baseline systems by function density and QoS violation rate, with the four real-world traces. The density result is the average value during the evaluation weighted by duration.

According to Figure 13, all QoS-aware schedulers improve function density compared with Kubernetes (which is normalized to 1). Without dual-staged scaling, Gsight and JIAGU (Jiagu-NoDS) achieve higher function density than Owl. It shows Owl's limitation of allowing only two colocated functions prevents optimal scheduling decisions. With dual-staged scaling, JIAGU further optimizes resource utilization. Higher sensitivity performs better in improving the instance density. Specifically, Jiagu-30 achieves up to 54.8% higher function density than Kubernetes, 22.0% higher than Gsight and 38.3% higher than Owl, while its QoS violation rate is comparable with Gsight and Jiagu-NoDS. This is because when user load drops, JIAGU can quickly utilize the resources of the unsaturated instances. Moreover, on four traces, all schedulers achieve an acceptable QoS violation rate of less than 10%, and the four results are similar. Therefore, we only show the result on Trace A (Figure 14-a).

Reducing cold starts by migration of cached instances. This section describes how many additional cold starts can be avoided by migrating cached instances. According to Figure 14-b, for 45 seconds sensitivity, all re-routing operations are logical cold starts, needless to migrate instances. With



(a) Trace A. **(b) Trace B.** **(c) Trace C.** **(d) Trace D.**

Figure 12: **Total cold start overheads during evaluations with various runtimes and optimizations.** "BL" is the baseline with no fast path and scales instances directly with a 30/45 second keep-alive duration. "+DS" applies dual-staged scaling (60-second keep-alive duration and 30/45-second release duration) which reduces the number of cold starts. "+PS" further applies pre-decision scheduling, which reduces the overhead of each cold start.

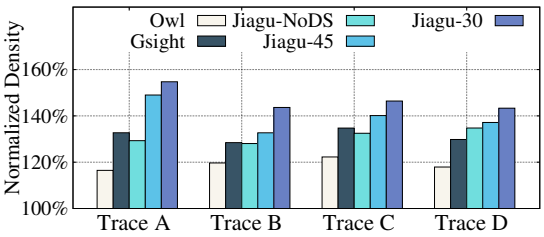
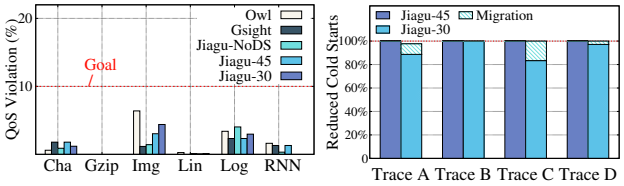


Figure 13: **Normalized function density.** Density on K8s is normalized to 100%. JIAGU achieves the highest density with all of the real-world traces.



(a) QoS violation of Trace A. **(b) Reduced cold starts.**

Figure 14: **QoS violations and reduced cold starts.** Applications are chameleon (Cha), gzip, image resize (Img), linpack (Lin), log processing (Log) and RNN.

30-second sensitivity, only a small proportion of re-routing, i.e., $< 20\%$ for four traces and 0 for Trace B, require "real" rather than "logical" cold starts, most of which can be avoided by migrating cached instances in advance. Only Trace A has very few additional real cold starts at 30-second sensitivity. Similar results can be found in Figure 12, i.e., after applying dual-staged scaling, the cold start overheads are almost the same for both 30/45 second sensitivities. It illustrates that with dual-staged scaling, JIAGU effectively improves resource utilization at minimum cost.

In conclusion, with the prediction-based scheduler and dual-staged scaling, JIAGU can achieve high resource utilization while limiting QoS violations with less scheduling overhead than state-of-the-art model-based scheduling policy.

7.4 Prediction Analysis

Prediction accuracy. We evaluate the prediction accuracy of our model, as shown in Figure 15. We define the prediction error rate of a model as $\frac{|\hat{P}-P|}{P}$, where $\hat{P}$ and P are predicted

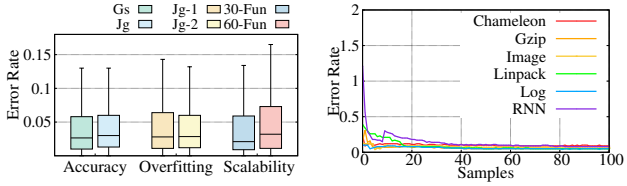


Figure 15: **Model accuracy**. (a) is the error rate of the prediction models. (b) shows that the prediction error drops with new samples.

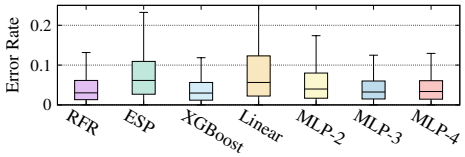


Figure 16: **Prediction errors of various models.**

performances and real performances, respectively. First, we analyze the accuracy of the model used by previous evaluations. The model could accurately predict the performance of the six functions, with similar prediction error to Gsight (Jg and Gs in the Figure). Second, we evaluate the overfitting of the model by splitting the test set into two equal-sized sets (Jg-1 and Jg-2). The model achieves similar prediction errors on the two different sets, showing that it does not overfit to a specific test set. Third, we evaluate the model's scalability, i.e., how the model can predict the performance of an increasing number (30/60) of functions. The result shows that additional functions do not affect the prediction precision. Fourth, we evaluate how the model is resilient to the changes in the function's behavior. Each time, we train the model with five functions, and add the remaining function to the system, collect performance metrics as new samples and retrain the model once a sample is added. Figure 15-b shows that the performance prediction error of the new function drops rapidly under increasing number of samples, and converges after about 5–30 samples. It means that it takes limited time to retrain an accurate model after collecting a couple of samples.

Finally, we evaluate the model choices, comparing JIAGU's random forest regression (RFR) with ESP [50], XGBoost, linear regression, and three multi-layer perception (MLP) models with 2,3,4 layers respectively. The result (Figure 16) confirms our choice of the RFR model considering its high accuracy,

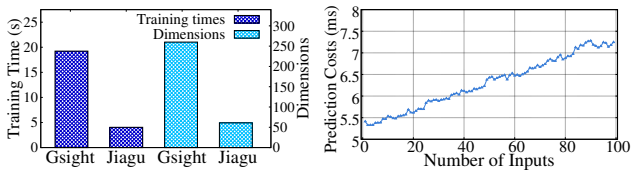


Figure 17: **Model performance.** (a) is the training time and the total number of dimensions of the two models. (b) JIAGU's model inference overhead with the number of inputs increases.

low training overhead and capability of incremental learning, etc. Possible other models or algorithms (e.g., ridge regression) can be explored to improve JIAGU in future work.

Model performance. We evaluate JIAGU's model training and inference costs. Figure 17-a shows that JIAGU's model has better training performance and a lower number of input dimensions. Moreover, Figure 17-b demonstrates that the model inference costs increase only 2 milliseconds even when the number of inputs increases to 100. Therefore, although JIAGU batches multiple inputs to the prediction model when calculating the capacity, it will not significantly increase the inference overhead.

In conclusion, JIAGU's model can predict the performance accurately with a much lower number of input dimensions and training overhead.

8 Related Work

Resource schedulers. Another widely studied (and orthogonal) scheduler is the online resource scheduler [12, 15, 18–21, 23, 24, 27, 29, 30, 39, 40, 44, 46, 47, 51, 52, 54, 55, 57, 59, 60, 72, 74, 77, 79, 80, 82], which tunes the allocated resources according to functions' online performance to guarantee QoS and improve resource efficiency. Many of them are designed for long-running services [19, 20, 27, 29, 30, 44, 47, 54, 55, 57, 59, 72, 74, 77, 79, 80], which are challenging to apply directly to transient serverless instances. In addition, some serverless resource schedulers [12, 60, 82] use algorithms like Bayesian Optimization to search the resource configuration space and make optimal decisions based on runtime performance. They are resilient to runtime performance variations but still fail to guarantee QoS before convergence.

In practice, the resource scheduler can function orthogonally to the instance scheduler, effectively complementing the instance scheduler's shortcomings in dynamics and rapid decision-making. The coordination of the two types of schedulers enhances efficiency through broader resource allocation insights, leading to better load balancing and utilization. This synergy maybe crucial in serverless computing, aligning with principles of minimal user intervention and optimized automation, ensuring instance schedulers are effective and adaptable in dynamic environments.

Request schedulers. Rather than scheduling instances, many prior works tackle the problem of assigning instances for requests. Atoll [65] overcomes many latency challenges in the serverless platform via a ground-up redesign of system control and data planes, utilizing deadline-aware scheduling. Hermod [36] analyzes and tunes different design choices on OpenWhisk and improves function performance. Fifer [31] queues requests to warm containers to reduce the number of instances and prewarms instances with LSTM resource interference to optimize the cold starts. Sequoia [67] is deployed as a proxy to control user requests and enable QoS. Neither of them considers resource interference. Golgi [41] applies a model for the router, predicting requests' performance and sending them to overcommitted instances if the performance does not violate QoS, or otherwise to non-overcommitted instances. It can avoid QoS violation for a fixed number of instances, but does not guide for how many overcommitted instances need to be deployed, while an excessive number of overcommitted instances could cause resource under-utilization.

Other optimizations. There are many related serverless optimizations [32, 38, 42, 45, 57, 58, 66, 67]. For example, Orion [45] optimizes the colocation of parallel invocations, enhancing the performance of serverless DAG systems. WiseFuse [46] introduces a joint optimization approach for both serial functions and parallel invocations, further improving serverless DAG efficiency. Their designs contain resource schedulers to right-size VMs, while also adapting other key optimizations for other aspects, such as bundling for parallel executions. FaasFlow [42] proposes a worker-side workflow schedule pattern that enables better scalability and performance. All of these systems can benefit from JIAGU with an effective scheduler to also improve the resource utilization for cloud vendors without introducing high scheduling costs.

9 Conclusion

This paper presents our experience in optimizing resource utilization in Huawei Cloud for serverless computing with JIAGU. JIAGU decouples prediction and decision making to achieve efficient and fast scheduling, and decouples resource releasing and instance eviction to achieve a fast reaction to load fluctuation without incurring excessive cold start overhead. Our results show JIAGU brings significant benefits for resource efficiency using real-world traces.

10 Acknowledgement

We sincerely thank our shepherd Somali Chaterji and anonymous reviewers for their insightful suggestions. This work is supported in part by National Key Research and Development Program of China (No. 2022YFB4502003), China National Natural Science Foundation (No. 62302300, 61925206), the HighTech Support Program from STCSM (No. 22511106200), and Startup Fund for Young Faculty at SJTU (SFYF at SJTU). Dong Du is the corresponding author.

References

[1] Apache openwhisk is a serverless, open source cloud platform. http://openwhisk.apache.org/, 2023. Referenced 2023.

[2] Cloud computing trends and statistics: Flexera 2023 state of the cloud report. https://www.flexera.com/blog/cloud/cloud-computing-trends-flexera-2023-state-of-the-cloud-report/, 2023.

[3] Deployments | kubernetes. https://kubernetes.io/docs/concepts/services-networking/service/, 2023.

[4] Knative autoscaling. https://knative.dev/docs/serving/autoscaling/, 2023.

[5] Kubernetes scheduler. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/, 2023.

[6] Openfaas - serverless functions made simple. https://www.openfaas.com, 2023.

[7] Openfaas autoscaling. https://docs.openfaas.com/architecture/autoscaling/, 2023.

[8] Prometheus - monitoring system and time series database. https://prometheus.io/, 2023.

[9] scikit-learn: machine learning in python. https://scikit-learn.org/, 2023.

[10] Service | kubernetes. https://kubernetes.io/docs/concepts/services-networking/service/, 2023.

[11] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 365–380, New York, NY, USA, 2023. Association for Computing Machinery.

[12] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. Cose: Configuring serverless functions using statistical learning. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 129–138, 2020.

[13] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. *SAND*: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[14] R.E. Bellman. *Dynamic Programming*. Dover Books on Computer Science Series. Dover Publications, 2003.

[15] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Cilantro: Performance-Aware resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 623–643, Boston, MA, July 2023. USENIX Association.

[16] Marc Brooker. Lambda snapstart, and snapshots as a tool for system builders. https://brooker.co.za/blog/2022/11/29/snapstart.html, 2022. Referenced Dec 2022.

[17] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[18] Jiahao Chen, Zeyu Mi, Yubin Xia, Haibing Guan, and Haibo Chen. CPC: Flexible, secure, and efficient CVM maintenance with Confidential Procedure Calls. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, Santa Clara, CA, July 2024. USENIX Association.

[19] Quan Chen, Zhenning Wang, Jingwen Leng, Chao Li, Wenli Zheng, and Minyi Guo. Avalon: Towards qos awareness and improved utilization through multi-resource management in datacenters. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 272–283, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 107–120, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Shuang Chen, Angela Jin, Christina Delimitrou, and José F. Martínez. Retail: Opting for learning simplicity to enable qos-aware power management in the cloud. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 155–168, 2022.

[22] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. Association for Computing Machinery.

[23] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. Association for Computing Machinery.

[24] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 97–110, New York, NY, USA, 2015. Association for Computing Machinery.

[25] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, pages 797–813, New York, NY, USA, 2022. Association for Computing Machinery.

[26] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. *Caladan: Mitigating Interference at Microsecond Timescales*. USENIX Association, USA, 2020.

[28] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.

[29] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Practical and scalable ml-driven performance debugging

in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 135–151, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 19–33, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C. Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. Fifer: Tackling resource underutilization in the serverless era. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, pages 280–295, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 691–707, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 152–166, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[35] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. How does it function? characterizing long-term trends in production serverless workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 443–458, New York, NY, USA, 2023. Association for Computing Machinery.

[36] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Hermod: Principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, pages 289–305, New York, NY, USA, 2022. Association for Computing Machinery.

[37] Jeongchul Kim and Kyungyong Lee. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 477–477, 2019.

[38] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, 2018. USENIX Association.

[39] Neeraj Kulkarni, Gonzalo Gonzalez-Pumariega, Amulya Khurana, Christine Shoemaker, Christina Delimitrou, and David Albonesi. CuttleSys: Data-Driven Resource Management for Interactive Applications on Reconfigurable Multicores. In *53rd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, October 2020.

[40] Qian Li, Bin Li, Pietro Mercati, Ramesh Illikkal, Charlie Tai, Michael Kishinevsky, and Christos Kozyrakis. Rambo: Resource allocation for microservices using bayesian optimization. *IEEE Computer Architecture Letters*, 20(1):46–49, 2021.

[41] Suyi Li, Wei Wang, Jun Yang, Guangzhen Chen, and Daohe Lu. Golgi: Performance-aware, resource-efficient function scheduling for serverless computing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 32–47, New York, NY, USA, 2023. Association for Computing Machinery.

[42] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: Enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 782–796, New York, NY, USA, 2022. Association for Computing Machinery.

[43] Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, and Haibo Chen. The gap between serverless research and real-world systems. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 475–485, New York, NY, USA, 2023. Association for Computing Machinery.

[44] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, New York, NY, USA, 2015. Association for Computing Machinery.

[45] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[46] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), jun 2022.

[47] Amiya K. Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. Mitigating interference in cloud services by middleware reconfiguration. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 277–288, New York, NY, USA, 2014. Association for Computing Machinery.

[48] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 619–630, New York, NY, USA, 2013. Association for Computing Machinery.

[49] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 248–259, New York, NY, USA, 2011. Association for Computing Machinery.

[50] Nikita Mishra, John D. Lafferty, and Henry Hoffmann. Esp: A machine learning approach to predicting application interference. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pages 125–134, July 2017.

[51] Joe Novak, Sneha Kumar Kasera, and Ryan Stutsman. Auto-scaling cloud-based memory-intensive applications. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 229–237, 2020.

[52] Joe H. Novak, Sneha Kumar Kasera, and Ryan Stutsman. Cloud functions for fast and robust resource auto-scaling. In *2019 11th International Conference on Communication Systems & Networks (COMSNETS)*, pages 133–140, 2019.

[53] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. *SOCK*: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, 2018.

[54] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware colocation of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206, 2020.

[55] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. *FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices*. USENIX Association, USA, 2020.

[56] Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 387–402, Boston, MA, July 2023. USENIX Association.

[57] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pages 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[58] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 753–767, New York, NY, USA, 2022. Association for Computing Machinery.

[59] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[60] Aakanksha Saha and Sonika Jindal. Emars: Efficient management and allocation of resources in serverless. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 827–830, 2018.

[61] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. Lukewarm serverless functions: Characterization and optimization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 757–770, New York, NY, USA, 2022. Association for Computing Machinery.

[62] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[63] Simon Shillaker and Peter Pietzuch. Faasm: lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020.

[64] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. Fireworks: A fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 663–677, New York, NY, USA, 2022. Association for Computing Machinery.

[65] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pages 138–152, New York, NY, USA, 2021. Association for Computing Machinery.

[66] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, sep 2020.

[67] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 311–327, New York, NY, USA, 2020. Association for Computing Machinery.

[68] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. Owl: Performance-aware scheduling for resource-efficient function-as-a-service cloud. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, pages 78–93, New York, NY, USA, 2022. Association for Computing Machinery.

[69] Dmitrii Ustiugov, Dohyun Park, Lazar Cvetković, Mihajlo Djokic, Hongyu Hè, Boris Grot, and Ana Klimovic. Enabling in-vitro serverless systems research. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless*, WORDS '23, page 1–7, New York, NY, USA, 2023. Association for Computing Machinery.

[70] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 443–457. USENIX Association, 2021.

[71] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[72] Ziliang Wang, Shiyi Zhu, Jianguo Li, Wei Jiang, K. K. Ramakrishnan, Yangfei Zheng, Meng Yan, Xiaohong Zhang, and Alex X. Liu. Deepscaling: Microservices autoscaling for stable cpu utilization in large scale cloud systems. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, pages 16–30, New York, NY, USA, 2022. Association for Computing Machinery.

[73] Ran Xu, Subrata Mitra, Jason Rahman, Peter Bai, Bowen Zhou, Greg Bronevetsky, and Saurabh Bagchi. Pythia: Improving datacenter utilization via precise contention prediction for multiple co-located workloads. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, pages 146–160, New York, NY, USA, 2018. Association for Computing Machinery.

[74] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. *SIGARCH Comput. Archit. News*, 41(3):607–618, jun 2013.

[75] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '20. Association for Computing Machinery, 2020.

[76] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association.

[77] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 379–391, New York, NY, USA, 2013. Association for Computing Machinery.

[78] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 724–739, New York, NY, USA, 2021. Association for Computing Machinery.

[79] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 167–181, New York, NY, USA, 2021. Association for Computing Machinery.

[80] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 406–418, USA, 2014. IEEE Computer Society.

[81] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. Understanding, predicting and scheduling serverless workloads under partial interference. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[82] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, pages 1–14, New York, NY, USA, 2022. Association for Computing Machinery.