Providing Performance SLO Guarantees for Multi-tenant Serverless Computing

Haoran Qiu, Beitong Tian, Ragini Gupta
University of Illinois at Urbana-Champaign
{haoranq4,beitong2,raginig2}@illinois.edu

Abstract

Serverless computing, as a way to construct the services that enable developers to build more agile applications so they can innovate and respond to changes faster, has been heavily invested by all major cloud providers in the form of Function-as-a-Service (FaaS). In contrast to traditional cloud service architectures, the application logic written by the developers is running in stateless compute containers that are event-triggered, ephemeral, and fully managed by the cloud providers. However, the unique characteristics of FaaS workloads make the performance predictability challenging. No cloud provider allows application owners to specify performance service-level objectives (SLOs), which hinders the adoption of serverless computing to latency-critical applications (e.g., Web services and machine learning model serving). This project aims to close the gap and provides performance SLO guarantees in a multi-tenant serverless computing platform. The results demonstrate that our proposed resource management framework achieves almost 2x better performance than the comparison baseline (i.e., OpenWhisk's default resource manager), thus optimizing the user-defined SLOs. A real-time web-based profiling dashboard is also implemented to visualize the serverless computing platform performance under different configuration options.

1 Introduction

Serverless computing has been a recent emerging paradigm in cloud computing [39, 42, 52]. It is typically defined as the Function-as-a-Service (FaaS) model where a user application is disintegrated into smaller *triggers* (i.e., events) and *action* (i.e., functions) that are hosted on a seamless platform for execution [41]. Serverless computing enables a new way of building microservice-based applications [3, 4, 33], having the benefit of greatly reduced operational complexity. It has been instrumental in addressing key features in cloud computing such as auto-scaling and loose-coupling of monolithic systems into smaller services (i.e., microservices). Serverless computing follows a pay-as-you-go model allowing the end users to only be concerned about their application functionality where they are charged for the resources and time allocated for running the functions without the associated cost for the server idle time.

It is worth mentioning that the concept of serverless computing does not imply "no-servers" in the framework but rather accounts for no scaling management or provisioning of servers (as a customer) in a cloud computing paradigm. In a typical serverless computing scheme, the developer is only responsible for uploading the code or the function image[1] that needs to be executed. On the other hand, the cloud service providers have a wider range of responsibilities spanning across the management of datacenters, cloud resources, and runtime environment.

1.1 Serverless Computing Architecture and Workflow

A serverless computing platform is an event-driven compute platform that runs function codes in response to events or direct invocations (i.e., client requests). Figure 1 shows the high-level architecture of a serverless computing platform and the request serving workflow. The serverless computing platform consists of a central controller (master node) and a group of invokers (worker nodes). Each function is packaged into and run as containers (or Pods in Kubernetes). The controller makes scheduling decisions on container placement, request routing, and load balancing. The invoker will execute the function after it gets the request from the controller. After a period of time (defined by a keep-alive timeout value), the container will be invalidated or evicted from the invoker if there is no subsequent invocations for that function.

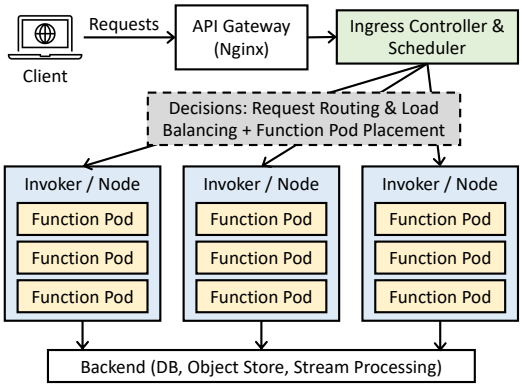


Figure 1. Serverless computing platform architecture and request-serving workflow

The workflow for serving a request is: (i) A client sends a request to invoke function F to the API gateway. (ii) If the function pod of F is kept alive, then the controller sends the

[1]The container entirely encapsulates the serverless function (i.e., libraries, handler code, OS, runtime, etc.) so that all the customer needs to do after that is point an event at it to trigger it.

invocation to the associated invoker (a "warm-start"). (iii) Otherwise, the scheduler chooses an invoker on which the function pod can be initialized and send the invocation of F to that invoker (a "cold-start"). (iv) Depending on application logic, F may use additional back-end services.

1.2 Life Cycle of a Function Container in OpenWhisk

Figure 2 illustrates an overview of different stages during the life cycle of a container. The way function invocation works in OpenWhisk is that a pair of unique <user, function> is associated with a couple of containers and a container can be reused for the function's multiple invocations indentified as the <user, function> pair. In order to obtain a highly scalable performance and low-latency response, container scheduling, reusing and caching is of prime importance. A cold-start will create a new container. Once the container is created, it will be initialized first. In the initialization phase, the function code is loaded into the container and an initialization function (`/init`) is called to prepare to execute their functions (e.g., initialize the language runtime, prepare database connection, caching, etc.). Creating containers is an expensive process, a pool of "pre-warmed" containers is maintained that are ready to run for workloads with frequently used language runtimes. Finally, the function is executed and the results are returned to the user. Once the function returns, the container is kept on the "running" phase for a small period of time (called grace period or keep-alive timeout value in the order of a few milliseconds). Within this period, if there is another request from the same user, this container can be reused immediately. That is called the "hot" path. The container is suspended after the expiration of the grace period. After suspension, any request will then follow a "warm" path to resume the container. If the suspended container is idle for a long time (longer than the keep-alive timeout value), it will be destroyed.

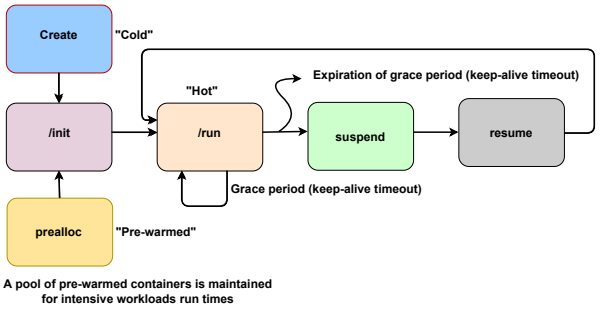


Figure 2. Life-cycle of a function container in OpenWhisk

1.3 Motivation and Problem Statement

Despite the benefits brought by serverless computing, no existing cloud provider allows application owners to specify performance service-level objectives (SLOs), which hinders the adoption of serverless computing to latency-critical applications (e.g., Web services and machine learning model serving). However, significant performance variation is observed for running FaaS workloads which ranges from a few milliseconds to seconds. In this project, we aim to close the gap by proposing a resource management framework for a multi-tenant serverless computing platform, which provides service-level objectives (SLOs)[2] to latency-critical serverless applications without over-provisioning. The resource management decisions for each function consist of (i) the number of containers to spawn for the function, i.e., the concurrency, and (ii) the size of each container, e.g., the memory and CPU limit. There are other decisions in the serverless computing platform such as request scheduling, request load balancing, request admission control, and container placement, but they are out of scope of this project.

1.4 Challenges

Based on the survey papers on serverless computing and characterization papers [5, 9, 18, 24, 27, 32, 43] on FaaS workloads, we summarize the following three challenges that worsen the performance unpredictability problem of FaaS workloads:

Costly but inevitable cold-start latencies. Despite recent advances such as AWS Firecraker [1] and snapshot-based approach [46], the runtime or sandbox initialization latencies still remain and can be substantial, compared to the execution times of each function. The SLO of each latency-critical function typically cannot afford several seconds of increase in end-to-end latency. However, unless there is a perfect future workload predictor, cold-start is inevitable if cloud providers do not want to keep unused containers forever in the memory.

Diverse FaaS applications. FaaS applications include short-lived scripts (such as machine learning inference, Web API serving, and IoT applications), and parallel services in big data processing, HPC in cloud, and scientific computing applications. The execution times of functions range from milliseconds to minutes (i.e., 5 orders of magnitude).

Multi-tenant cloud environments. Cloud providers offer their services to applications from different cloud users, allowing them to share the underlying infrastructure to increase the resource utilization. However, this comes at the cost of competing for limited resources. Although recent papers propose machine learning based resource management for each applications, a central controller is not scalable given the number of functions submitted to a cluster, while per-application controllers competing for shared resources may lead to more SLO violations in the worst case.

2 Approach Overview

Figure 3 shows an overview of our proposed solution. Our approach consists of multiple techniques to tackle the three main challenges mentioned in Section 1.4.

[2] An example of the types of performance SLO this project uses is: 99% of the requests are processed within 100ms.

Application categorization and prioritization. We categorize FaaS applications into latency-critical (LC) and best-effort (BE) as most cloud providers do for client VMs. LC applications will have client-defined SLOs for end-to-end latencies and each of them will be given a priority. The lower-bound for scaling down is at least 1 container, which eliminates cold-starts (Challenge 1). Clients will pay for the cold-start elimination and application priority. BE applications do not have SLO requirements and they are assigned the lowest priority. Upon resource scarcity, BE applications will be compromised first to avoid SLO violation of LC applications.

Per-application resource controller. Due to the diversity of FaaS applications (Challenge 2), we will use per-application resource controller (1 in Figure 3) to manage the resources allocated to it. In specific, it manages the number of CPU cores and the amount of memory associated with each instance, as well as the number of instances (i.e., concurrency). There are ML-based solution (e.g., FIRM [40] and Sinan [53], as mentioned in the related work), and also static approach (e.g., Heracles [31], PARTIES [10], CLITE [38]). We will focus on ML-based approach because of the highly dynamic environment in the cloud and the varying behavior of FaaS workloads, to avoid painstakingly tuning heuristics or thresholds.

Per-system-pool resource manager. To resolve conflicts in a multi-tenant environment (Challenge 3), a resource manager (2 in Figure 3) will gather global information on resource availability and coordinate the resource allocation to multiple per-application resource controllers. By doing this, each per-application resource controller can compete for shared resources by proposing resource requests and the resource manager can grant resources based on SLOs and priorities. We divided the cluster into multiple system pools, each of which manages multiple FaaS applications, to tackle the scalability problem of a centralized resource manager.

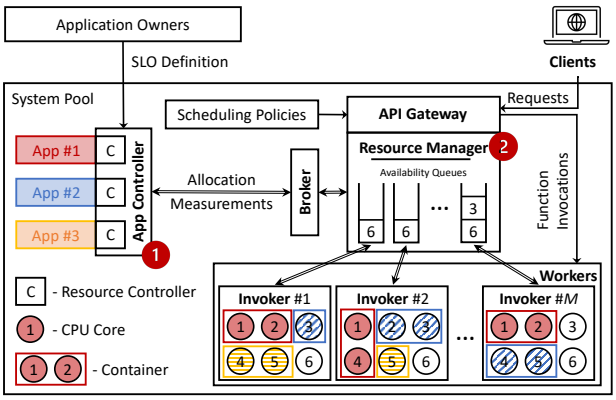


Figure 3. Overview

Our implementation is based on OpenWhisk [51], an open-sourced serverless computing platform. We use open-sourced FaaS benchmarks [44, 54] and profilers [44] for running the workloads and retrieving the measurements. The scope of the project consists of two main parts: (1) ML-based resource controllers for functions, and (2) coordination of asynchronous function resource controllers based on priorities and SLO requirements. This is challenging because of the asynchronous nature of distributed systems in practice. Consider an example of two applications A and B, where A has higher priority. If A's resource request arrives later than B's, then the resources could be granted to B, whose priority is low.

2.1 Application Controller

In our design, the resource allocation of each function or each chain of functions will be controlled by an application controller (represented as 1 in Figure 3). The input to the application controller is the measurements on the function container resource usage, workload characteristics (i.e., invocation frequency), and the history performance profiles. The output of the application controller is the resource allocation decisions including the size of the containers (e.g, memory and CPU limits) and the number of containers (i.e., function concurrency). The output is then sent to the central resource manager as the resource request. We plan to use an ML-based approach with a supervised learning algorithm trained with the profiling database consisting of mappings of resource allocation combinations, invocation frequency, and the performance statistics. The algorithm for the application controller is shown in Algorithm 1.

Algorithm 1 Application Resource Controller

Require: Function Profile db, Function ID fid
Require: $isProfilingBased$, Current Measurements m
Require: Trained random forest rf, Function SLO
1: **if** $isProfilingBased$ **then**
2: $perfList \leftarrow db.get(fid, m.workload)$
3: $containerSize, concurrency \leftarrow \text{argmax}(perfList)$
4: **else**
5: $containerSize, concurrency \leftarrow rf(fid, m, SLO)$
6: **end if**
7: **return** containerSize, concurrency

We first implemented a profiling-based approach (in our midterm report). Given the workload, we dynamically choose the resource allocation based on history performance profile database. Currently, we only support single-function applications, we leave the implementation for function-chain applications to the next stage. After that, we also implemented an ML-based application resource controller based on a simple supervised learning algorithm called random forest due to its relatively good performance among all supervised learning algorithms [7]. Random forests are an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of

the classes (classification) or mean/average prediction (regression) of the individual trees.

In the case of resource management for performance optimization, we model the training of such an application resource controller as a regression problem. The "class" or output variable y is the end-to-end latency of a function invocation. The "features" or input variable X is a list of resource controlling knobs including CPU limit, memory limit, vertical concurrency and horizontal concurrency. The training dataset is collected during the offline profiling process. Compared to the profiling-based approach, the ML- based application controller has similar performance because the random forest algorithm will learn the pattern from the training dataset shared with the profiling database. However, an ML-based approach is preferable because both the workload and the environment in serverless computing could change from time to time, not to mention the update frequency of FaaS applications. Traditional heuristics- or threshold-based approaches may suffer from painstakingly tuning and testing of the heuristics, which could be repetitive and is a waste of human effort.

2.2 Central Resource Manager

In a datacenter with thousands of physical machines, it may not be scalable to have a central resource manager of a single node or a couple of state-replicated nodes (with consensus protocols such as Raft or Paxos). Therefore, due to scalability concerns, we choose to have a per-system-pool central resource manager, where each system pool can contain hundreds of machines[3]. In our design, the central resource manager (represented as 2 in Figure 3) is responsible for coordinating the resource allocation for all applications. It receives the resource allocation requests and grants/denies the requests based on application priorities. In a multi-tenant environment, all functions are consolidated on limited number machines to drive up system resource utilization. However, all application controllers are competing for limited shared resources to guarantee each application's SLO. Without global states, no SLO will be satisfied in the worst case. For example, application A and B are co-located on the same physical machine. To satisfy their SLOs, both A and B requires 40% of the CPU time and both SLOs can be satisfied. Now consider a case when the request arrival rates of both A and B are increased such that both A and B require 60% of the CPU time. Selfish or greedy application controllers without a communication mechanism or shared-information will ask for more CPU times at the same rate (assuming they are synchronous) and may end up with 50%-50% division of the CPU time, which violates both A's and B's SLOs.

[3]This is a reasonable assumption given that a typical Kubernetes cluster in modern cloud datacenters consists of 5 to 500 machines, and Kuberentes cluster with a central API server or management node can support up to 5000 nodes. Reference: https://learnk8s.io/kubernetes-node-size

Algorithm 2 Central Resource Manager

Require: Max quota of globally available resources: $W_{max} = f(n_{cpu}, n_{memory}, n_{storage}, n_{bw})$
Require: Application profile $A_i \in$ Application ID set (A_{id})
Require: Application priority (δ_i)
Require: Resource demand from application A_i: $A_{i,n} = f(n_{cpu}, n_{memory}, n_{storage}, n_{bw})$
Require: Priority Queue of Application IDs in 10ms, Q_t
Require: Number of applications in queue Q_t: k
Sort Q_t in decreasing order of Application's δ_i
Init key-value store ($StatusStore$) for status of application resource request: $<A_i : Status>$
$R_{all} = cpu, memory, storage, bw$
for $i \leftarrow 1 \ldots k$ **do**
  $dominant_i \leftarrow \arg\min_{r \in R_{all}}(W_{max,n_r}/A_{i,n_r})$
  if $dominant_i < 1$ **then**
    $StatusStore(A_i) \leftarrow Denied$
  else
    $StatusStore(A_i) \leftarrow Accepted$
    $continue$
  end if
  $W_{max,n_{cpu}} \leftarrow W_{max,n_{cpu}} - A_{i,n_{cpu}}$
  $W_{max,n_{memory}} \leftarrow W_{max,n_{memory}} - A_{i,n_{memory}}$
  $W_{max,n_{storage}} \leftarrow W_{max,n_{storage}} - A_{i,n_{storage}}$
  $W_{max,n_{bw}} \leftarrow W_{max,n_{bw}} - A_{i,n_{bw}}$
end for
Requeue Q_t for the next interval of 10 ms
return $StatusStore$, W_{max}

On arrival of an application (A_i) elimination event:
$W_{max,n_{cpu}} \leftarrow W_{max,n_{cpu}} + A_{i,n_{cpu}}$
$W_{max,n_{memory}} \leftarrow W_{max,n_{memory}} + A_{i,n_{memory}}$
$W_{max,n_{storage}} \leftarrow W_{max,n_{storage}} + A_{i,n_{storage}}$
$W_{max,n_{bw}} \leftarrow W_{max,n_{bw}} + A_{i,n_{bw}}$

The central resource controller, on the other hand, will gather global information on resource availability and coordinate the resource allocation to multiple per-application resource controllers, based on application priorities. In the previous case, if application A has higher priority, then A will be allocated 60% of the CPU time and B will be migrated to another machine or scaled out to create another container, instead of scaling up to 60% on the same machine with A.

Here, we envision a central resource manager to operate periodically (every 10 milliseconds) to accommodate incoming requests from the user submitted function/application every 10 ms. These requests are stored in a priority queue that are ready to be executed but are not yet allocated with resources. The central resource manager allocates resources based on the application priorities and their execution times. The algorithm is presented in Algorithm 2. The algorithm proposes allocation based on global view of resource avail abilities and application priorities.

Table 1. FaaS benchmark profiling parameters.

Parameter	Value
Function type	base64, prime, json
Invocation rates	10, 20, 30, 40, 50
Cluster setup	single-node, ten-node
Memory size (MB)	256,320,384,448,512
CPU-share	128,256,384,512,640,768,896,1024
Horizontal concurrency	2, 4, 6, 8
Vertical concurrency	0, 1, 2, 4

3 Benchmark Profiling Results

We have set up two OpenWhisk clusters: (i) a single-node cluster deployed on a physical machine with 8 CPUs of Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz model and 16GB memory, and (ii) a multi-node cluster deployed on UIUC CS VM server farm with 1 master node and 9 invokers. To understand how function performance is affected by factors including container size (e.g., CPU share and memory limit), vertical concurrency (i.e, the number of requests that can be concurrently sent to a container), and horizontal concurrency (i.e., the number of containers spawned for a function), we did intensive profiling (see Table 1) on selected benchmarks from [44]: (i) base64 (CPU-intensive workload doing string transformation), (ii) json (memory-intensive workload doing JSON input object processing), and (iii) primes (CPU- and memory-intensive workload doing large prime number computation). We plan to use more benchmarks including real-world FaaS applications from ServerlessBench [54] in the next stage of the project. The profiling code for different application workloads and parameters is available on Github (Git link anonymized for blind review, 2021).

Before go into the discussion of the results, three performance metrics are defined as follows: (i) Wait time is the time spent waiting in the internal OpenWhisk system. This is roughly the time spent between the controller receiving the activation request and when the invoker provisioned a container for the action. (ii) Initialization time is the time spent initializing the function. If this value is present, the action required initialization and represents a cold-start. A warm activation will skip initialization, and in this case, the annotation is not generated. (iii) Execution time is the time for processing the function. (iv) End-to-end latency corresponds to the time between when a controller receives an invocation request and the time when the controller receives the result from the invoker.

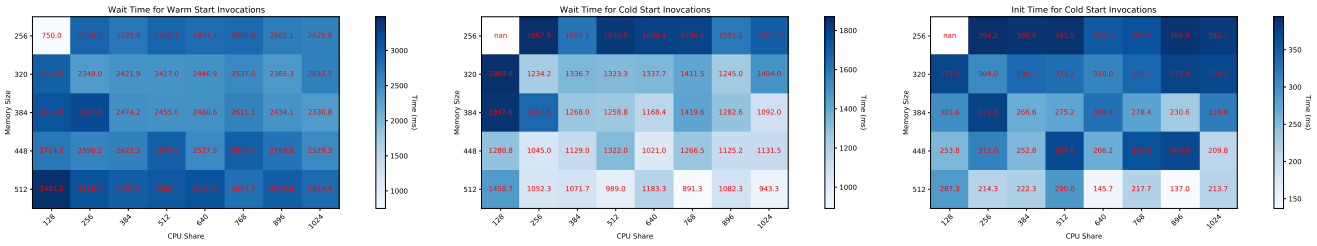
3.1 Container Size

We first study the effect of container size on the function invocation performance. Figure 4 and 5 illustrate the different results for different parameters values to validate how memory limit and CPU share affect the function invocations. We consider the following types of performance metrics: wait time for warm-start and cold-start, initialization-time for cold-start, execution time, end-to-end latency for warm-start and cold-start. For execution time, the results that we show are from cold-starts because the execution time does not vary with the invocation type.

Running base64 function benchmark with invocation rate = 10 on a single-node cluster. Without further specification, the invocation of actions (in all following figures) is chosen to be in a uniform distribution over a period of time (e.g, [2, 7], in total 5 seconds). Figure 4 demonstrates the different profiling results for base64 function on a single node cluster when the invocation rate is set to 10. Figure 4(a) and (b) demonstrate the average waiting time for warm- and cold-start invocations where cold-start takes a much smaller time than warm-starts. This is because in cold-starts, the invocation does not need to wait for an existing container to finish executing the function, while in warm-starts, the waiting time comes from queuing to wait for an available container. As the CPU-share and assigned memory to the single node cluster increases, the average waiting time reduces linearly. The average standard deviation for waiting time for both warm-start and cold-start invocations are 1344.62 and 609.43 ms respectively.

Additionally, for a given CPU-share and memory,the initialization time for cold-start invocations is less than the wait time. Low startup (initialization) time indicates that the system takes less time to prepare for the request handling. From the figure, it is evident that the end-to-end latency for warm-start invocations is much larger than that for the cold-start invocations when using our profiling invocation schedule. Since base64 application is a highly CPU-intensive task the execution latency is also quite high (900 ms). Our experiments show that under low arrival rates (e.g., when rate = 10), varying CPU-share has less impact on execution time of the containers, partly also due to the execution time is not the dominant factor in the end-to-end latency. CPU shares play a larger role when the arrival rate is much higher (such as 50 per second). The average standard deviation for the execution response time for warm-start and cold-start invocations is computed to be 60.59 and 72.90 ms. The smaller measure for standard deviations indicate stable consistency in execution response times.

Running base64 function benchmark with invocation rate =10 on a multi-node cluster. Figure 5 shows the different results for invoking base64 functions on a 10-node OpenWhisk cluster with various combinations of allocated memory size and CPU-share in the range shown in Table 1. From the Figure 5(a) and (b), we can infer that for a given memory size (512MB) and CPU share (1024) the wait time for warm-start invocations on a cluster increases by a factor of 71% whereas the wait time for cold-start invocations reduces by a factor of 88% for the multi-node cluster in comparison with a single-node cluster. This is due to the additional overhead caused by fresh start for containers across



(a) Avg. wait time (warm) (b) Avg. wait time (cold) (c) Avg. init time

(d) Avg. execution time (e) Avg. e2e latency (warm) (f) Avg. e2e latency (cold)

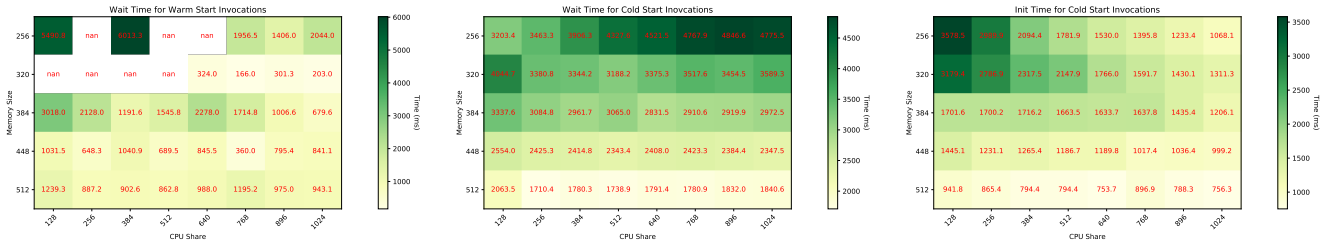
Figure 4. Profiling of base64 benchmark on a single-node cluster with invocation rate=10.

all the nodes in the distributed cluster that impacts the overall waiting time for cold-start invocations. The overhead for waiting time is also due to the network bottleneck in communication between the nodes to handle various dependencies. The sudden spike in Figure 5(a) for memory limit at 384 MB and CPU-share at 256 happens due to noise or an outlier observation. Moreover, for a given memory size, as the CPU-share across nodes increase, the waiting time reduces proportionally by a factor of 16% which follows a similar trend as observed in the single-node system. For Figure 5(c), increasing the CPU-share reduces the initialization time for cold-start invocations by 25% whereas increasing the memory limit increases the initialization time. That is because the larger the memory limit, the less concurrent containers will run in each invoker, thus, the more CPU shares each container can use. For Figure 5(d), we can observe that increasing CPU-share does not have significant impact on the execution latency for cold-start invocations. Similarly, for Figure 5(e) and (f), it is evident that as the CPU-share increases, the end-to-end latency is only accelerated by a factor of 13% and 6% for warm-start and cold-start invocations. With the increase in memory limit, a significant change in the latencies is observed. The end-to-end latency for cold-start increases with increase in memory limit. This is because the memory limit currently controls the number of containers that are created (i.e, the number of containers = $total\_capacity/memory\_limit$). In case of cold-start end-to-end latency, for memory limit 256MB, eight containers are created (since the total capacity is 2GB) and for 512MB, four containers should be created. However, the invocation rate (10Mbps) is much less than the container creation time, leading to increase in the cold-start time for those four and eight containers.

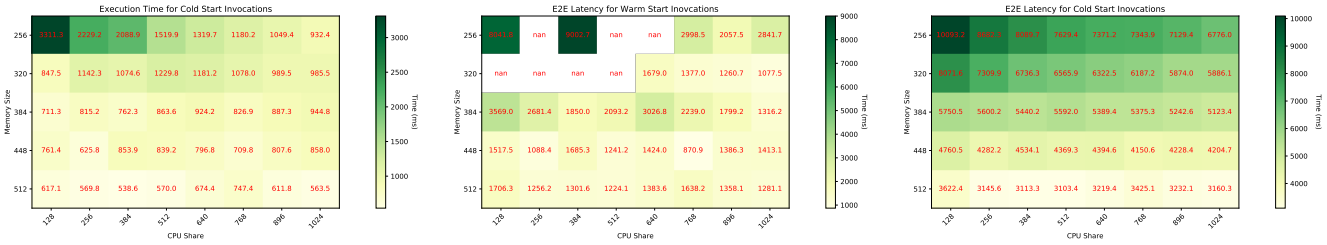
Due to page limit, in the Appendix Section C, we will discuss more profiling results for different invocation rates set to 20, 30, 40 and 50, as well as for more benchmarks (JSON and Prime numbers). A general trend observed from the figures is that as the invocation rates for the functions increases, the execution time, initialization time and end-to-end latencies for cold-start invocations are higher than the results obtained for invocation rate as 10. The impact of external factors such as CPU share and memory limit are the same as observed in previously discussed results. Additionally, for the other two FaaS benchmarks (json and primes), it was found they are less CPU-intensive and therefore, their latencies are much smaller than the ones obtained for base64 workload.

3.2 Horizontal Concurrency

Horizontal concurrency refers to the number of containers spawned for executing a burst of requests concurrently for a function. This is an important parameter in serverless computing platforms as it provides the serverless functions the ability to scale out to cater the increasing arrival rate, and to scale in for saving resource utilization (containers do not need to be always-on). In our evaluation, we choose the horizontal concurrency to be 2, 4, 6, 8, and measure the performance metrics for three FaaS micro-benchmarks: primes, base64, and json. Table 2 shows the results for the warm-start invocations. In all three benchmarks, the waiting time dominates the end-to-end latency. As the horizontal concurrency level decreases from 8 to 2, the waiting time for each invocation increases significantly. The waiting time at concurrency level 2 is more than two times compared to the waiting time at concurrency level 8. However, the function execution time decreases when the concurrency level drops



(a) Avg. wait time (warm) (b) Avg. wait time (cold) (c) Avg. init time



(d) Avg. execution time (e) Avg. e2e latency (warm) (f) Avg. e2e latency (cold)

Figure 5. Profiling of base64 benchmark on a multi-node cluster with invocation rate=10.

from 8 to 2. This is because when there are less concurrency containers, the CPU time is abundant for execution of limited number concurrent containers. Since the waiting time dominates the end-to-end latency, a higher concurrency level leads to better end-to-end function invocation latency.

Table 2. Profiling results on horizontal concurrency.

Wait Time (ms)	Horizontal Concurrency Level			
	8	6	4	2
prime	1767±423	3085±1224	3078±808	4665±1377
base64	4276±1144	4916±1107	5737±925	9205±696
json	122±167	81±104	63±74	66±76

Exec Time (ms)	Horizontal Concurrency Level				
	8	6	4	2	1
prime	643±17	568±22	394±14	252±21	194±18
base64	981±5	804±16	606±7	444±2	301±9
json	2.2±0.4	2.3±0.5	1.8±0.6	1.7±0.5	1.3±0.7

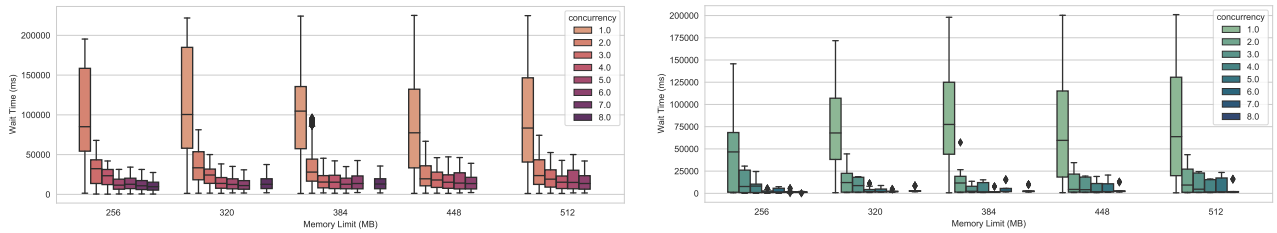
Latency (ms)	Horizontal Concurrency Level			
	8	6	4	2
prime	2410±406	3654±1246	3472±823	4917±1045
base64	5258±1139	5720±1106	6343±924	9649±696
json	125±167	83±105	65±75	68±76

Since the waiting time dominates the end-to-end latency of each function invocation, we dig deeper into how horizontal concurrency affects the waiting time in both warm-starts and cold-starts. Previously we have shown that the long waiting time in warm-starts is attributed to the queueing time. After a request is scheduled to an invoker and directed through Kafta messaging mechanism to that invoker, each request spends much longer time on waiting the existing requests in the queue (at each invoker side) when the concurrency is lower (as also shown in Figure 6(a)). A higher concurrency helps reduce the waiting time as the incoming workload is spread to more queues in each invoker. Note that when there are multiple available worker processes (and thus worker queues) at an invoker, the invoker will randomly select a queue to direct the request to.

In terms of cold-starts (as shown in Figure 6(b)), we see similar trends, i.e, higher concurrency level leads to lower waiting time. Recall that the waiting time in cold-starts is mostly the time to provision the container and load the function into the container. When the horizontal concurrency level is 1, the waiting time is basically to pull the function image or function source code from the remote database. But when the concurrency level is higher than 1, the function image or the source code is already cached at the invoker, so the waiting time is cut significantly. From concurrency level = 2, adding more concurrency containers does not help reduce the waiting time because of the same reason.

3.3 Vertical Concurrency

Vertical concurrency refers to the number of concurrency requests each container can serve. If the value is greater than 1, the controller will send the request to the warm container (which is processing a previous request) and the request will be queued at the warm container until the previous request has been processed. If the value is equal to 1 (the default setting in OpenWhisk), the controller will create a new container instead of sending the request to the warm container (which is processing a previous request). A higher-than-one vertical concurrency may benefit the function if the function can tolerate multiple activations being processed at once. We



(a) Waiting time (warm-starts)

(b) Waiting time (cold-starts)

Figure 6. The effect of horizontal concurrency and memory limit on the waiting time for both warm-starts (left) and cold-starts (right).

found that most functions in the benchmarks do not tolerate that and will suffer the queuing time when the queued requests are waiting for the current request to be finished. The execution time could be up to 1 second, which significantly increases the end-to-end latency of subsequent requests due to waiting/queuing. Therefore, if we do not know in advance the execution time of a request, we choose not to waste time queuing at the container (i.e., vertical concurrency is 1). If we know the execution time of a request is smaller than the cold-start time, then we choose to send the request to the current warm container (i.e., vertical concurrency is greater than 1).

4 Monitoring Infrastructure

4.1 Online Visualization Platform

We cannot find an existing tool to help us easily get run-time CPU and memory usage of each container. These information can provide us more metrics such as the peak CPU and memory usage for profiling workloads and more details like the relative CPU usage among all containers for analyzing profiling results. In this case, we build our own monitoring and visualization tool to trace these data and also record important timestamps $t_1 - t_7$ of each function invocation as listed in Sec. 4.2. The monitoring tool architecture is shown in Figure 7.

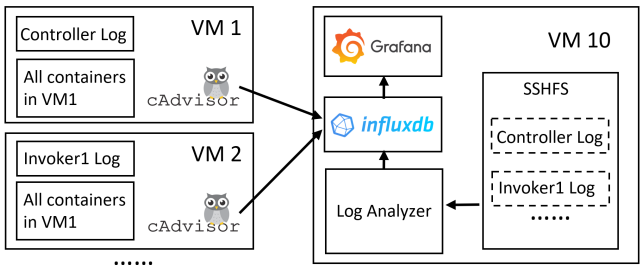


Figure 7. Monitoring tool architecture

On each VM, we used cadvisor[16] to collect CPU and memory usage of each container with a 2Hz sampling rate. The collected data will be sent to a VM and stored in the influxdb, a time series data base. In the same VM, we use SSHFS to map remote log files of the controller and invokers to local file system and use our customized log analyzer to parse the log. Parsed information are stored into influxdb. Finally, we use the open-source visualization tool, Grafana [17], for visualizing the data stored in the influxdb in predefined dashboards. An example of the visualization is shown in Appendix C (Figure 17). A FaaS profiler implemented for the online monitoring is available on Github together with our OpenWhisk codebase [4].

4.2 End-to-end Latency Modelling

The end-to-end latency depends on the internal processing flow in OpenWhisk and the total response time can be computed as an accumulated delay for each step. A sequence of steps is illustrated as follows:

- Timestamp for forwarding incoming user request to the Controller: t_1
- Timestamp for posting to Kafka (where Kafka message distributing system is responsible for communication between controller and the invokers): t_2
- Invoker receiving the message from Kafka: t_3
- Running cold start begin (where the Invoker executes the *docker run* command): t_4
- Running `\init` to initialize the function: t_5
- Timestamp for execution begin and end: t_6, t_7

Therefore, we have:

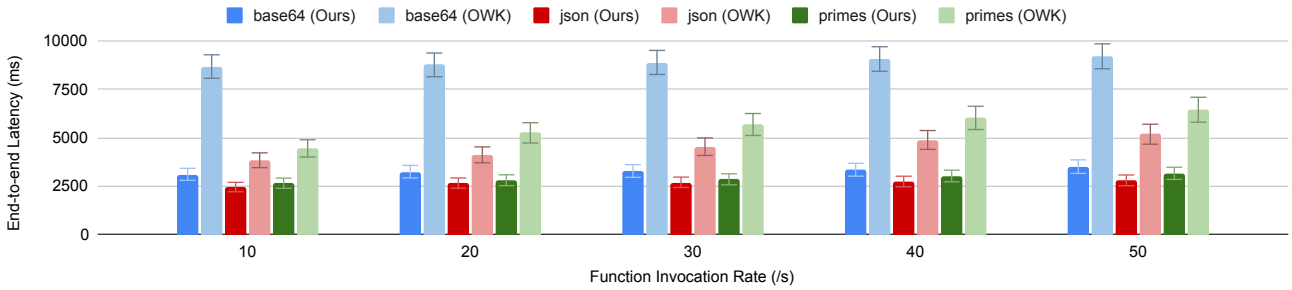
- Total waiting time (RT_w)= $t_1 \longrightarrow t_2 \longrightarrow t_3 \longrightarrow t_4 \longrightarrow t_5$
- Total initialization time (RT_i) = $t_5 \longrightarrow t_6$
- Total execution time (RT_e) = $t_6 \longrightarrow t_7$
- Total end-to-end latency: $RT_{e2e} = t_7 - t_1$

5 Evaluation Results

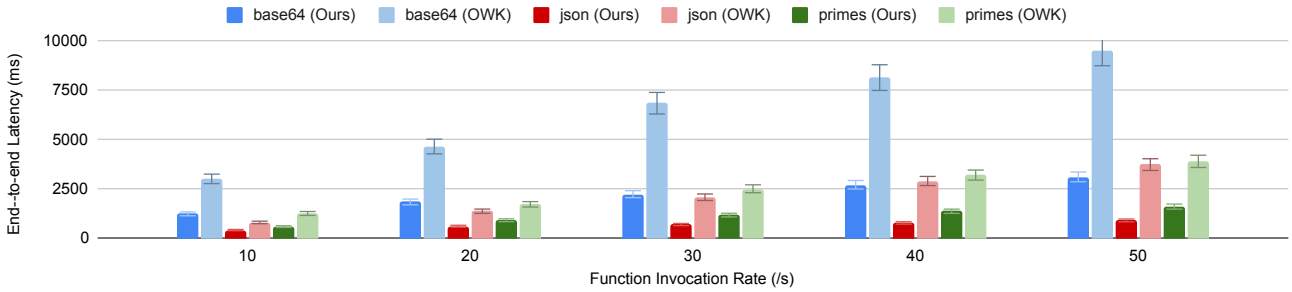
5.1 Evaluation Setup

As mentioned in Section 3, we have set up two OpenWhisk clusters, one is a single-node clsuter on a physical server; the other is a ten-node cluster on UIUC CS VM server farm. Our evaluation experiments are done on the multi-node cluster with 1 master node and 9 worker nodes. Our serverless workload benchmarks are the same as those on which we performed characterization and generated function profiles: `base64`, `json`, and `primes`. The function invocation follows

[4] https://github.com/James-QiuHaoran/openwhisk/tree/master/online-monitoring-tool



(a) Cold-starts



(b) Warm-starts

Figure 8. End-to-end latency comparison in cold-start and warm-start function invocations between our application controller and OpenWhisk's default way of resource management for three micro-benchmarks: `base64`, `json`, and `primes`.

a Poisson distribution with rate ranges from 10 to 50 per second [5]. We set the upper bound of the invocation rate to be 50 per second because based on the function trace analysis on a 14-day Azure production function trace dataset [45], 90% of the functions are triggered only 1.2 times per minute. It means that customers typically use serverless computing for those infrequent bursty/constantly-invoked workloads. The client is on a separate node from the ten-node OpenWhisk cluster to avoid potential resource contention between the client process and the OpenWhisk processes.

5.2 Per-Application Resource Controller

The resource allocation of each application is controlled by an application controller (represented as 1 in Figure 3). The output of the application controller is the resource allocation decisions including the size of the containers (e.g, memory and CPU limits) and the number of containers (i.e., function concurrency). The output is then send to the central resource manager as the resource request. However in this section, we plan to evaluate the effectiveness of resource allocation decisions separately, therefore, we disable the central resource manager first (i.e., by granting whatever resource requests from the per-application resource controller).

Figure 8 shows the end-to-end latency comparison between our application controller and the OpenWhisk's default way of resource management (denoted as OWK in the figure). For cold-starts (Figure 8(a)), the end-to-end latency does not vary too much as the invocation rate increases from 10/s to 50/s. This is because the dominant factor in cold-starts is the container provisioning and the function initialization time (based on our function benchmark characterization results), which does not change too much with the invocation rate. Although different processes are concurrently accessing the CouchDB for function images or source code (note that CouchDB stores all function call results including metadata and execution times), but we found that this is not the bottleneck (due to caching and database parallelism). The performance of all three benchmarks benefits from the per-application resource controller. Compared to OpenWhisk's default resource manager, our approach leads to performance improvement ranging from 1.5x to 2.8x.

For warm-starts (Figure 8(b)), the invocation rate affects the performance quite a bit. This is because waiting time is much more significant compared with the initialization time and the execution time of the function invocation. Most of the waiting time is spent at the invoker queue, and each function request is waiting to be processed after the previous function requests. Therefore, if the invocation rate is larger, the queueing time will increase significantly based on the queueing model. The end-to-end latency for function invocations managed by OpenWhisk's default resource manager is 1.9x to 4.1x larger than those managed by the per-application resource controller. By selecting the optimal combinations of CPU/memory limit and concurrency level, the controller

[5]The profiling dataset is collected by using a workload generator with uniform distributions.

will lead to lower end-to-end latency compared to the default CPU/memory limit setting and OpenWhisk's threshold-based concurrency scaling. The drawback of our approach is that the resource controller does not know how to manage resources for unknown functions or new type of workloads (the performance is based on the profiling dataset).

5.3 Per-System-Pool Resource Manager

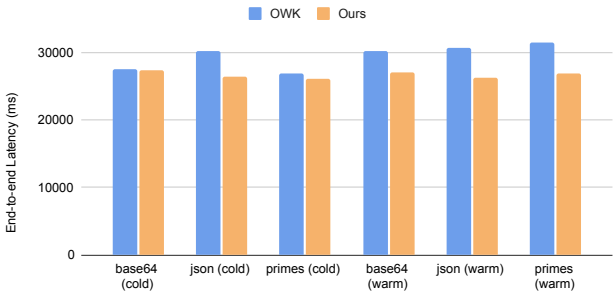
The central resource manager (represented as 2 in Figure 3) is responsible for coordinating the resource allocation for all applications. It receives the resource allocation requests and grants/denies the requests based on application priorities. In this experiment, we have a synthetic workload of three benchmarks. We set the priority of `base64` to be 0 (no SLO) and set the priorities of `json` and `primes` to be 1 and 2 respectively. The arrival rates of `base64`, `json`, and `primes` are set to be 10/s, 20/s, and 20/s respectively. Figure 9 shows the evaluation results of the central resource manager. We disabled our ML-based application resource controller so that we can find the improvement only from the central resource management. In terms of the end-to-end latency (Figure 9(a)), our approach outperforms OpenWhisk by up to 17%. There is little improvement on `base64` because `base64` has the lowest priority among all functions and it is not associated with any SLO. However, its performance is not sacrificed at all.

The improvement is larger for warm-start invocations compared to cold-start invocations because the latency does not vary much for cold-starts (recall our characterization finding in Section 3). The improvement mainly comes from the reduction of waiting times (see Figure 9(b)). By assigning priorities and allocating resources based on priority dynamically, we are able to achieve up to 17% reduction in latency. Similarly, the low-priority non-SLO-driven workload (i.e., `base64`) is not sacrificed as well.

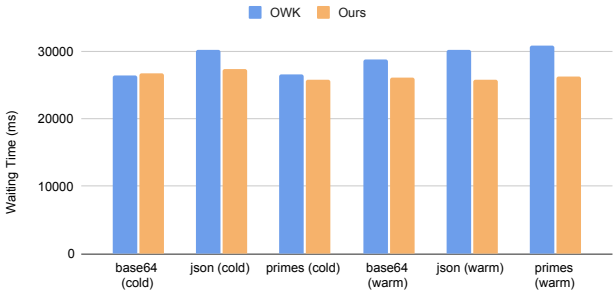
However, our approach relies on a correct and reliable priority assignment from the function owners. If the priority is set maliciously high (e.g., a function with a very loose SLO but very high priority), the performance of those functions with tight SLOs will be sacrificed.

6 Discussion and Limitations

What do your techniques gain you? In the original OpenWhisk design, the concurrency level is fixed. The CPU share is proportional to the user-defined memory limit. However, this causes significant performance variations (as shown in our evaluation results), which is detrimental for those latency-critical applications. Besides, the assumption of CPU share should be proportional to the memory limit is not valid, given the diversity of serverless workloads. Our per-application profiling-based resource allocation approach provides performance predictability. It optimizes for the user-defined SLOs with the history profiling database and makes the two main scheduling decisions: container size and the concurrency level. Traditional heuristics- or threshold-based resource management typically works like this: (1) one comes up with a clever heuristic and tests or tunes it until it is optimal; (2) after the workload changes or application is updated, one needs to re-tune the heuristic and test it; (3) the process is repeated again and again. With an ML-based approach, one does not need to go through this painstaking heuristics tuning and re-tuning process, which could be a significant waste of human expert effort.



(a) End-to-end latency



(b) Waiting time

Figure 9. Evaluation results of the central resource manager.

What do your techniques lose you? That is, what are the tradeoffs/limitations? Profiling-based approaches will not work if the workload changes or the underlying system infrastructure changes. Even if the next stage of our project is to use an ML-based resource controller, it may not give the optimal results if the inputs to the neural network never appears in the training datasets. In addition, our approach may suffer long profiling time if the number of functions is huge, which is a waste of time or monetary cost. In addition, our approach relies on a correct and reliable priority assignment from the function owners. If the priority is set maliciously high (e.g., a function with a very loose SLO but very high priority), the performance of those functions with tight SLOs will be sacrificed.

What do your experiments tell you? Our experiments support our idea of per-application resource controller, which can reduce performance variation and provide predictability for latency-critical applications. With the whole design (i.e., per-application controller provides local optimal solution for

resource management of an application and central resource manager supports communication between controllers and provides global optimal solutions based on priorites), the serverless computing platform is able to serve multi-tenant FaaS workloads.

Extended work on proposed cost model for serverless functions based on pay-per-request: The cost model for serverless computing platform depends on a variety of factors including data transfer (ingress/egress) charges, computational charge per second with the pre-configured memory and storage settings, and the cost of additional services (such as a BaaS database or MLaaS) used by the function. Thus, the cumulative cost incurred by serverless functions depends on pay-per-use pricing and the runtime cost for computational execution (e.g., memory/CPU usage). We can predict the cost model based on an estimate of cold start probabilities, which is deterministic in our case as we implement the resource manager to ensure compliance with SLOs. Probability of cold start (P_{cold}) is computed as the ratio of requests causing cold start to the total number of requests invoked during the experiment. To compute a pay-per-use pricing model for serverless functions, we first pre-define the cost per request for both cold- and warm-start invocations. Assuming the model parameters, we will then obtain: the number of requests ($N_{requests}$), the cost per warm-start (C_{warm}), cost per cold-start (C_{cold}), and thus the pricing model is formulated as follows:

$$Cost = N_{requests} \cdot ((P_{cold} \times C_{cold}) + ((1 - P_{cold}) \times C_{warm}))$$

This cost model is a preliminary design, which will be explored for optimizing it further in the future work.

7 Related Work

Recent studies has demonstrated various efforts taken to address some of the existing challenges in serverless computing environment such as global resource management, flexible scheduling of the functions, launch starts (cold-start delays), and performance isolation. In [28], the authors highlight different performance issues related to serverless computing due to several factors such as limited available resources (such as CPU and memory) for function invocation, number of concurrent requests, restricted runtime resources, and lack of optimized code. Although some efforts have been made to overcome a few of these challenges [41, 52], most of these works are recent and in a nascent stage that are still far from overcoming serverless computing issues in totality. Some of the existing work along these lines are discussed in the following sections.

7.1 Serverless Computing Platforms

Different commercial applications have been successful deployed on serverless computing platforms such as Google App Engine [11], Amazon Lambda AWS [21], Google Cloud Functions (GCF) [20], Kubeless [30] FaaS platform with Kubernetes. Different other open-sourced serverless computing platforms and profilers such as KNative [29], Apache OpenWhisk [51], FaaS-Profiler [44], Nuclio [36], Kubernetes Fission [15], OpenLambda [19], and OpenFaaS [13] prevail currently in the serverless computing domain, facilitating users to split their application across multiple simple services (i.e, the trend towards microservices). In [39, 42], authors elaborate on emerging challenges in serverless computing that prevent maximizing the best utilization of serverless computing including lack of proper development and testing tools, poor performance as infrequently used serverless functions have higher runtime with increased end-to-end response delay, resource limitation by cloud service providers, security and privacy vulnerabilities in serverless model.

7.2 Function Scheduling in Serverless Computing Platforms

In [49], the authors address the job scheduling of serverless environments by implementing a QoS enabled drop-in framework, called Sequioa. Sequioa allows the developers to define how the serverless functions can be deployed and prioritized using adaptable policies. Archipelago [47] is another delay-sensitive serverless computing framework that deploys a shortest-remaining-slack-first algorithm for scheduling serverless functions. Other function scheduling approaches such as GrandSLAM [26] are based on reorganizing function requests depending on the least slack time. However, it does not consider generic QoS level guarantees required by the applications. Similarly, FnSched [48], is another design for function-level scheduler to manage resource allocation across co-residing functions on each invoker depending upon the traffic variability. However, one underlying assumption of this framework is that the function execution times are constant which can be determined via detailed profiling for CPU-shares. This assumption, however, may fail in a public cloud setting, given the dynamic cloud environment and constantly changing client workloads. In another work [25], the authors deploy a cluster-level centralized and core-granular scheduler for serverless functions. The centralized design allows for a global view of the resources while the core-granularity helps in eliminating resource interference. This facilitates the adaptability of the proposed scheduler across various time sensitive applications via SLAs.

7.3 QoS-Aware Resource Management in Microservices

For smooth functioning of microservices, efficient run time resource scheduling is important to meet the QoS guarantees. Lately, machine learning based scheduling approaches have gained a significant momentum towards achieving this goal. The scheduling exploration space increases linearly with an increase in number of computational server resources (CPU cores, cache, network bandwidth, etc. Thus, machine learning based models play a pivotal role in proposing an optimal

solution for scheduling mechanisms. While machine learning is still very popular for applications like image or speech recognition, it's contribution towards resource scheduling is still limited and open for more research directions. In [40], the authors proposed an ML-based resource management framework, FIRM, to tackle the issue of microservices underutilization and SLO violations. The two-tier ML model is first, responsible for identifying the microservices that cause SLO violations and second mitigating those violations via dynamic reprovisioning. However, a few drawbacks of this work include its limited scalability with the number of applications, and no capability for identifying the SLO violations from sources such as global resource sharing and the hardware. Sinan [53], a QoS-aware resource management framework for mircoservices is another model that employs machine learning algorithms to reduce resource utilization while meeting the end-to-end QoS. Other static-threshold based resource allocation approaches have also been explored in the literature [10][31] to ensure resource sharing and management, but do not account for application dynamism. These approaches are either for single application or container-based long-running microservices do not suit for ephemeral and fine-granular serverless workloads.

7.4 Cold Start Management

One of the major bottlenecks in existing serverless computing paradigms is the performance degradation due to significant cold-start latencies (compared to function execution time). Typically, cold-start refers to the time it takes for a function logic to deploy along with it's associated dependencies. Different approaches have been explored in the literature to address this issue so far. The magnitude of the cold-start problem is illustrated in [50], where the authors measure the cold-start delay for two different benchmarks (I/O intensive using Tensorflow for image recognition and CPU intensive for Fibonacci) on Amazon AWS lambda. The results demonstrated the cold-start delay for I/O intensive workload is almost 10 times more than the CPU-intensive benchmark, which makes it a raging concern for real-time applications. In [45], the authors design a hybrid histogram policy that is responsible for reducing the number of cold start requests with least amount of resource utilization. The policy is based on different rules that uses different keep-alive values according to the actual invocation frequency and pattern (time series analysis for unpredictable functions). Even though the designed policy works efficiently to eliminate large number of cold-start invocations, it is computationally expensive to operate. Similarly, freshen [22], a new primitive proposed to serverless runtimes allows the developers to pre-initialize functions depending upon their predictability. A potential integration of freshen with open source serverless architectures to reduce cold-start latency and improve function responsiveness is a pressing need that can be explored in this work.

7.5 Other Optimization Techniques

There are some other recent work [2, 6, 8, 12, 14, 23, 34, 35, 37] on optimization of latency in serverless computing platforms, which is complementary to our project. For example, OFC [35] uses the idle memory (because of overprovisioning) to serve as the in-memory cache per-node to reduce the latency of accessing backend datastore. Nightcore [23] schedules function chains locally on one worker node instead of going through the gateway again and again and also does OS optimizations to reduce IPC overhead. SEUSS [6] deploy functions from unikernel snapshots, bypassing expensive initialization steps. It reduces the memory footprint of snapshots by applying page-level sharing across the entire software stack that is required to run a function. USETL [14] also proposes that unikernels are a natural fit for execution contexts with these properties: they are minimal kernels packaged with a single application in a single address space, which makes them incredibly lightweight.

8 Conclusion and Future Work

With the increasing popularity of serverless architectures and services, the issue of cold-start delays and performance variation is a growing concern. However, no cloud provider allows application owners to specify performance SLOs, which hinders the adoption of serverless computing to latency-critical applications. In this project, we perform comprehensive characterization on a popular serverless computing platform OpenWhisk and we can deduce that the latencies are largely affected by the warm and cold start conditions. CPU-intensive application workloads such as base64 benefit largely with higher CPU-share without affecting the memory-limits. The profiling results with different FaaS application workloads and varying external parameters such as CPU-share, memory limit, and concurrency levels provide a deeper insight into the resource allocation in the serverless computing platform design.

Based on those insights and serverless platform characteristics, we further proposed and implemented a performance-aware resource allocation framework for multi-tenant FaaS workload. Given the user-defined SLOs, our approach provides performance guarantees with a profiling-based per-application resource allocation mechanism. We design an ML-based resource controller to draw insights on per-function resource allocation instead of doing plenty of repetitive profiling work for each function. Evaluation results show that our approach outperforms OpenWhisk by 1.5x to 4.1x in terms of the end-to-end latency. In the future, we plan to extend our framework to a chain or graph of functions instead of a single function for optimizing end-to-end application SLOs.

The source code for the project is publicly available at: https://github.com/James-QiuHaoran/openwhisk

References

[1] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI '20)* (2020), pp. 419–434.

[2] Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. SAND: Towards high-performance serverless computing. In *2018 Usenix Annual Technical Conference (USENIX ATC '18)* (2018), pp. 923–935.

[3] Serverless microservices - microservices on AWS. https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/serverlessmicroservices.html.

[4] Building serverless microservices in Azure - sample architecture. https://azure.microsoft.com/is-is/blog/building-serverless-microservices-in-azure-sample-architecture/.

[5] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[6] Cadden, J., Unger, T., Awad, Y., Dong, H., Krieger, O., and Appavoo, J. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–15.

[7] Caruana, R., and Niculescu-Mizil, A. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd International Conference on Machine Learning* (2006), pp. 161–168.

[8] Carver, B., Zhang, J., Wang, A., Anwar, A., Wu, P., and Cheng, Y. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (2020), pp. 1–15.

[9] Castro, P., Ishakian, V., Muthusamy, V., and Slominski, A. Serverless programming (function as a service). In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (2017), IEEE, pp. 2658–2659.

[10] Chen, S., Delimitrou, C., and Martínez, J. F. 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* (New York, NY, USA, 2019), ASPLOS '19, Association for Computing Machinery, p. 107–120.

[11] Cloud, G. App Engine application platform. https://cloud.google.com/appengine, 2020.

[12] Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q., and Chen, H. 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* (2020), pp. 467–481.

[13] Ellis, A. OpenFaas: Serverless functions, made simple. https://www.openfaas.com/.

[14] Fingler, H., Akshintala, A., and Rossbach, C. J. Usetl: Unikernels for serverless extract transform and load why should you settle for less? In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems* (2019), pp. 23–30.

[15] Open source, Kubernetes-native serverless framework. https://fission.io/.

[16] google. cadvisor. https://github.com/google/cadvisor, 2021.

[17] Grafana: The open observability platform.

[18] Hellerstein, J. M., Faleiro, J., Gonzalez, J. E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., and Wu, C. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651* (2018).

[19] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Serverless computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (Denver, CO, June 2016), USENIX Association.

[20] Hendrix, R. W. Google Cloud Functions. https://cloud.google.com/functions.

[21] Hendrix, R. W. Amazon Lambda. https://aws.amazon.com/lambda/, 1983.

[22] Hunhoff, E., Irshad, S., Thurimella, V., Tariq, A., and Rozner, E. Proactive serverless function resource management. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing* (New York, NY, USA, 2020), WoSC'20, Association for Computing Machinery, p. 61–66.

[23] Jia, Z., and Witchel, E. 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* (2021), pp. 152–166.

[24] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[25] Kaffes, K., Yadwadkar, N. J., and Kozyrakis, C. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 2019)* (2019), pp. 158–164.

[26] Kannan, R. S., Subramanian, L., Raju, A., Ahn, J., Mars, J., and Tang, L. GrandSLAm: Guaranteeing SLAs for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, Association for Computing Machinery.

[27] Kanso, A., and Youssef, A. Serverless: beyond the cloud. In *Proceedings of the 2nd International Workshop on Serverless Computing* (2017), pp. 6–10.

[28] Khatri, D., Khatri, S. K., and Mishra, D. Potential bottleneck and measuring performance of serverless computing: A literature study. In *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)* (2020), pp. 161–164.

[29] Knative. https://knative.dev/, 2021.

[30] Kubeless: The Kubernetes native serverless environment. https://kubeless.io/.

[31] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. Heracles: Improving resource efficiency at scale. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)* (2015), pp. 450–462.

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

[33] Architecture: Scalable commerce workloads using microservices. https://cloud.google.com/solutions/architecture/scaling-commerce-workloads-architecture.

[34] Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., and Sukhomlinov, V. Agile cold starts for scalable serverless. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (2019).

[35] Mvondo, D., Bacou, M., Nguetchouang, K., Ngale, L., Pouget, S., Kouam, J., Lachaize, R., Hwang, J., Wood, T., Hagimont, D., et al. OFC: an opportunistic caching system for FaaS platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems* (2021), pp. 228–244.

[36] Nuclio: Automate the data science pipeline with serverless functions. https://nuclio.io/, 2017.

[37] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 Usenix Annual Technical Conference (USENIX ATC '18)* (2018), pp. 57–70.

[38] Patel, T., and Tiwari, D. Clite: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers. In

2020 IEEE International Symposium on High Performance Computer Architecture (HPCA) (2020), IEEE, pp. 193–206.

[39] Pérez, A., Moltó, G., Caballer, M., and Calatrava, A. Serverless computing for container-based architectures. *Future Generation Computer Systems 83* (2018), 50–59.

[40] Qiu, H., Banerjee, S. S., Jha, S., Kalbarczyk, Z. T., and Iyer, R. K. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 805–825.

[41] Rajan, R. A. P. Serverless architecture - a revolution in cloud computing. In *2018 Tenth International Conference on Advanced Computing (ICoAC)* (2018), pp. 88–93.

[42] Shafiei, H., Khonsari, A., and Mousavi, P. Serverless computing: A survey of opportunities, challenges and applications, 2019.

[43] Shafiei, H., Khonsari, A., and Mousavi, P. Serverless computing: A survey of opportunities, challenges and applications. *arXiv preprint arXiv:1911.01296* (2019).

[44] Shahrad, M. FaaSProfiler: A tool for testing and profiling Function-as-a-Service platforms. http://parallel.princeton.edu/FaaSProfiler.html, 2019.

[45] Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 205–218.

[46] Silva, P., Fireman, D., and Pereira, T. E. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference* (2020), pp. 1–13.

[47] Singhvi, A., Houck, K., Balasubramanian, A., Shaikh, M. D., Venkataraman, S., and Akella, A. Archipelago: A scalable low-latency serverless platform, 2019.

[48] Suresh, A., and Gandhi, A. FnSched: An efficient scheduler for serverless functions. In *Proceedings of the 5th International Workshop on Serverless Computing* (New York, NY, USA, 2019), WOSC '19, Association for Computing Machinery, p. 19–24.

[49] Tariq, A., Pahl, A., Nimmagadda, S., Rozner, E., and Lanka, S. Sequoia: Enabling Quality-of-Service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (New York, NY, USA, 2020), SoCC '20, Association for Computing Machinery, p. 311–327.

[50] Vahidinia, P., Farahani, B., and Aliee, F. S. Cold start in serverless computing: Current trends and mitigation strategies. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)* (2020), pp. 1–7.

[51] Open source serverless cloud platform. https://openwhisk.apache.org/.

[52] Wu, M., Mi, Z., and Xia, Y. A survey on serverless computing and its implications for jointcloud computing. In *2020 IEEE International Conference on Joint Cloud Computing* (2020), pp. 94–101.

[53] Yanqi, Z., Weizhe, H., Zhuangzhuang, Z., G. Edward, S., and Christina, D. Sinan: ML-based & QoS-aware resource management for cloud microservices. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), ASPLOS '21.

[54] Yu, T., Liu, Q., Du, D., Xia, Y., Zang, B., Lu, Z., Yang, P., Qin, C., and Chen, H. Characterizing serverless platforms with serverlessbench. In *Proceedings of the ACM Symposium on Cloud Computing* (2020), SoCC '20, Association for Computing Machinery.

A Project Source Code

The source code for the project is publicly available at: https://github.com/James-QiuHaoran/openwhisk

B Detailed Discussion

It is worth mentioning that the cold-start is controlled by the docker daemon, and docker daemon uses docker run to create a new runtime container. Initialization process executes inside the created container, so it is influenced by the CPU-share. In the OpenWhisk design, the Node.js runtime container works as a server, which has two end points, `/init` and `/run`. In the initialization phase, the invoker will send a binary of the user function as a message to the runtime container's `/init` end-point. The runtime container will unzip the user function, store it locally and set up a handler for this user function. If there is a request, the invoker will send the parameter as a message to the runtime container's `/run` end-point. The container will execute it and return the results back to the controller. In a running invoker, there are two kinds of container running: invoker container and runtime containers. Invoker container uses the default CPU-share: 1024. Runtime containers use the CPU-share that we assign to them ranging from 128 - 1024. When we increase the runtime containers' CPU-share, runtime containers will "steal" more CPU time from the invoker. More CPU time means better performance, so the average initialization time and execution time decrease when we set a higher CPU-share for runtime containers. Moreoever, it is worth mentioning that changing CPU-share has different effects on the multi-node cluster and single-node cluster. On the multi-node cluster, it impacts the cold-start time, initialization time and execution time. This is because, the larger the memory limit, the less concurrent containers will run in each invoker, and thus, the more CPU shares each container can use. This leads to faster cold-start, initialization time, execution time on the cluster. On the other hand, on a single node, the controller, invoker and runtime container are running concurrently (say with a CPU share of 1024, 1024 and 128 each). Thus, when these processes run concurrently, the runtime container is only bound to get 128/(1024+1024+128) share of the CPU, and hence the CPU-share impacts the latencies for single-node system.

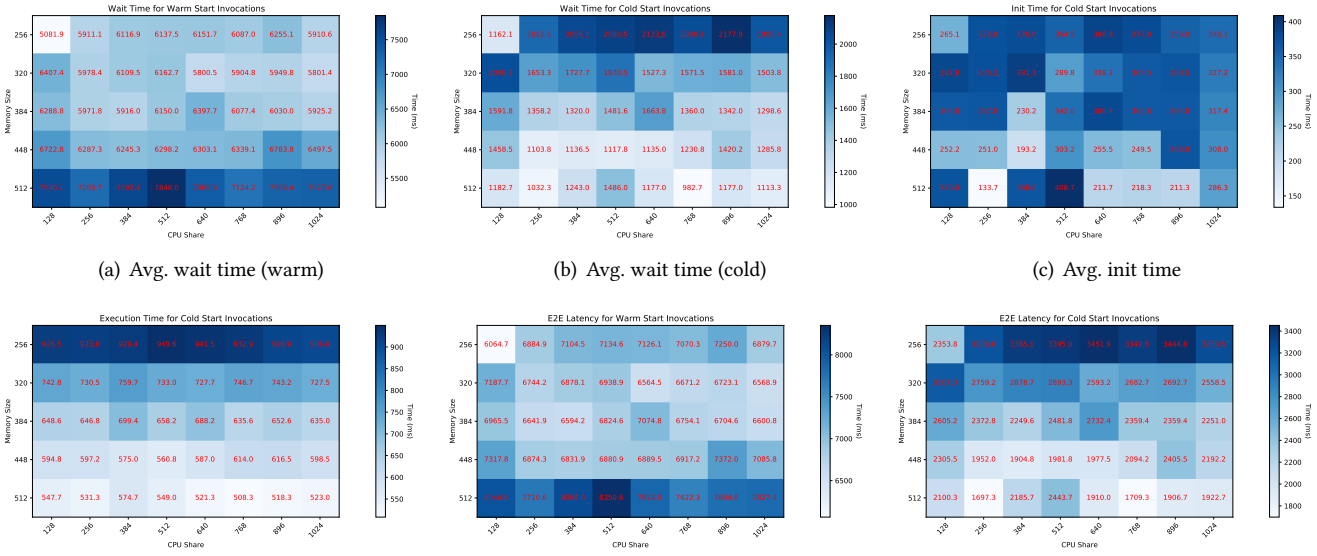
C Additional Benchmarking Results

We present additional benchmarking results in the appendix section, due to the page limits. It includes the profiling results for base64 benchmark at rate 20, 30, 40, 50 in the single-node OpenWhisk cluster (refer to Figure 10, 11, 12, and 13). It also includes the profiling result comparison among different benchmarks (i.e., the base64, json and primes benchmark) on the multi-node (1-master and 9-invoker) OpenWhisk cluster (refer to Figure 14, 15, and 16).

We also show an example of the result of our online monitoring and visualization system in Figure 17. We demonstrate the CPU and memory usage traces of selected containers and processes including container daemon, docker daemon,

invoker container and runtime containers during a test[6]. From Figure 17-(a)&(b), we can clearly find the peak CPU usage and peak memory usage of each container. We can also find the relative CPU usage among different containers. In Figure 17-(c), we show the latency breakdowns of function invocations during a run-time container's life time. we can find there are in total three function invocations assigned to this runtime container. X-axis is the timeline and y-axis indicates the event index which can be mapped back to the event name with the list in Sec. 4.2. The blue line represents the latency breakdown of the very first invocation assigned to this container, so it will wait for the finish of cold start. We can clearly find the duration from 4^{th} to 5^{th} events which is the cold start time dominates the whole end-to-end latency of this function invocation. The green line represents the latency breakdown of a warm start invocation. We can find it still suffers from long latency because it arrives right after the first invocation and is queued until the previous invocation (the blue line) finishes the execution. The orange line represents the latency breakdown of a warm start which is assigned to the container when the cold start has finished. We can find the total latency of this invocation is short because it does not need to wait for cold start or be queued due to cold start. In summary, we can use the visualization to confirm our assumption that the cold start is the root cause of long end-to-end latency.

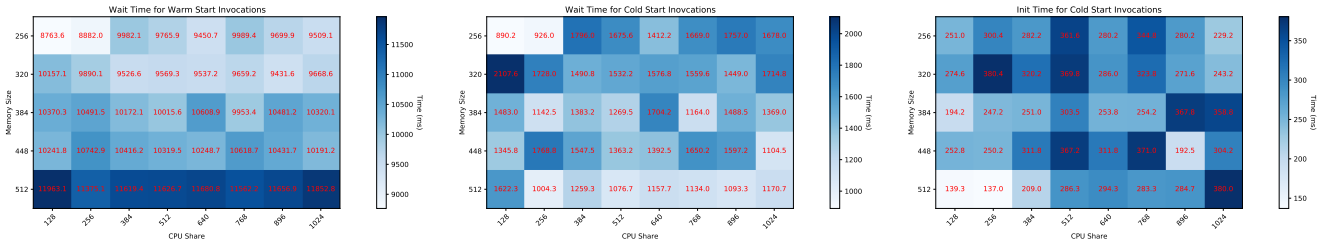
[6]The CPU-share is set to 512; The memory limitation varies from 320 to 512 with step size 64; We use primes as the workload; The invocation rate is 10 invocations per second; The invocation distribution is uniform distribution; The invocation duration is 5 seconds



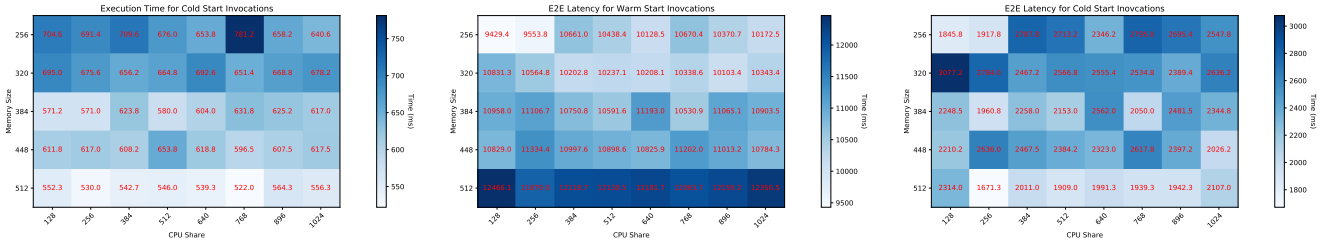
(a) Avg. wait time (warm) (b) Avg. wait time (cold) (c) Avg. init time

(d) Avg. execution time (e) Avg. e2e latency (warm) (f) Avg. e2e latency (cold)

Figure 10. Profiling of base64 benchmark on a single-node cluster with invocation rate=20.

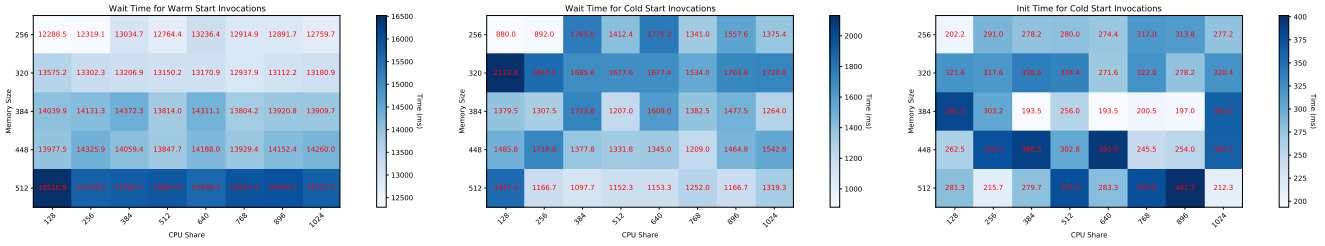


(a) Avg. wait time (warm) (b) Avg. wait time (cold) (c) Avg. init time

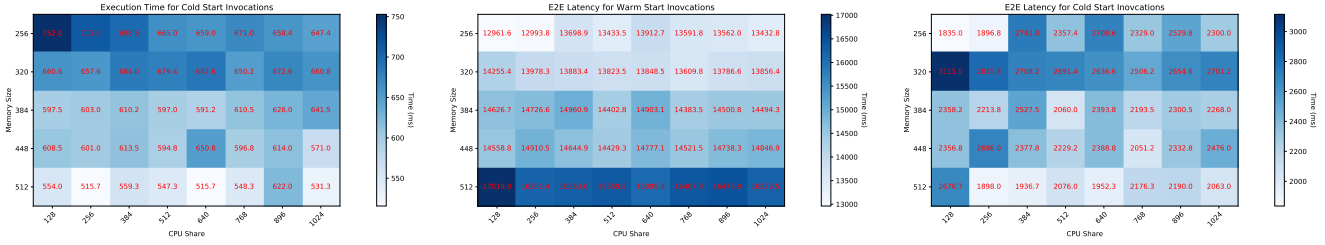


(d) Avg. execution time (e) Avg. e2e latency (warm) (f) Avg. e2e latency (cold)

Figure 11. Profiling of base64 benchmark on a single-node cluster with invocation rate=30.

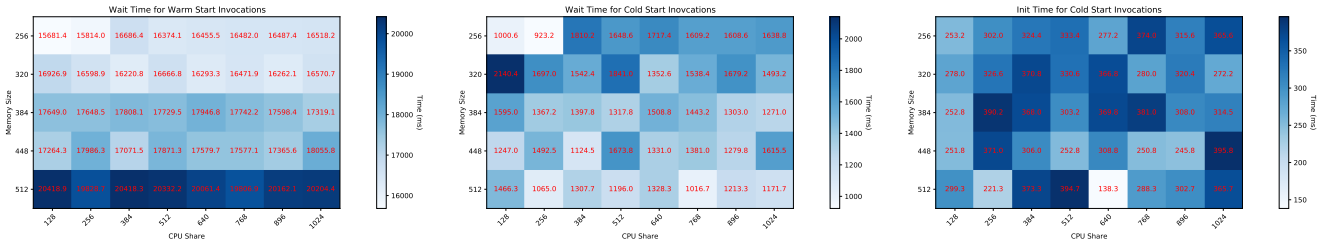


(a) Avg. wait time (warm) (b) Avg. wait time (cold) (c) Avg. init time

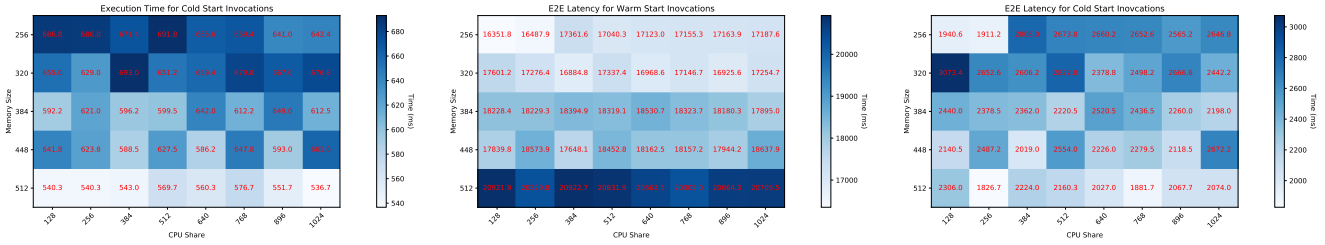


(d) Avg. execution time (e) Avg. e2e latency (warm) (f) Avg. e2e latency (cold)

Figure 12. Profiling of base64 benchmark on a single-node cluster with rate=40.

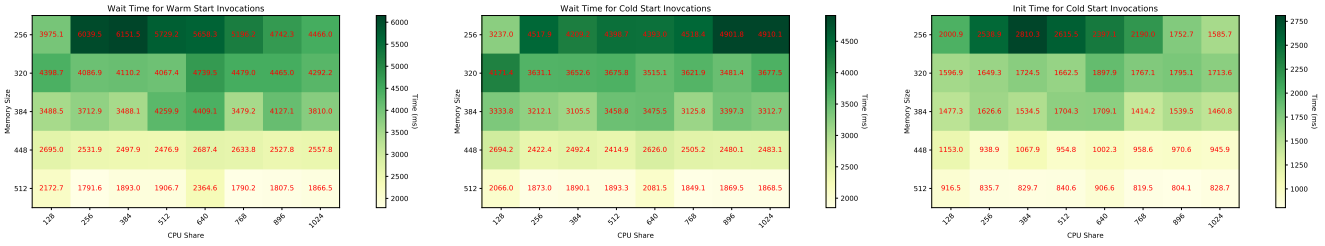


(a) Avg. wait time (warm) (b) Avg. wait time (cold) (c) Avg. init time

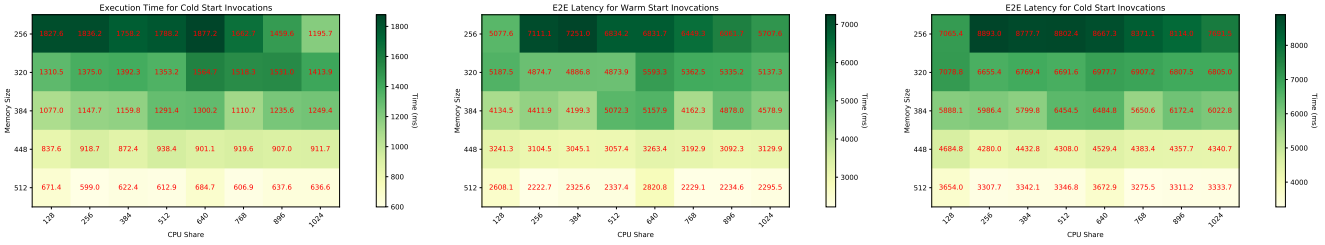


(d) Avg. execution time (e) Avg. e2e latency (warm) (f) Avg. e2e latency (cold)

Figure 13. Profiling of base64 benchmark on a single-node cluster with invocation rate=50.

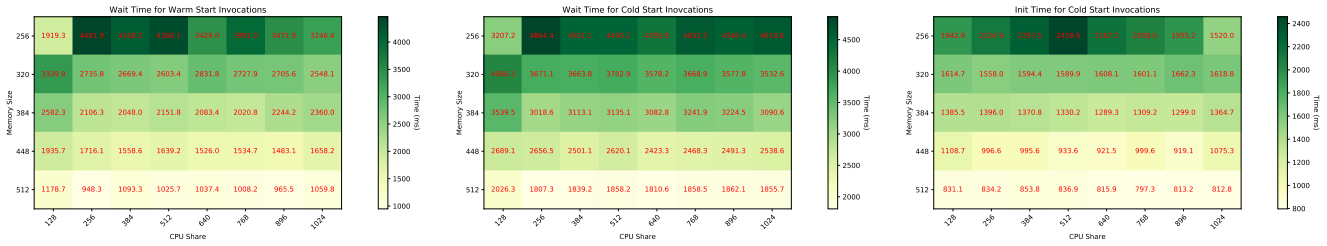


(a) Avg. wait time (warm) (b) Avg. wait time (cold) (c) Avg. init time

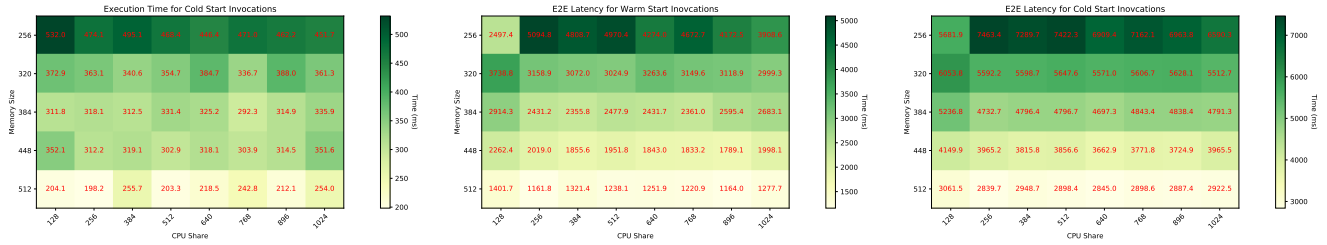


(d) Avg. execution time (e) Avg. e2e latency (warm) (f) Avg. e2e latency (cold)

Figure 14. Profiling of base64 benchmark on multi-node cluster with invocation rate=30.

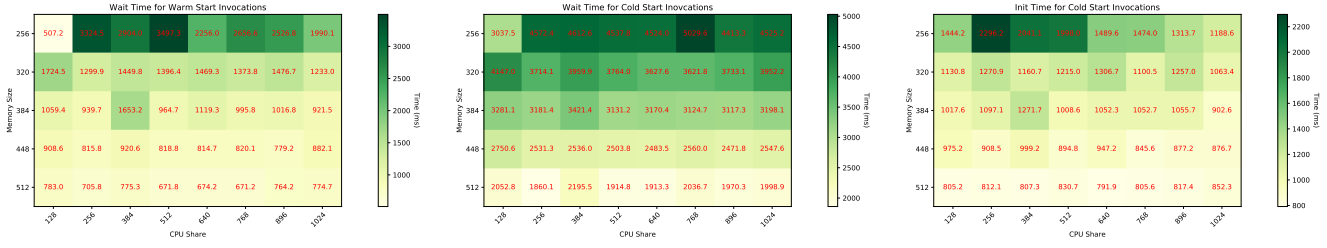


(a) Avg. wait time (warm) (b) Avg. wait time (cold) (c) Avg. init time

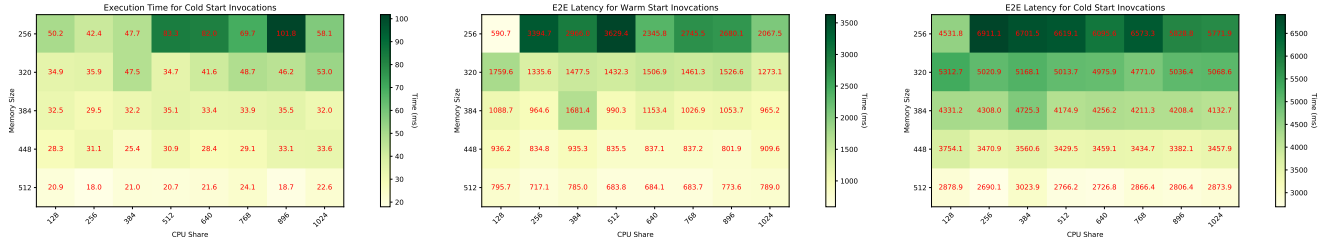


(d) Avg. execution time (e) Avg. e2e latency (warm) (f) Avg. e2e latency (cold)

Figure 15. Profiling of primes benchmark on multi-node cluster with invocation rate=30.

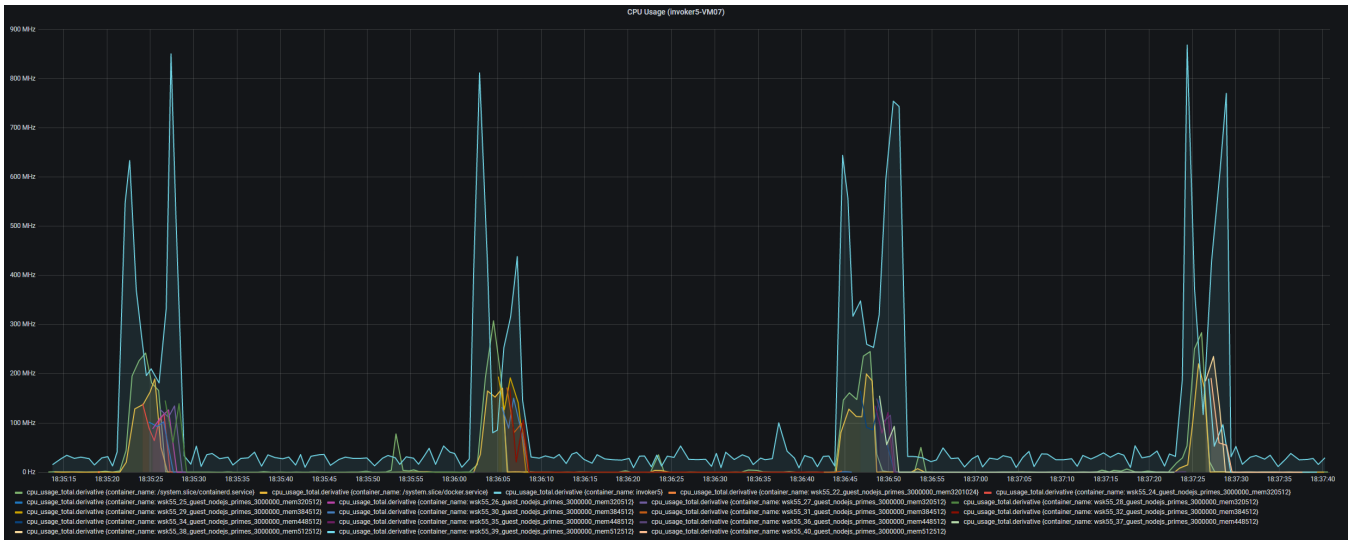


(a) Avg. wait time (warm) (b) Avg. wait time (cold) (c) Avg. init time

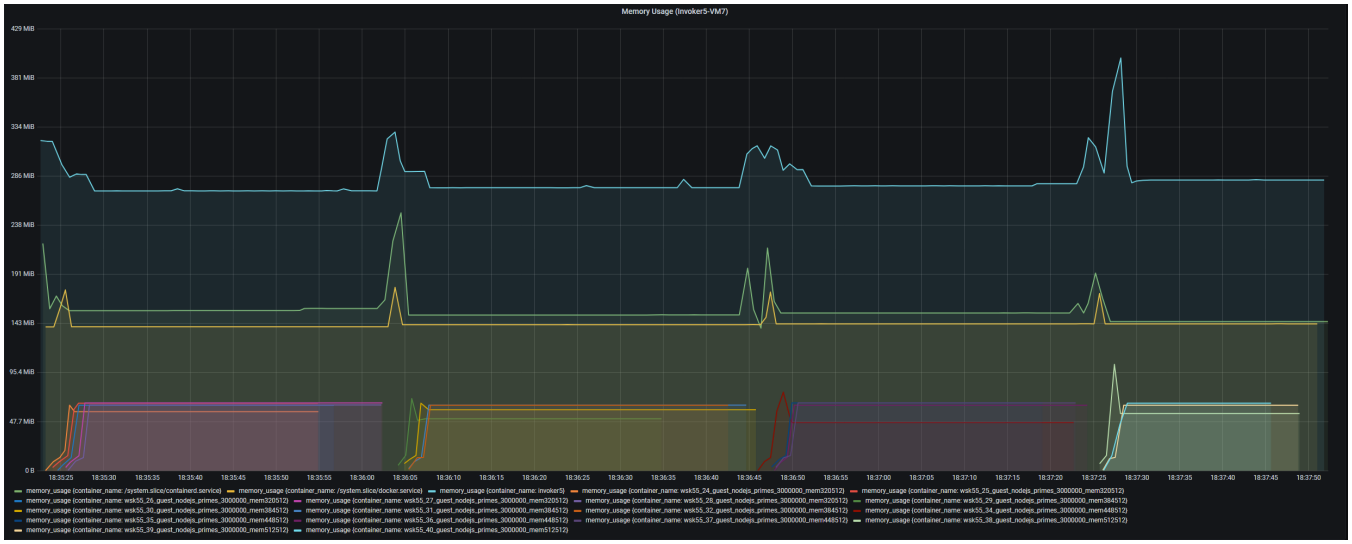


(d) Avg. execution time (e) Avg. e2e latency (warm) (f) Avg. e2e latency (cold)

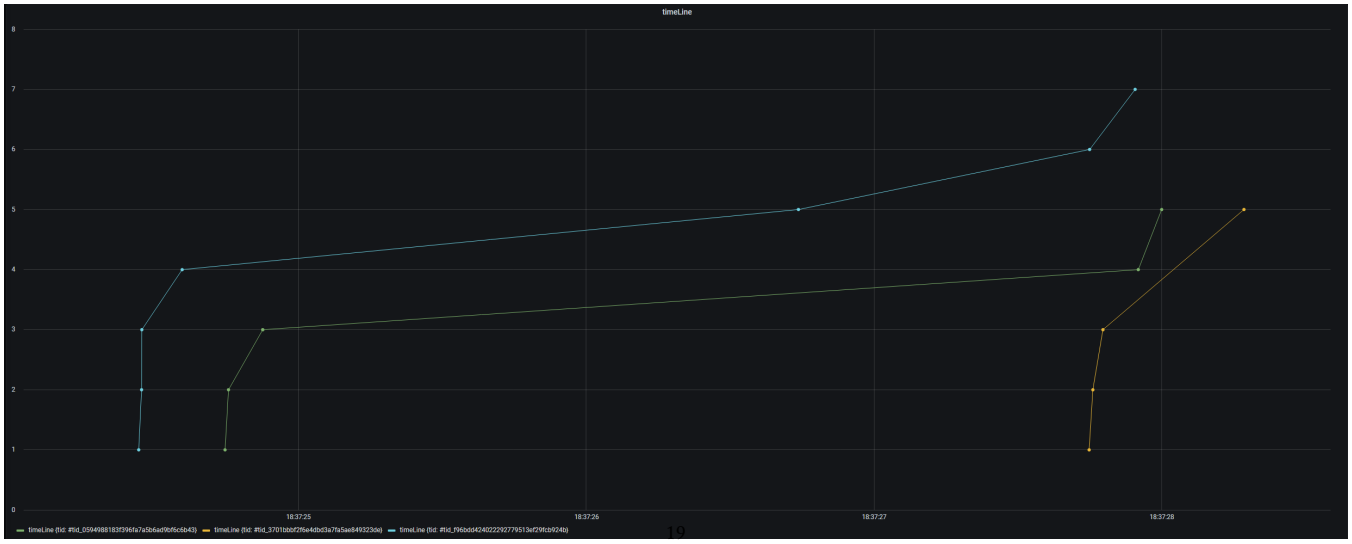
Figure 16. Profiling of json benchmark on multi-node cluster with invocation rate=30.



(a) CPU utilization of different containers



(b) Memory utilization of different containers



(c) Latency breakdowns of function invocations in one container

Figure 17. Examples (screenshots) of the CPU usage, Memory usage, latency breakdown visualization.