Supporting Multi-Provider Serverless Computing on the Edge

Austin Aske
School of Engineering and Computer Science
Washington State University
Vancouver, WA, USA
austin.aske@wsu.edu

Xinghui Zhao
School of Engineering and Computer Science
Washington State University
Vancouver, WA, USA
x.zhao@wsu.edu

ABSTRACT

Serverless computing has recently emerged as a new execution model for cloud computing, in which service providers offer compute runtimes, also known as Function-as-a-Service (FaaS) platforms, allowing users to develop, execute and manage application functionalities. Following the rapid adoption of FaaS technologies and the introduction of numerous self hosted FaaS systems, the need for real time monitoring and scheduling of functions in an ecosystem of providers has become critical. In this paper, we present MPSC, a framework for supporting Multi-Provider Serverless Computing. MPSC monitors the performance of serverless providers in real time, and schedules applications across these providers. In addition, MPSC also provides APIs for users to define their own scheduling algorithms. When compared to scheduling on a single cloud resource MPSC provides a 4X speedup across multiple providers in a volatile edge computing environment.

CCS CONCEPTS

• **General and reference → Metrics**; **Performance**; • **Information systems → Computing platforms**; • **Computing methodologies → Parallel computing methodologies**; • **Computer systems organization → Multicore architectures**;

KEYWORDS

Serverless Computing; Function-as-a-Service; Edge Computing; Scheduling; Performance

ACM Reference Format:
Austin Aske and Xinghui Zhao. 2018. Supporting Multi-Provider Serverless Computing on the Edge. In *ICPP '18 Comp: 47th International Conference on Parallel Processing Companion, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3229710.3229742

1 INTRODUCTION

Over the last two decades, cloud computing has revolutionized the IT industry by providing users the ability to provision on-demand resources in near real-time, and eliminate capital expenses involved in purchasing hardware [4, 11]. The virtualization of hardware infrastructure allowed cloud computing to deliver elasticity and the illusion of infinite capacity for a wide spectrum of developers [4]. Enabled by container technologies, serverless computing further perfects the subdivision of computation resources, elasticity, and abstraction of underlining hardware pioneered by containers [5]. Serverless technologies, also known as Function-as-a-service, have proven to be more scalable, elastic, developer-friendly, and cost-effective than previous cloud architectures [9, 10, 19]. FaaS technology allows developers to publish microservices, referred to as functions or actions, using a variety of shared runtime environments, without needing to manage a single server. Each function represents a small stateless snippet of code which can be triggered through events. Events most commonly come in the form of HTTP request, database changes, or from another function. Due to the nature of stateless functions, they are embarrassingly parallelizable and can be horizontally scaled, utilizing the full power of the hardware resources.

Serverless technologies shift all server management and execution of functions to the service provider, allowing efficient automatic scaling and fast market development [17]. Serverless computing enables developers to focus on their code, relieving them from server and deployment details as seen in virtual machines and containers. The increased abstraction level provided by FaaS minimizes developer boiler plate code and configuration/management of scaling systems. Due to the decoupling of functions through HTTP interfaces, serverless architecture reinforces modular development. Modular development allows clear separation between components and isolation of functionality, both of which are widely considered best practices in software design, increasing software quality, test-ability, and productivity.

Serverless technology becomes more and more popular with new providers emerging rapidly. The introduction of open-source host agnostic offerings are shown in Table 1. With the emergence of new providers, clients are gaining options of where to run their code. Each provider has unique features and offers a multitude of companion services, i.e., monitoring, logging, and databases. Additionally, each provider has unique performance characteristics and are susceptible to outages. Performance characteristics and service outages can be caused by a number of reasons, including hosting location or preparatory implementations of FaaS infrastructure [4, 7].

One might imagine an ecosystem where developers write code once and deploy on any FaaS provider. In fact, quite the opposite is the case, providers require unique function signatures, naming conventions, and event compatibility, causing a provider lock-in effect. Luckily, two open source frameworks have jumped in to help with interconnecting serverless providers, Serverless Framework, and Event Gateway. Serverless Framework simplifies and standardizes function development allowing deployment on all of the major FaaS

ICPP '18 Comp, August 13–16, 2018, Eugene, OR, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6523-9/18/08…$15.00
https://doi.org/10.1145/3229710.3229742

Provider	Runtime	Classification
Amazon	Lambda	commercial
Microsoft	Azure Functions	commercial
Google	Cloud Functions	commercial/beta
IBM	Apache OpenWhisk	open-source
Host Agnostic	Kubeless	open-source
	Fission	open-source
	OpenFaas	open-source
	Nuclio	open-source
	FnProject	open-source

Table 1: Major Providers and Runtimes

providers with minimal code changes. Meanwhile, Event Gateway standardizes events allowing for cross provider communication.

Even with the advent of the Serverless Framework and Event Gateway laying the foundation for cross provider utilization, to date there is no infrastructure for scheduling functions in an ecosystem of providers with unique performance characteristics. To assist in choosing the "right" provider, we have developed MPSC framework, a serverless monitoring and scheduling system. MPSC first monitors the performance of serverless providers in real time. Second, based on the performance results and a user-defined schedule MPSC can recommend the optimal location to run the user's code. MPSC allows users to implement scheduling algorithms based on a provider's moving average execution time, resource affinity, and cost of provider. Utilizing MPSC's adaptive scheduling has shown to increase performance and QoS over conveniently single provider utilization.

The remainder of the paper is organized as follows. Section 2 presents related work in FaaS technologies, including a background of FaaS, and research on providing quality of service for FaaS platforms. In Section 3, we describe our design and implementation details of the MPSC framework. The evaluation of the MPSC is presented in Section 4, including experimental design and results. Finally, Section 5 concludes the paper and presents future work for this research.

2 RELATED WORK

As shown in Figure 1, there are many degrees of virtualization. The dedicated machine architecture has no virtualization which results in strong dependencies on the underlining hardware, OS, with no subdivision of physical resources. Virtual machine architecture decouples the hardware dependencies with the utilization of a hypervisor, allowing increased flexibility in the underlining hardware and added migration capabilities. In container virtualization, the host OS layer is virtualized, making for smaller images and better utilization of resources. Finally, the serverless architecture is built on top of container virtualization, where multiple apps/functions in the FaaS model can make use of the same runtime.

With the ability to share all aspects of the software stack, the serverless systems are able to quickly provision and auto-scale resources based on demands. Provisioning results show cold starts on AWS lambda, IBM Bluemix, and Google Cloud Functions executing in the sub 2 second range. However, when utilizing warm

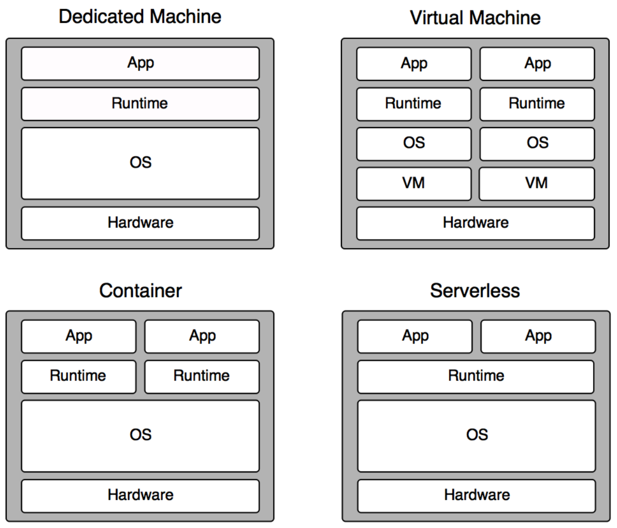

Figure 1: Virtualization Architectures

functions, or functions ran on already provisioned containers, execution times average under 1 second [15]. Serverless providers handle auto-scaling of concurrent functions based on the volume of requests with many providers having user defined limits between 1000-3000 [1–3].

A significant amount of work has been done in exploring the potentials of the FaaS technologies. For instance, Spillner et al. have evaluated the usability and work-flow of developing serverless applications in four different scientific computing domains [17], including mathematics, computer graphics, cryptography, and meteorology. In addition to demonstrating the feasibility of serverless computing across a diverse set of scientific computing domains, Spillner et al. also present FaaSification, a tool which automates the subdivision of monolithic applications based on function features.

In [13], Jonas et al. state that despite many years of availability, scale and elasticity from traditional cloud computing systems are too difficult and that the serverless model could be used instead. Arguing that traditional data processing systems like Apache Hadoop or Spark require decisions on instance type, cluster size, and pricing model before hand. Additionally, their work presents a prototype framework named PyWren which implements a MapReduce like workflow using AWS Lambda. PyWren simplifies the setup and usage of data processing systems utilizing serverless technology and reduces the developer overhead involved in publishing functions. By cashing a minimized Python Anaconda runtime and user functions in attached AWS S3 storage, PyWren is able to use a single published function for most data processing work-flows. As a result, PyWren lowers complexity of setting up and using cloud resources while increasing parallelism over traditional data processing systems.

In parallel to the advancements in serverless computing, edge computing is also evolving into a promising architecture for providing a superior quality of service (QoS) and enabling new classes of applications. Edge Computing pushes computation resources to the edge of the network creating an additional layer of resources between the end user and the cloud. With the addition of an intermediate layer, edge computing can facilitate applications with strict latency, power, security, and/or availability requirements [6, 12, 16]. Because of FaaS's unique characteristics, namely, fast provisioning, statelessness, and minimal overhead, it remains one of the top paradigms in consideration for extending the two layer cloud architecture to a multi-layer edge computing architecture, making it an emerging area of research.

Much of the research in edge computing to this point has been focused around optimizing virtual machines (VM) or containers virtualizaiton methods[8, 14, 20]. Edge computing adds additional computational resources or cloudlets in close proximity to the client devices, but each node is limited in computational capacity. The problem of limited resources at the edge of the network creates a difficult distributed scheduling problem which has been studied extensively in [18]. With respect to scheduling on the edge and cloud using FaaS paradigm, scheduling is simplified slightly due to the fact that requests are sent and completed at some non-deterministic time. From the client's perspective, there is no knowledge of server workload, memory consumption, or resource competition making for simplified scheduling algorithms. Note that both VM's and containers incur additional overhead due to runtime and operating system image sizes. Utilizing FaaS architectures helps address this problem.

Baresi et al. [5] evaluate the performance benefits associated with utilizing FaaS at the edge in an augmented reality application. Their research shows that running OpenWhisk deployments locally reduces latency and increases throughput by as much as 80% over AWS Lambda. Baresi et al. outline much of the foundational work for utilizing FaaS in the context of edge computing, which in many respects inspires our work.

3 MPSC FRAMEWORK

We have designed and implemented MPSC, a framework for supporting Multi-Provider Serverless Computing. MPSC aims for providing a platform for users to schedule their applications across multiple serverless providers based on a customizable scheduling algorithm.

3.1 System Design

MPSC architecture consists of loosely coupled microservices, utilizing REST style communication and multiple extensible plug-able components. Designed for maximum flexibility, MPSC supports creation of customized scheduling logic and executors for usage with new FaaS providers. Additionally, by allowing communication through standard HTTP, the application can be easily containerized and deployed in a multitude of environments. MPSC architecture is shown in Figure 2 and further details of the components described below.

- **Monitoring Controller** is the key component of MPSC framework responsible for coordinating model objects (Configuration File, Metrics Database, and User Defined Schedules) data with the Function Executors and REST API.
- **REST API** provides a series of HTTP end-points for user and their applications to communicate with the MPSC Framework. As shown in Table 2, the four end-points included in the current version of MPSC are: *Metrics*, *Test*, *AddSchedule*, *Schedule*.
- **Function Executors** is a set of extensible interconnects for invoking and measuring performance of functions on FaaS providers. Currently, there are two Function Executors, one for working with functions on AWS Lambda and another for functions on OpenWhisk.
- **Configuration File** holds user specific configuration properties for each provider.
- **Metrics Database** stores benchmark results for each provider specified in the Configuration File. Each entry in the database consists of the name of the function invoked, the round-trip time, and error status.
- **User Defined Schedules** retains Python schedules uploaded through the REST API. Each submitted schedule must be a subclass of *ComputeBaseClass* and implement the *schedule* method.

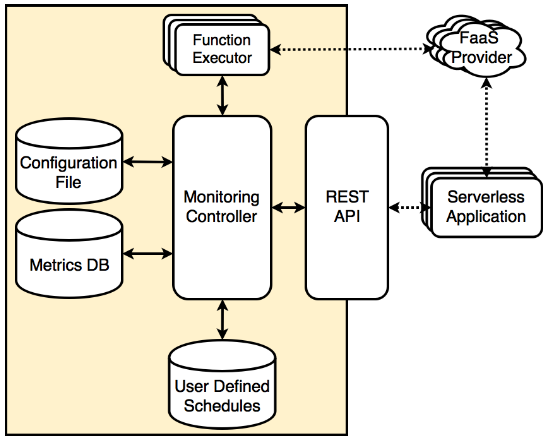

Figure 2: MPSC System Architecture

API	Arguments	Return Value
Metrics	none	Dictionary of metrics for each provider
Test	ProviderID	None
Add Schedule	ScheduleID, Schedule	None
Schedule	ScheduleID	ProviderID of the best fit provider

Table 2: MPSC REST API

3.2 Workflow

MPSC framework is a serverless monitoring and scheduling system created to achieve a higher QoS for serverless applications. Shown in Figure 3 is the basic workflow of MPSC framework, where a user uploads a schedule and querying the system to determine which provider to use for their serverless application. As a result of this workflow, the serverless application runs on the best fit provider determined by their scheduling algorithm.

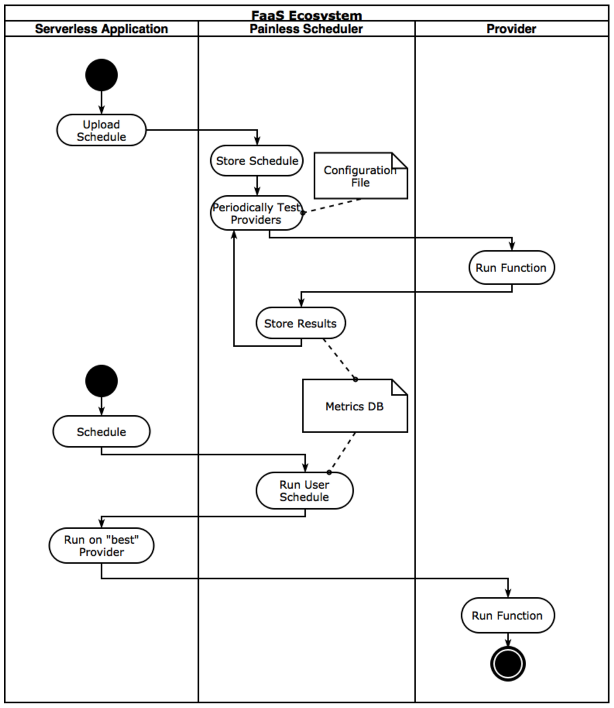

Figure 3: MPSC Workflow

There are two prerequisites for running the work flow shown in Figure 3, completing the configuration file and writing a user defined schedule.

```
compute:
    providers:
        - openwhisk:
            apihost: "openwhisk.ng.bluemix.net"
            namespace: "guest@example.com"
            cost: 0.01
```

Figure 4: Example Configuration File

The configuration file is loaded at the launch of the MPSC system and includes user specific information pertaining to each FaaS provider, as shown in Figure 4. Because the configuration file is only loaded at the launch of the system, it is important to note that any changes to the file will not be recognized until a restart is preformed. Within the YAML formatted file, users must specify the following properties for each provider they wish to monitor/schedule.

- **Provider Type** - The provider type is denoted as a key under the *compute* : *providers* dictionary. Currently, supporting the values of AWS or OpenWhisk.
- **Apihost** - The apihost is a string identifying the region of the provider to be used.
- **Namespace** - Namespace or user domain to be used for that specific provider.
- **Cost** - Cost per invocation in USD.

3.3 User Defined Schedules

MPSC provides users the flexibility of defining their own scheduling algorithm. User defined schedules are implemented by subclassing the *ComputeBaseClass* found in the scheduler package. In the subclass implementation, it is required that the *schedule* method be implemented and return the name of the chosen provider using the provider's *name* method. An example schedule is shown in Figure 5. Furthermore, it is important to note that there are currently three metrics available to the user when implementing a custom schedule. The three metrics included with each provider are average round trip time, the number of request resulting in errors, and the cost per invocation. By utilizing a plugin interface for schedules, users can customize and tailor MPSC for their unique serverless requirements. For example, a provider with a cost less then $0.01/invocation with an average round trip time less then 20 milliseconds.

```
#!/usr/bin/env python3
import scheduler

class LowLatency(scheduler.ComputeBaseClass):
    def schedule(self, providers):
        providers.sort(key=lambda provider: ↩
                                    provider.avg↩
                                    ())
        return providers[0].name()
```

Figure 5: Example of a user defined schedule, which utilizes the provider with the lowest average round-trip time.

4 EVALUATION

We have carried out two sets of experiments to evaluate the performance of MPSC for scheduling applications across a number of service providers. Specifically, the performance of MPSC is evaluated under different workload scenarios, including a scenario with no background workload, and one with unknown background workload.

4.1 Providers

In our experiments, we used three providers, two cloud providers and one local provider. The two cloud providers consisted of AWS Lambda and IBM Bluemix. The AWS Lambda functions were hosted in the US-WEST-2 region located in Oregon. As for the IBM Bluemix, it was hosted in the US-SOUTH region located in Texas. The cloud provider's regions were chosen because they are the closest region offered from each provider to the Washington State University Vancouver campus where the experiments were conducted.

The local provider consisted of an Apache OpenWhisk instance running on the local machine. OpenWhisk was ran in local deployment mode, using the standard Ansible setup. For performance optimization, two changes were made when configuring OpenWhisk. The first change was increasing the limits shown in Table 3. Secondly, the local OpenWhisk implementation was changed to use Python 3 pre-warmed containers instead of the default NodeJS 6 containers. Because all our functions were implemented in Python 3, it was a substantial performance increase to have Python 3 environments ready to use or pre-warmed.

Limit	Default	Used
invocationsPerMinute	60	6000
concurrentInvocations	30	3000
firesPerMinute	60	6000

Table 3: Group variables changed in local OpenWhisk deployment

	AWS	Bluemix	Localhost	User Schedule
RT (s)	57.26446	26.13867	4.72654	4.72654
Var	0.00184	0.00413	0.00007	0.00007
Std	0.04292	0.06430	0.00824	0.00824

Table 4: Experimental results for low latency scheduling

The hardware used for the local OpenWhisk provider consisted of a 2015 MacBook Pro with a 2.5 GHz Intel Core i7 and 16 GB of ram. It is important to note that OpenWhisk is made up of containerized components, which are run as Docker containers on top of Docker for Mac. Within Docker for Mac preferences it is allocated 4 CPU cores and 2 GB of memory, making for a resource constrained server.

4.2 Low Latency Scheduling

The low latency scheduling test is an example implementation of a user defined schedule where the user aims to minimize round trip time. The example algorithm used is demonstrated in Figure 5, and in summary picks the provider with the lowest sum of the last 5 function calls. After running MPSC framework for approximately 400 seconds, testing each provider every 5 seconds, it is clear that the local OpenWhisk provider offers superior performance.

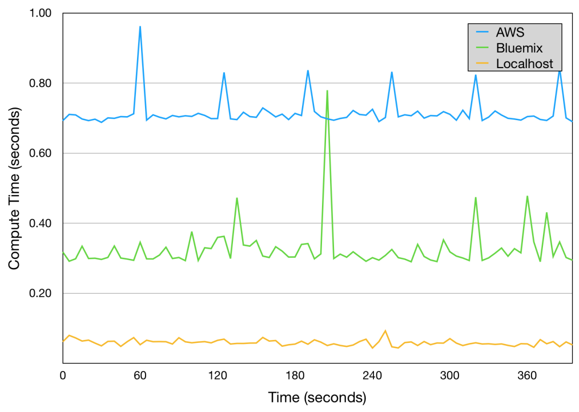

Figure 6: The round trip times of each provider running Miller-Rabin primality test on one millionth prime (15,485,863).

Analyzing the results from the low latency scheduling we see that the local OpenWhisk provider not only offers lower round-trip times but also far less variance and standard deviation in round-trip times. Table 4 compares the results from the experiment looking at the sum of round-trip times (RT), variance (Var), and standard deviation (Std) for each provider from the low latency test. From the data, we conclude that resources located at the edge of the network and closer to the user can provide an overall higher QoS. This result is consistent with findings from other related works [5, 10, 14] but the question still remains how the edge provider performs with additional traffic.

4.3 Low Latency Scheduling with Background Workload

The second set of experiments were carried out to evaluate the performance of MPSC under unknown workload. Specifically, we ran the same experiments as described in Section 4.2, but with added background workload. The introduction of simulated traffic on the local provider adds additional stress on the already resource constrained edge provider. The added traffic or noise was created by asynchronously sending 1000 function invocations to the OpenWhisk server while MPSC is running. This test was performed for approximately 130 seconds while taking measurements every 5 seconds.

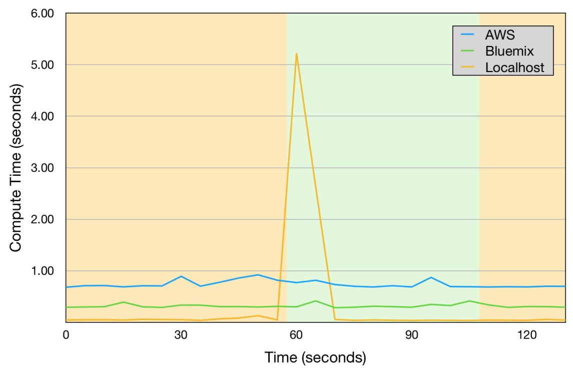

Figure 7: The round trip times of each provider with the introduction of 1000 function invocation requests sent to the local provider. Background color represents the best provider based on the user defined schedule.

As we can see from Figure 7, the background color corresponds to the chosen provider based on the provided schedule. MPSC adapts to the performance degradation of the local provider and resorts to utilizing the Bluemix provider temporarily. In this test, the scheduling algorithm used was the Low Latency schedule from Figure 5. Based on this schedule and the frequency of testing providers, MPSC takes 5 seconds to switch providers. Additionally, MPSC continues usage of Bluemix for approximately 30 seconds after the traffic spike accrued due to the moving average of the local provider being above that of Bluemix.

Further analysis was carried out on the experimental results of the low latency schedule with background workload, shown in Figure 8. Each provider's total execution time is displayed as a baseline and compared to the low latency schedule. In addition, Figure 8 compares to the optimal schedule, which is the sum execution time of the fastest provider at each time step. Comparing these results, we see 468% decrease in latency using MPSC vs AWS Lambda

	AWS	Bluemix	Localhost	User Schedule
RT (s)	20.03395	8.61793	9.15323	4.26473
Var	0.00527	0.00131	1.19856	0.01927
Std	0.07263	0.03614	1.09479	0.13881

Table 5: Experimental results for low latency scheduling with background workload

and over 200% decrease vs IBM Bluemix and the local OpenWhisk provider. For comparison, we also show the variance and standard deviation of the user schedule vs the other providers, which actually is higher than the two cloud providers. We argue the poor variance and standard deviation performance is a result of the user define algorithm used, and if we wanted to optimize for these metrics in addition to round-trip MPSC framework could be achieved too. In conclusion, our results show that MPSC framework can provide superior QoS based on application demands, independent of resource congestion.

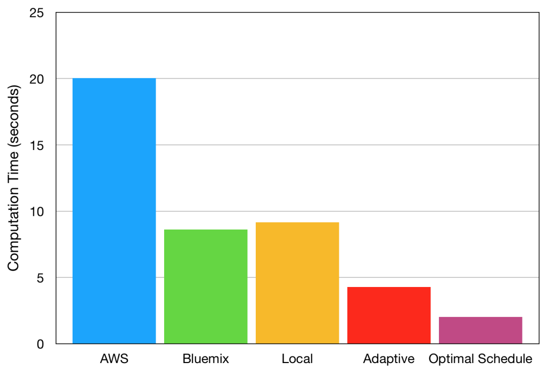

Figure 8: Total round trip times for each provider compared with our adaptive schedule and the optimal schedule.

5 CONCLUSION

Serverless technologies simplify microservice development while removing server administrative tasks. Consequently, developers of serverless applications are left with little control over the performance of their code. In response to this problem our proposed MPSC framework provides superior user defined QoS metrics in an ecosystem of FaaS providers. Results showing a 200% faster round trip execution time compared to execution on any one of the cloud providers and increased system agility when using constrained edge computing resources. As a result, we conclude that the utilization of multiple FaaS providers including edge deployments can substantially improve the QoS for serverless applications. For future work, we would like to integrate MPSC as a plugin for the Serverless framework to increase supported providers through a unified interface. In addition, we see a need to add storage providers like Amazon S3 and IBM Cloud Object Storage to MPSC . These enhancements to the MPSC framework would drastically improve its value and versatility for scheduling serverless applications.

ACKNOWLEDGMENTS

The authors would like to thank Brandon Hedden and other members in Distributed Systems Research Lab at Washington State University, for their valuable inputs and feedback on this research.

REFERENCES

[1] 2018. AWS Lambda Limits. https://docs.aws.amazon.com/lambda
[2] 2018. Google Cloud Functions Limits. https://cloud.google.com/functions/quotas
[3] 2018. IBM Bluemix Limits. https://console.bluemix.net/docs/openwhisk/openwhisk_reference.html
[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. 2009. *Above the clouds: A berkeley view of cloud computing.* Technical Report. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley.
[5] Luciano Baresi, Danilo Filgueira Mendonça, and Martin Garriga. 2017. Empowering Low-Latency Applications Through a Serverless Edge Computing Architecture. In *European Conference on Service-Oriented and Cloud Computing.* Springer, 196–210.
[6] Michael Till Beck, Martin Werner, Sebastian Feld, and S Schimper. 2014. Mobile edge computing: A taxonomy. In *Proc. of the Sixth International Conference on Advances in Future Internet.* Citeseer, 48–55.
[7] Kamil Figiela and Maciej Malawski. [n. d.]. Grafana. http://cloud-functions.icsr.agh.edu.pl/dashboard/db/description
[8] Kiryong Ha, Padmanabhan Pillai, Wolfgang Richter, Yoshihisa Abe, and Mahadev Satyanarayanan. 2013. Just-in-time provisioning for cyber foraging. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services.* ACM, 153–166.
[9] Alex Handy. 2014. Amazon introduces Lambda, Containers at AWS re:Invent. https://sdtimes.com/amazon/amazon-introduces-lambda-containers
[10] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Serverless computation with openlambda. *Elastic* 60 (2016), 80.
[11] Pei-Fang Hsu, Soumya Ray, and Yu-Yu Li-Hsieh. 2014. Examining cloud computing adoption intention, pricing mechanism, and deployment model. *International Journal of Information Management* 34, 4 (2014), 474–488.
[12] Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2016. Quantifying the impact of edge computing on mobile applications. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems.* ACM, 5.
[13] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing.* ACM, 445–451.
[14] Grace A Lewis, Sebastian Echeverría, Soumya Simanta, Ben Bradshaw, and James Root. 2014. Cloudlet-based cyber-foraging for mobile systems in resource-constrained edge environments. In *Companion Proceedings of the 36th International Conference on Software Engineering.* ACM, 412–415.
[15] Garrett McGrath and Paul R Brenner. 2017. Serverless computing: Design, implementation, and performance. In *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on.* IEEE, 405–410.
[16] Ola Salman, Imad Elhajj, Ayman Kayssi, and Ali Chehab. 2015. Edge computing enabling the Internet of Things. In *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on.* IEEE, 603–608.
[17] Josef Spillner, Cristian Mateos, and David A Monge. 2017. FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC. In *Latin American High Performance Computing Conference.* Springer, 154–168.
[18] John A Stankovic. 1985. Stability and distributed scheduling algorithms. *IEEE transactions on software engineering* 10 (1985), 1141–1152.
[19] Mario Villamizar, Oscar Garces, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, and Angee Zambrano. 2016. Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. In *the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid).* IEEE, 179–182.
[20] Dale Willis, Arkodeb Dasgupta, and Suman Banerjee. 2014. ParaDrop: a multi-tenant platform to dynamically install third party services on wireless gateways. In *Proceedings of the 9th ACM workshop on Mobility in the evolving internet architecture.* ACM, 43–48.