Is Function-as-a-Service a Good Fit for Latency-Critical Services?

Haoran Qiu
University of Illinois, Urbana-Champaign
Urbana, Illinois, USA
haoranq4@illinois.edu

Saurabh Jha
University of Illinois, Urbana-Champaign
Urbana, Illinois, USA
sjha8@illinois.edu

Subho S. Banerjee
University of Illinois, Urbana-Champaign
Urbana, Illinois, USA
ssbaner2@illinois.edu

Archit Patke
University of Illinois, Urbana-Champaign
Urbana, Illinois, USA
apatke@illinois.edu

Chen Wang
IBM Research
Yorktown Heights, New York, USA
chen.wang1@ibm.com

Franke Hubertus
IBM Research
Yorktown Heights, New York, USA
frankeh@us.ibm.com

Zbigniew T. Kalbarczyk
University of Illinois, Urbana-Champaign
Urbana, Illinois, USA
kalbarcz@illinois.edu

Ravishankar K. Iyer
University of Illinois, Urbana-Champaign
Urbana, Illinois, USA
rkiyer@illinois.edu

Abstract

Function-as-a-Service (FaaS) is becoming an increasingly popular cloud-deployment paradigm for serverless computing that frees application developers from managing the infrastructure. At the same time, it allows cloud providers to assert control in *workload consolidation*, i.e., co-locating multiple containers on the same server, thereby achieving higher server utilization, often at the cost of higher end-to-end function request latency. Interestingly, a key aspect of serverless latency management has not been well studied: the trade-off between application developers' latency goals and the FaaS providers' utilization goals.

This paper presents a multi-faceted, measurement-driven study of latency variation in serverless platforms that elucidates this trade-off space. We obtained production measurements by executing FaaS benchmarks on IBM Cloud and a private cloud to study the impact of workload consolidation, queuing delay, and cold starts on the end-to-end function request latency. We draw several conclusions from the characterization results. For example, increasing a container's allocated memory limit from 128 MB to 256 MB reduces the tail latency by 2× but has 1.75× higher power consumption and 59% lower CPU utilization.

WoSC '21, December 6, 2021, Virtual Event, Canada
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9172-6/21/12...$15.00
https://doi.org/10.1145/3493651.3493666

CCS Concepts

• **Software and its engineering** → **Cloud computing**; **Scheduling**; • **General and reference** → **Measurement**.

Keywords

Serverless Computing, Function-as-a-Service, Resource Management, Performance Modeling, Multi-tenancy

ACM Reference Format:
Haoran Qiu, Saurabh Jha, Subho S. Banerjee, Archit Patke, Chen Wang, Franke Hubertus, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2021. Is Function-as-a-Service a Good Fit for Latency-Critical Services? . In *Seventh International Workshop on Serverless Computing (WoSC7) 2021 (WoSC '21), December 6, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3493651.3493666

1 Introduction

Serverless Function-as-a-Service (FaaS) is an emerging cloud computing paradigm that frees customers (i.e., application developers) from infrastructure management tasks such as resource provisioning and scaling. Customers are charged only based on function execution time, hence it is becoming a popular cloud computing paradigm for the deployment of bursty services (e.g., social media [6] and machine learning model serving [11]) as cloud functions [3, 6]. However, FaaS suffers from high variability in end-to-end latency (i.e., function request completion time) [20]. Such variation hinders the adoption of FaaS for latency-critical, user-facing cloud services with strict performance-based service-level objectives (SLOs) [3, 8, 12, 20]. In traditional cloud computing paradigms, customers could configure cloud resources (e.g., number of cores and memory limits) to overprovision to meet their end-to-end SLOs. However, in FaaS platforms, cloud providers completely manage resource provisioning

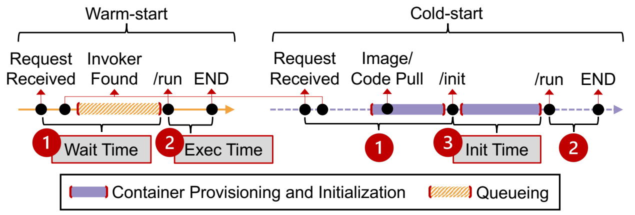


Figure 1: Breakdown of end-to-end latency for cold starts and warm starts in OpenWhisk [2] (defined in Section 2).

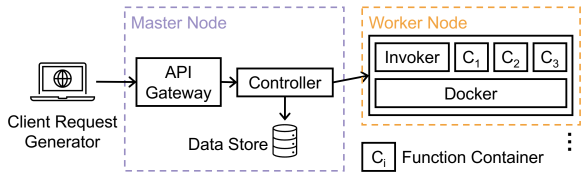


Figure 2: Distributed OpenWhisk [2] architecture.

to achieve *workload consolidation*, i.e., co-locating multiple containers on the same server. Such consolidation achieves higher server utilization, often at the cost of higher end-to-end latency [14, 16].

In order to address this end-to-end latency variation problem, the trade-offs between customers' latency goals and cloud providers' utilization goals must be studied. Understanding the trade-off space will allow cloud providers and customers to more optimally set configuration parameters to provide end-to-end latency guarantees and other SLOs (e.g., FaaS availability). This paper presents a multi-faceted, measurement-driven study of latency variation in serverless platforms that elucidates the trade-off space and the conflicting goals between the two parties. We obtained production measurements by executing widely used FaaS benchmarks [18, 23] on an open-source serverless platform, OpenWhisk [2], running on both a public (IBM Cloud) and our private cloud. We show an in-depth latency variation analysis done by breaking down the end-to-end latency into three phases: wait, execution, and initialization times (labeled as 1, 2, and 3, respectively, in Fig. 1).

Results. Our main results are as follows.

(i) *Latency-Utilization-Power Trade-off.* Increasing resource granularity (e.g., increasing a container's allocated memory limit from 128 MB to 256 MB) reduces the 99th percentile (P99) latency by up to 2× (from 3192 ms to 1063 ms) but consumes up to 1.75× more power (49 W higher), while reducing the average CPU utilization by up to 59% (from 99%). We describe these results further in Section 3.1.

(ii) *Latency Variability due to Shared Resource Contention.* Compared to isolated runs, P99 latencies increase by up to 32.6×, 28.9×, and 4.4× for CPU, memory, and last-level cache (LLC) sensitive workloads, respectively, because of shared resource contention. Moreover, state-of-the-art resource partitioning or distribution technologies like Linux CFS `cpu.shares` and Intel CAT/MBA [10] fail to mitigate tail latency variations. When using these technologies, P99 latencies increase (compared to isolated runs) by up to 8.3×, 21.5×, and 2.3× for CPU, memory, and LLC sensitive workloads, respectively. We discuss these results further in Section 3.2.

(iii) *Breakdown of End-to-end Latency.* Increasing the horizontal concurrency (i.e., number of containers) from 2 to 12 on a single server via decreasing resource granularity reduces P99 wait time by 49.5× from 1820 ms, increases P99 initialization time by 1.3× from 409 ms, and increases P99 execution time by 15.6× from 484 ms. Wait times dominate the end-to-end latency at lower concurrencies (2–4) while execution times dominate at higher concurrencies. We describe these results further in section 3.3.

Table 1: Serverless benchmarks adopted from [18, 23].

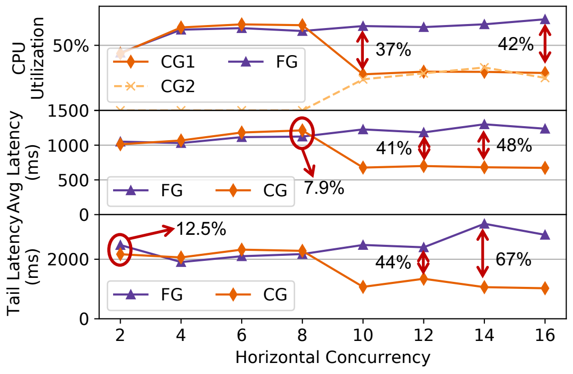
Benchmark	Description
Base64	Encode and decode a string with the Base64 algorithm.
Primes	Find the list of prime numbers less than 10^7.
Markdown2HTML	Render a Base64 uploaded text string as HTML.
Sentiment-Analysis	Generate a sentiment analysis score for the input text.
Image-Resize	Resize the Base64-coded image with new sizes.

2 Experimental Setup

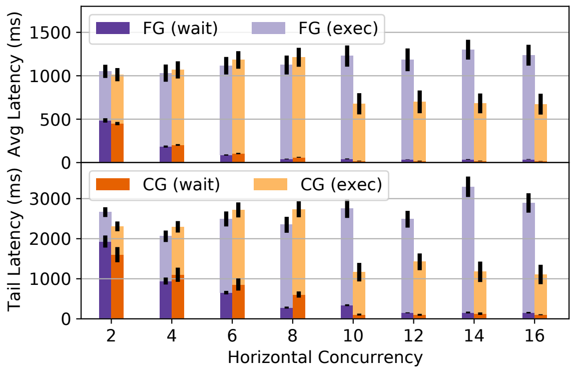
Serverless FaaS Platform. A serverless FaaS platform runs functions in response to invocations (i.e., requests). It consists of a central *controller* and a group of *invokers*. We chose OpenWhisk [2], a production-grade serverless platform based on Docker containers. Fig. 2 shows the architecture of a distributed OpenWhisk platform. The controller allocates CPU and RAM for each function container and places the container at an invoker. When requests come in via the *API Gateway*, the controller distributes the requests to invoker(s). An invoker executes a function after it gets the request and the results are written to a *Data Store*. A container is evicted after an idle timeout of 10 minutes (default in OpenWhisk).

Serverless Benchmarks. The benchmarks used in this study (listed in Table 1) are from widely used open-source FaaS benchmark suites [18, 23]. They include both microbenchmarks and macrobenchmarks, which have different runtime behaviors and resource demands (e.g., CPU utilization and memory bandwidth utilization). The functions are written in either Python or Java.

Cluster Setup and Experimental Methodology. We deployed OpenWhisk [2] on five VMs in IBM Cloud as well as our private cloud with five physical nodes. Both clusters have 1 master node and 4 worker nodes (shown in Fig. 2). Each VM in IBM Cloud has 4 vCPUs and 16 GB memory. Each node in our private cloud has a dual-socket Intel Xeon E5-2683 v3 processor with 14 cores per socket and 500 GB



(a) End-to-end latency and CPU utilization



(b) Wait time and execution time

Figure 3: Performance and CPU utilization comparison between FG and CG workload consolidation policies.

memory. We used the private cloud to conduct experiments for which we need privileged access to the machines, e.g., for power measurement and cache partitioning. All nodes run Ubuntu 18.04.3 LTS with Linux kernel version 4.15. Memory swapping is disabled for the Docker service. We ran the workload generator [18] from a separate node in the same cluster with Poisson-distributed inter-arrival of requests and used FaaSProfiler [18] to trace requests to measure the end-to-end latency. Recall that Fig. 1 shows the breakdown of end-to-end latency for serving a request in cold/warm starts:

Definition 1. *Wait time* is the time spent waiting in OpenWhisk before running (queueing time for warm starts).

Definition 2. *Init time* is the time spent initializing the function container (e.g., language runtime) in cold starts.

Definition 3. *Exec time* is the time spent by each function container in executing the request.

3 Measurements

To address the end-to-end latency variation problem in serverless platforms for serving latency-critical workloads, we performed a measurement-driven study on the breakdown of the end-to-end latency, as well as the trade-offs among latency, CPU utilization, and power consumption. In particular, we are interested in the resource granularity configuration in a workload consolidation policy. In our experiments, we tune the memory limits for each function container. We do this because CPU and other resources are allocated proportionally to memory limits in serverless platforms. Resource granularities are discrete points in a spectrum where a fine-grained (FG) workload consolidation policy allocates a smaller memory limit for containers while a coarse-grained (CG) policy allocates a larger limit. A CG policy is used in traditional Platform-as-a-Service (PaaS) or Infrastructure-as-a-Service (IaaS) paradigms where customers pay for the uptime of the allocated resources. In this study, we consider the memory limits above 256 MB as coarse-grained because PaaS platforms [15] like Heroku, Google Compute Engine, and AWS ECS/EKS allow a minimum memory capacity of 512 MB and the smallest step size is 250 MB. We consider the limits below 256 MB as fine-grained since the smallest configurable limit is 128 MB in OpenWhisk [2].

To help understand what role a workload consolidation policy plays in the end-to-end latency variation problem, our experiments focused on the following questions:

Question 1. What is the trade-off among power consumption, CPU utilization, and end-to-end latency in the decision-making of cloud providers for choosing a workload consolidation policy? (Section 3.1)

Question 2. How is the performance variation affected by fine-grained workload consolidation? (Section 3.2)

Question 3. How do different workload consolidation policies affect the breakdown percentages of different phases in the end-to-end latency? (Section 3.3)

3.1 Latency-Utilization-Power Trade-off

Resource overcommitment achieves cost efficiency by allocating an amount of virtualized CPUs or memory that exceeds the amount of available physical resources. Combined with multi-tenancy, overcommitment allows serverless platforms to multiplex limited resources across thousands of functions and reduce the total number of servers in use [1, 21]. Overcommitment is possible because of the use of FG workload consolidation policies and the bursty, high-idleness nature of FaaS workloads [3, 6, 19]. We show that the resource granularity in the CG and FG policies leads to trade-offs among function end-to-end latency, resource utilization, and cluster operation costs (measured in power consumption).

3.1.1 Latency Variation

In this study, each invoker was allocated 4 cores and 2 GB memory for hosting function containers. We have chosen to use Base64 to demonstrate the comparison of FG and CG policies; we observed similar results for the other benchmarks. We set the container memory size to 128 MB and 256 MB for the FG and CG policies, respectively. With a fixed request arrival rate of 4 RPS (50% of the max load that sustains SLO, given the high idleness [19]), we increased the horizontal concurrency (the number of containers) from 2 to 16 (the maximum concurrency is limited by cluster capacity).

Fig. 3(a) shows the end-to-end latency and CPU utilization comparison of the FG and CG policies when the concurrency varies from 2 to 16. Note that the CG policy scales out to two nodes (denoted as CG1 and CG2 in Fig. 3(a)) when the concurrency is greater than 8. For all concurrency values between 2 and 8, the difference of average and tail latencies between the two policies is smaller than 7.9% and 12.5%, respectively. When the concurrency is between 10 and 16, the FG policy has a 41–48% higher average latency and a 44–67% higher tail latency. The reason is that when the concurrency level is high, the execution time increases because of resource contention under the FG policy, although the wait time is decreased because of increased concurrency (see Fig. 3(b)). However, the CG policy has 37–42% less per-node average CPU utilization as it scales out to two nodes.

With the same experiment setup, to show the performance difference between the two policies, we also searched through a larger search space of granularities between the FG (128 to 192 MB) and CG (256 to 1024 MB) policies based on the configurations of OpenWhisk and existing PaaS platforms. Fig. 4 shows the difference between the tail latencies of the two policies; positive values indicate performance degradation, and negative values indicate performance gain of the FG policy compared to the CG policy. The first row of Fig. 4 is the latency difference between the two policies shown in Fig. 3(a). In the cases where the FG policy consolidated function containers using fewer nodes than the CG policy did, the performance degradation caused by the FG policy ranged from 29% to 89%. The larger the granularity gap, the larger the tail latency degradation, i.e., higher value in Fig. 4. However, when the CG policy reached the limit of 4 invoker nodes in the cluster, the performance difference started to reduce (e.g., for CG, a drop from 69.4% to 50.9% at 768 MB and a drop from 79.5% to 51.3% at 1024 MB with concurrency equal to 16) because of resource contention.

Implication. *Compared to FG policies, a CG policy scales out containers on more number of servers, resulting in less resource contention and thus up to 67% lower end-to-end latency.*

3.1.2 *Trade-off Assessment*

In cloud-managed service hosting, cloud providers value lower operation costs and better resource utilization efficiency [14]. Customers with latency-critical FaaS services prefer lower costs and lower end-to-end latency (i.e., fewer

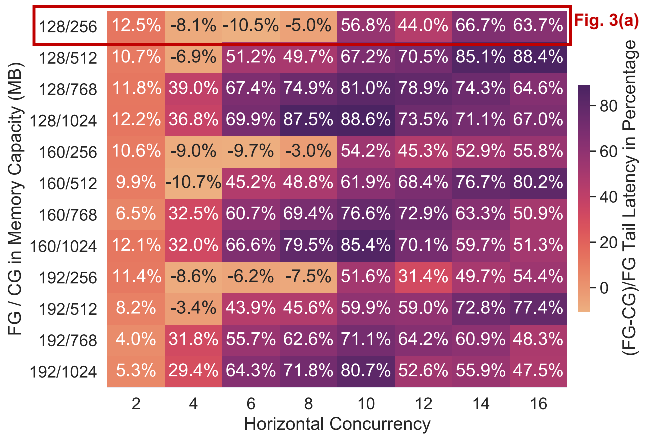
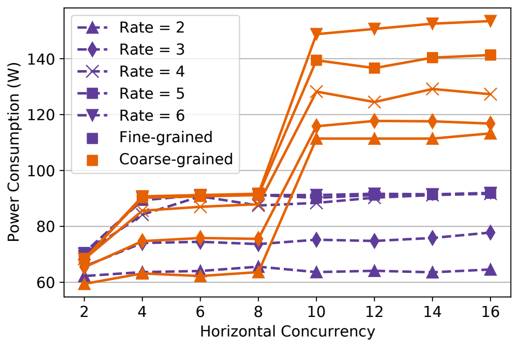


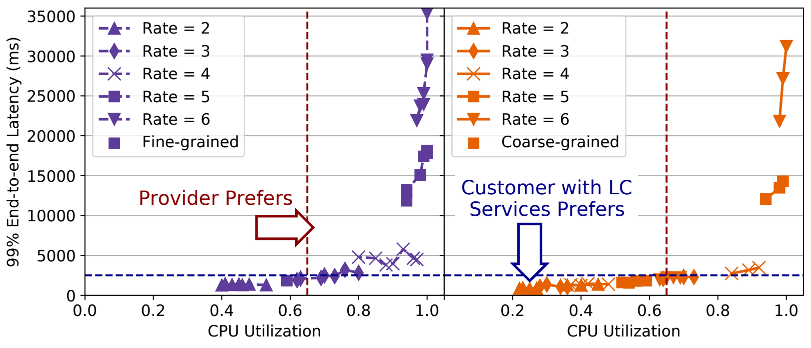
Figure 4: FG and CG policy configuration space search.

SLO violations) [6, 8]. As a result, the two conflicting goals raise an economic quandary and questions about the existing FaaS pricing models. Fig. 5 illustrates a performance and cost comparison between the two policies under different workload scenarios (for which we varied the arrival rate and concurrency). For each scenario, we measured the average total power consumption of all nodes (used to estimate operation costs), the 99th-percentile end-to-end latency, and the average CPU utilization across all invoker nodes. From the cloud provider's point of view, the FG policy helps keep the power consumption at a lower level when the concurrency is high (see Fig. 5(a)), and helps keep the server utilization at a higher level (see Fig. 5(b)). Therefore, cloud providers prefer to stay on the right side in Fig. 5(b), with low operation costs and high revenue. The CG policy consumes much more power because of the scaling-out actions at high concurrency levels, but it leads to less resource contention and lower tail latency. Therefore, customers would prefer a CG policy (typically in PaaS and IaaS where they can easily have control of the resources that they pay for) to achieve low end-to-end latencies (the bottom side in Fig. 5(b)).

However, the CG policy has up to 1.75× higher power consumption (when the arrival rate = 2/s and concurrency = 16) and up to 59% less average CPU utilization (when arrival rate = 4 and concurrency = 16). The FG policy, on the other hand, leads to data points at the right and top corner in Fig. 5(b), where latency-critical services suffer severe performance degradation but cloud providers achieve high server utilization efficiency. Current FaaS pricing models [8, 9] (with no function performance indicators) incentivize providers to go over server capacity by selling more function invocations per unit time, while also increasing the execution time, which conflicts with the customers' latency objectives. An interface between the two parties to allow customers to negotiate function performance SLOs and resource demands is missing. Inclusion of performance SLOs helps resolve the dilemma



(a) Power consumption vs. concurrency



(b) Utilization vs. latency

Figure 5: Comparison of FG and CG policies in terms of power consumption, CPU utilization, and the 99th percentile tail latency of function invocations. The blue dotted line is the function SLO latency set based on the "knee" of the tail-latency-to-throughput curve [4] and the red dotted line is the cluster CPU utilization goal [14].

but SLO negotiation could be challenging due to various sources of SLO violations [16].

Implication. *An FG policy leads to lower operation costs (up to 1.75× less) and better server utilization efficiency (up to 59% higher), while a CG policy offers the customers lower end-to-end latency (up to 2× less). The conflicting goals of the two parties raise questions on the pricing model (how to balance the needs of both parties?) and on the provider-customer interface (how should resource and performance needs be conveyed?) when using FaaS for latency-critical workloads.*

3.2 Performance Interference

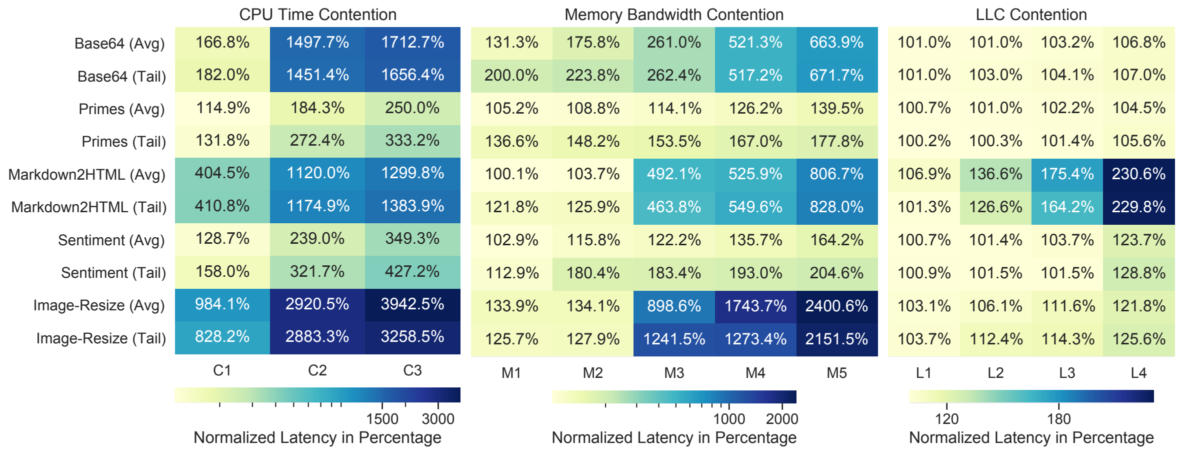
Resource overcommitment via FG workload consolidation policies helps cloud providers achieve lower operation costs and better resource utilization efficiency, which, in turn, results in a performance penalty to the function end-to-end latency: (i) Invocations suffer from cold starts when the containers are paused or evicted to save space (in terms of memory capacity) for active functions. (ii) Co-located function containers contend for shared resources such as CPU, memory bandwidth, last-level cache (LLC), and network bandwidth, leading to interference and performance unpredictability. Interference is particularly disruptive for interactive, latency-critical services with strict SLOs. Recent efforts (e.g., [1, 19]) have addressed the problem of reducing or avoiding cold-start latency. However, no prior work has addressed the performance interference problem due to resource contention in serverless platforms.

We first ran each serverless benchmark alone, without any background jobs running alongside. We then co-located each benchmark with contentious microbenchmarks running on each invoker with the same resource allocation (i.e., container size and concurrency). Doing so helped decouple sensitivity to resource allocation from sensitivity to resource contention. We studied compute-related resources (i.e., CPU and LLC) and storage-related resources (i.e., memory). Fig. 6 shows the diverse impacts of resource interference across the five serverless benchmarks. Typically, an application is more sensitive to the types of resources whose utilization tends to saturate. However, latency-critical workloads with tight SLOs are more sensitive to any resource contention. Thus, high resource usage does not always correlate with sensitivity to the same resource interference.

CPU Time. We ran iBench [5], a CPU-intensive benchmark, at different `cpu.shares` ratios with the serverless workloads (as OpenWhisk distributes CPU time by allocating CPU shares proportionally to the memory capacity). We found that all benchmarks are most sensitive to CPU time, such that the SLOs are violated when CPU time allocation is insufficient. For example, the tail latency for Image-Resize increases by 32.6× and 8.3× when the `cpu.shares` ratio is 1:4 and 1:1, respectively. Compared to the average latencies, tail latencies suffer more for all cases except for Image-Resize, as it is also dominated by I/O access and has less CPU demand.

Memory Bandwidth. We used Intel MBA [10] to enforce memory bandwidth limits at different levels. The results show that Base64, Markdown2HTML, and Image-Resize are relatively sensitive to memory bandwidth contention. Both average and tail latencies of these memory-intensive workloads are severely degraded while those of the Primes benchmark (which is computation-intensive) are not severely affected. For example, the tail latency of Image-Resize increases by 21.5× when setting MBA limit to be 20%. However, we observed that the increase of tail latency is even higher (28.9×) when no MBA partitioning is used (not shown in Fig. 6).

LLC Capacity. We used Intel CAT [10] to isolate the LLC access at different levels. The results revealed that most serverless benchmarks are not highly sensitive to LLC allocations, especially at low load. For instance, when the LLC



	CPU Time Contention			Memory Bandwidth Contention					LLC Contention			
	C1	C2	C3	M1	M2	M3	M4	M5	L1	L2	L3	L4
Base64 (Avg)	166.8%	1497.7%	1712.7%	131.3%	175.8%	261.0%	521.3%	663.9%	101.0%	101.0%	103.2%	106.8%
Base64 (Tail)	182.0%	1451.4%	1656.4%	200.0%	223.8%	262.4%	517.2%	671.7%	101.0%	103.0%	104.1%	107.0%
Primes (Avg)	114.9%	184.3%	250.0%	105.2%	108.8%	114.1%	126.2%	139.5%	100.7%	101.0%	102.2%	104.5%
Primes (Tail)	131.8%	272.4%	333.2%	136.6%	148.2%	153.5%	167.0%	177.8%	100.2%	100.3%	101.4%	105.6%
Markdown2HTML (Avg)	404.5%	1120.0%	1299.8%	100.1%	103.7%	492.1%	525.9%	806.7%	106.9%	136.6%	175.4%	230.6%
Markdown2HTML (Tail)	410.8%	1174.9%	1383.9%	121.8%	125.9%	463.8%	549.6%	828.0%	101.3%	126.6%	164.2%	229.8%
Sentiment (Avg)	128.7%	239.0%	349.3%	102.9%	115.8%	122.2%	135.7%	164.2%	100.7%	101.4%	103.7%	123.7%
Sentiment (Tail)	158.0%	321.7%	427.2%	112.9%	180.4%	183.4%	193.0%	204.6%	100.9%	101.5%	101.5%	128.8%
Image-Resize (Avg)	984.1%	2920.5%	3942.5%	133.9%	134.1%	898.6%	1743.7%	2400.6%	103.1%	106.1%	111.6%	121.8%
Image-Resize (Tail)	828.2%	2883.3%	3258.5%	125.7%	127.9%	1241.5%	1273.4%	2151.5%	103.7%	112.4%	114.3%	125.6%

1500 3000
Normalized Latency in Percentage
1000 2000
Normalized Latency in Percentage
120 180
Normalized Latency in Percentage

Figure 6: Performance impact of interference on shared resources including CPU time, memory bandwidth, and LLC capacity. The value of each cell is the end-to-end latency normalized to the latency with no interference. CPU contention levels (C1–C3) represent `cpu.shares` ratios 1:1, 1:2, and 1:4, respectively. Memory contention levels (M1–M5) represent Intel MBA limits of 80%, 65%, 50%, 35%, and 20%. LLC contention levels (L1–L4) represent Intel CAT limits of 16, 12, 8, and 4 bits, respectively .

capacity was cut to 50%, the latency increased for all benchmarks other than Markdown2HTML ranges from 1.5% to 14.3%. However, for Markdown2HTML, the latency increase due to LLC contention was substantially higher (2.3× for both tail and average latencies); we attribute this to data reuse among concurrent requests. Similar to memory bandwidth contention, we observed that the increase of tail latency is higher (4.4×) without LLC partitioning (not shown in Fig. 6).

Implication. *Performance isolation should be carefully assessed to prevent SLO violations due to resource sharing. However, when thousands of function containers are consolidated on a single server [1, 21], state-of-the-art resource partitioning fails to mitigate the performance interference, still with up to 8.3×, 21.5×, and 2.3× increase in tail latencies for CPU, memory, and LLC sensitive workloads, respectively.*

3.3 Breakdown of End-to-end Latency

To further explore how resource contention affects end-to-end latencies when overcommitting resources using an FG workload consolidation policy, we fixed the container memory limit to be 256 MB and increased the horizontal concurrency for function Base64 from 2 to 12. Fig. 7 shows the duration distribution of wait time, initialization time, and execution time. Same as the experiments in Section 3.1.1 and Section 3.2, we fixed the arrival rate to be 50% of the max load that sustains SLO. We observed that increasing the concurrency from 2 to 12 has three impacts: (i) it reduces the tail wait time by 49.5× from 1820 ms, (ii) it increases tail initialization time by 1.3× from 409 ms, and (iii) it increases tail execution time by 15.6× from 484 ms. The decrease in wait time is due to more concurrent containers and thus less

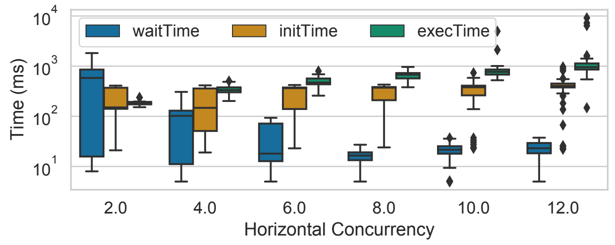


Figure 7: Breakdown of cold-start end-to-end latency.

queueing delay. The increase in execution time and initialization time is due to contention caused by running containers and the concurrent access to the data store for pulling function code/images, respectively. Wait times dominate the end-to-end latency when concurrency is between 2 and 4 while execution times are dominant when the concurrency ≥ 6 since the arrival rate is 4/s (as also shown in Fig. 3(b)).

Implication. *The three-phase breakdown of end-to-end latency varies with the concurrency-to-arrival-rate ratio.*

4 Related Work

Platform-level Studies. There have been end-to-end performance measurement studies on major cloud function providers and open-source serverless platforms from the viewpoint of a serverless customer. For instance, Wang et al. [22] examined several issues related to resource management, such as cold-start latencies and function placement strategies. A characterization study [19] on two-week production Azure Functions workloads showed function invocation frequencies, patterns, and resource needs, based on which customized keep-alive times are used for different functions. Figiela et al. [7] reported performance evaluation results (e.g., computation performance, network throughput, and instance lifetime) on major cloud function providers by treating serverless platforms as black-boxes. Lee et al. [13] evaluated the function throughput scalability for different types of workloads, and compared serverless computing with traditional virtual machines to study the cost-effectiveness for different workloads.

Server-level Studies. Instead of relying heavily on reverse engineering of commercial function services' behavior, several recent studies [18, 21, 23] characterize the server-level insights on serverless platforms. Shahrad et al. [18] identified several architectural impacts of certain FaaS workloads, which include 20× more mispredictions for branch predictors and 6× higher memory bandwidth. Yu et al. [23] evaluated key performance metrics, ranging from network latency and startup latency to execution time, on both commercial and open-source serverless platforms. Ustiugov et al. [21] studied the cold-start latency and function memory footprint, and found that the dominant factor is page-fault and poor locality in disk accesses. Each of the above measurement studies was done on a single node, while this paper presents results from a multi-node cluster.

5 Discussion

SLOs in Pricing Model. Existing FaaS pricing models [8, 9] and resource overcommitment via FG workload consolidation are favorable for workloads with loose latency objectives, and the customers can save money on high-idleness and bursty workloads. However, FaaS providers can shift the trade-off by charging customers for optional performance SLOs and using a more CG workload consolidation policy. The opportunity cost for using a CG policy (as shown in Section 3.1.2) can thus be offset by the SLO charge paid by the customer. In addition, when a coarse granularity is fixed, the containers can be distributed to multiple servers to avoid high-intensity packing. To maintain high system utilization, non-latency-critical or batch workloads can be placed together with latency-critical FaaS workloads. However, SLO violations can have various causes [16], including bugs in source code or function input variation, and that calls for research on the design of an interface for performance SLO negotiation between providers and customers.

Performance Interference Mitigation. Performance interference caused by resource sharing poses a significant challenge in migrating latency-critical workloads onto serverless platforms. As shown in Section 3.2, existing state-of-the-art resource allocation and isolation approaches fail to mitigate the interference in serverless computing. First, Linux CPU share is insufficient for CPU time distribution. A system-level enforcement mechanism should be applied (e.g., hardware thread priority or core affinity assignment) but prompt updates is needed for utilization efficiency in case of idle containers. Second, isolation or resource limit enforcement for other types of resources (e.g., LLC and memory bandwidth) should also be applied to improve the performance predictability of serverless functions. Although Intel CAT and MBA [10] can partition the LLC and memory bandwidth, the number of supported partitions is limited. There is a need to support thousands of partitions [1, 21] to enable fine-grained resource control that can be supported by this feature and a recent paper [17] has been working towards a fine-grained and scalable resource isolation mechanism.

Multi-tenancy in Decision-making. The decision to create a new container for executing a request (cold start) or to enqueue a request to an existing queue (warm start) depends on multiple factors, such as the request arrival patterns of all co-located functions and the difference between container initialization and wait time. As shown in Section 3.3, the horizontal concurrency and its mismatch with request arrival rates can affect wait-time behavior [18], initialization time and execution time. The initialization time for cold starts should be compared with the wait time for warm starts. Because of multi-tenancy, it is also needed to know if evicting other containers (in shortage of memory capacity) leads to frequent cold starts. Therefore, container-scaling conflict prediction and handling are required to minimize the global cold-start latencies in a multi-tenant serverless platform.

6 Conclusion and Future Work.

Our measurement study reveals that serverless FaaS is not yet ready to serve latency-critical workloads with stringent performance objectives. Our results show that to provide performance predictability, serverless providers must deal with the trade-offs in workload consolidation policies, avoid performance interference, and balance concurrency with arrival rates in a multi-tenant setting. For future work, we see the potential for serverless FaaS platforms to cater to latency-critical services, but there are open research challenges in areas such as fine-grained resource isolation, an interface for performance SLO negotiation between the FaaS provider and the customers, and fairness in multi-tenant decision-making.

Acknowledgment

We thank the anonymous reviewers for their valuable comments that improved the paper. We appreciate J. Applequist and K. Saboo for their insightful comments on the early drafts of this manuscript. This work is partially supported by the National Science Foundation (NSF) under grant No. CCF 20-29049; by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR), a research collaboration that is part of the IBM AI Horizon Network; and by the IBM-ILLINOIS Discovery Accelerator Institute (IIDAI). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or IBM. Saurabh Jha is supported by a 2020 IBM PhD fellowship.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2020)*. 419–434.

[2] Apache. 2021. OpenWhisk. https://github.com/apache/openwhisk.

[3] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The rise of serverless computing. *Commun. ACM* 62, 12 (Nov. 2019), 44–54.

[4] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware resource partitioning for multiple interactive services. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*. 107–120.

[5] Christina Delimitrou and Christos Kozyrakis. 2013. iBench: Quantifying interference for datacenter applications. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC 2013)*. 23–33.

[6] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Cristina L. Abad, and Alexandru Iosup. 2021. Serverless applications: Why, when, and how? *IEEE Software* 38, 1 (Jan. 2021), 32–39.

[7] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, and Maciej Malawski. 2018. Performance evaluation of heterogeneous cloud functions. *Concurrency and Computation Practice* 30 (2018).

[8] Geoffrey C. Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2017. Status of serverless computing and Function-as-a-Service (FaaS) in industry and research. In *Proceedings of the 3rd International Workshop on Serverless Computing (WoSC 2017)*.

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

[10] Intel. 2021. Intel RDT. https://github.com/intel/intel-cmt-cat.

[11] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2018. Serving deep learning models in a serverless platform. In *Proceedings of the 2018 IEEE International Conference on Cloud Engineering (IC2E 2018)*. 257–262.

[12] Joanna Kijak, Piotr Martyna, Maciej Pawlik, Bartosz Balis, and Maciej Malawski. 2018. Challenges for scheduling scientific workflows on cloud functions. In *Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD 2018)*. 460–467.

[13] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. 2018. Evaluation of Production Serverless Environments. In *Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD 2018)*. 442–450.

[14] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the 2015 ACM/IEEE 42nd International Symposium on Computer Architecture (ISCA 2015)*.

[15] PaaS Pricing. 2021. https://aws.amazon.com/fargate/pricing/.

[16] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: Intelligent fine-grained resource management for SLO-oriented microservices. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*. 805–825.

[17] Haoran Qiu, Yongzhou Chen, Tianyin Xu, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2021. SLO beyond the Hardware Isolation Limits. arXiv:2109.11666 [cs.OS]

[18] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of Function-as-a-Service computing. In *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO 2019)*. 1063–1075.

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

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

[21] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. 559–572.

[22] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC 2018)*. 133–146.

[23] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with ServerlessBench. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC 2020)*. 30–44.