SCHEDULING FUNCTIONS-AS-A-SERVICE AT THE EDGE

By

AUSTIN M. ASKE

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Engineering and Computer Science, Vancouver

MAY 2018

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of AUSTIN M. ASKE find it satisfactory and recommend that it be accepted.

Xinghui Zhao, Ph.D., Chair

Xuechen Zhang, Ph.D.

Scott Wallace, Ph.D.

ACKNOWLEDGMENTS

I would like to thank my committee members Dr. Xinghui Zhao, Dr. Xuechen Zhang, and Dr. Scott Wallace for their time and support. Additionally, I would like to thank all the staff and my peers at Washington State University Vancouver for enriching my education and helping me to exceed far beyond my expectations.

SCHEDULING FUNCTIONS-AS-A-SERVICE AT THE EDGE

Abstract

by Austin M. Aske, M.S.
Washington State University
May 2018

Chair: Xinghui Zhao

Following the recent 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 emerged. In this thesis, we present a novel performance monitoring and multi-provider scheduling framework, Painless Scheduling, for scheduling across FaaS providers. Painless Scheduling monitors the performance of serverless providers in real time such as AWS Lambda and Openwhisk, as well as edge resources. Additionally, Painless Scheduling has the ability to dynamically advise clients on the optimal resource to use based on their unique scheduling needs as conditions change for any reason. In our evaluation of Painless Scheduling framework utilizing two cloud providers and a local provider, Painless Scheduling selected a "near-optimal" schedule. When compared to scheduling on a single cloud resource Painless Scheduling provides a 4X speedup across multiple providers in a volatile edge computing environment. We conclude with future research directions for Painless Scheduling framework and more generally future research in FaaS technology.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

DEDICATION

I dedicate this work to Kelsey and Eleanor for their endless love, patients, and support throughout this process.

Chapter 1

INTRODUCTION

1.1 Cloud Computing

Over the last 2 decades, cloud computing has revolutionized the IT industry by providing users the ability to provision on-demand resources in near real-time, more efficiently utilize hosting budgets, and eliminate capital expenses involved in purchasing hardware[1, 2]. With major players Amazon Web Services(AWS) and Microsoft Azure both posting revenues in the tens of billions of dollars in 2017, it is clear that cloud computing has changed the economic landscape of computer infrastructure[3]. In the beginning, AWS Elastic Compute Cloud (EC2) allowed the renting of virtual machines giving users control of the entire software stack starting at the operating system. EC2 and similar services came to be known as Infrastructure-as-a-Service (IaaS), where the base unit of computation is virtual machines. Included in IaaS is storage and network infrastructure where base units are represented as hard disk and software defined network components respectively. The virtualization of hardware infrastructure allowed cloud computing to deliver elasticity and the illusion of infinite capacity for a wide spectrum of developers[1].

Fast forward to the emergence of container virtualization which provided further abstrac-

tion of the underlining hardware and finer subdivision of computation resources on a single shared operating system. Research shows that container virtualization outperforms traditional virtual machines in CPU, memory, I/O, and network operations, making container virtualization an increasingly popular choice for deployment on cloud infrastructures[4, 5]. In addition to performance advances, the container technology has also defined a new finer-grained unit of computational resources, the container. Currently, all major cloud providers offer container hosting services with advanced container orchestration enabling superior fault-tolerance, live migration, and auto-scaling capabilities. Enabled by container technologies, serverless computing further perfects the subdivision of computation resources, elasticity, and abstraction of underlining hardware pioneered by containers[6].

1.2 Serverless Architecture

Starting with Amazon's AWS Lambda offering in 2014, the serverless technology, also known as Functions-as-a-Service (FaaS) architecture, has proven to be more scalable, elastic, developer-friendly, and cost-effective than previous cloud architectures [7, 8, 9]. Serverless 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. In most cases, functions are loosely coupled together through HTTP interfaces, creating a serverless application. 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 parallel and can be horizontally scaled utilizing the full power of the backing hardware.

Serverless technologies shift all server management and execution of functions to the service provider, allowing efficient automatic scaling and fast market development[10]. Server-

less computing enables developers to focus on their code, relieving them from server and deployment details as seen in virtual machines and containers. For example, in the time it takes for a vm to provision, it is possible to deploy a function on AWS Lambda or any other serverless provider. The increased abstraction level provided by FaaS minimizes developer boiler plate code and configuration/management of scaling systems.

In addition to elevating developer overhead, serverless computing also utilizes a fine-grained pay-per-use economic model. This fine-grained pay-per-use model, has shown serverless applications can decrease hosting cost over more traditional hosting models[11]. With many FaaS providers billing in 100 ms increments, customers can dramatically reduce over provisioning of resources. Furthermore, the serverless pay-per-use economic model has no additional cost for concurrent versions, making it possible for developers to utilize rolling updates and/or various experimental versions. For example, the hosting cost associated with 20,000 requests to one version is the same as 15,000 to a stable version and 5,000 to a experimental version.

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.

1.3 Motivation

Serverless technology is becoming more popular with new providers emerging rapidly. The introduction of open-source host agnostic offerings are shown in Table 1.1. 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

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 1.1: Major providers and runtimes.

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[12, 1].

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[13] and Event Gateway[14]. Serverless Framework simplifies and standardizes function development allowing deployment on all of the major FaaS 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 Painless Scheduling framework, a serverless monitoring and scheduling system. Painless Scheduling first monitors the performance of serverless providers in real time. Second, based on the performance results and a user-defined schedule Painless Scheduling can recommend the optimal location to run the user's code. Currently

in alpha, Painless Scheduling allows users to implement scheduling algorithms based on a provider's moving average execution time, resource affinity, and cost of provider. Utilizing Painless Scheduling's adaptive scheduling has shown to increase performance and QoS over conveniently single provider utilization.

1.4 Contributions

In this thesis, we present the following three contributions in the areas of serverless computing and edge computing. To our knowledge, we present Painless Scheduling the first serverless scheduling framework for multiple cloud/edge provider utilization. We propose a user defined scheduling system, which enables developers to optimize FaaS performance for their specific application needs. Finally, we demonstrate that actively monitoring providers and scheduling accordingly across multiple providers through Painless Scheduling framework can achieve performance increases of more than 200% compared to using one provider.

1.5 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 covers related works, including a background of FaaS, quality of service research on FaaS platforms, and research evaluating FaaS in an edge computing context. Chapter 3 describes our design considerations for Painless Scheduling framework. In Chapter 4, we outline our evaluation of the framework and present results. Chapter 5 concludes the research and describes future research directions.

Chapter 2

RELATED WORK

2.1 Serverless Introduction

Since AWS Lambda was introduced in 2014, the serverless paradigm has become increasingly popular and successful for cloud providers[15]. Looking at the search term "Serverless" from Google Trends in Figure 2.1 and the increased amount of publications centered around the topic, we see serverless becoming the paradigm of choice over virtual machines and containers. Given that serverless is a relatively new technology, we will first introduce what it is, it's benefits, current economy, and future trends.

In our research, we claim that serverless computing is a means for abstracting computational resources through the publication of short lived and stateless functions that can be invoked by events. Across providers, the idea of functions and events vary, but typically functions are small self contained blocks of code which can be offloaded and executed in isolation, normally ran in a container. Events can take the form of HTTP request, database events, or calls from another function. One of the key features of serverless is the pay-per-use business model. In this model, users are charged based on the memory allotted to a function and its execution time [16].

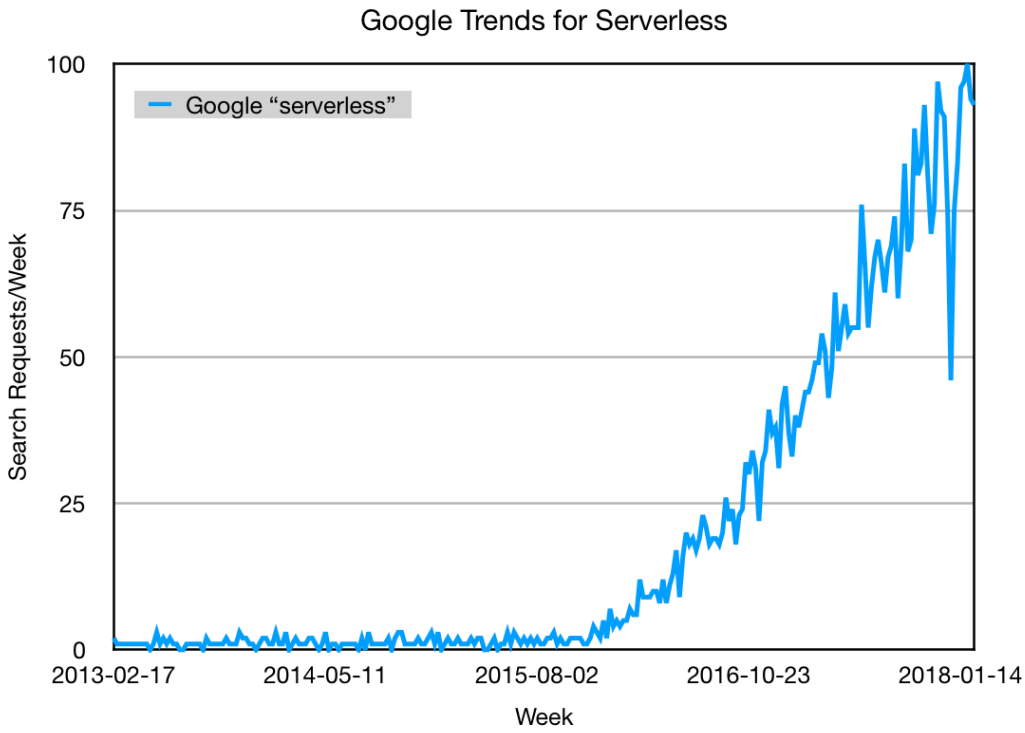


Figure 2.1: Google Trends based on searching “serverless”

To better understand serverless technologies, we will look at some popular methods of virtualization in cloud systems. As shown in Figure 2.2, there are many degrees of virtualization. At the upper left, 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 virtualizated, making for smaller images and better utilization of resources. Finally, the serverless architecture is built atop 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

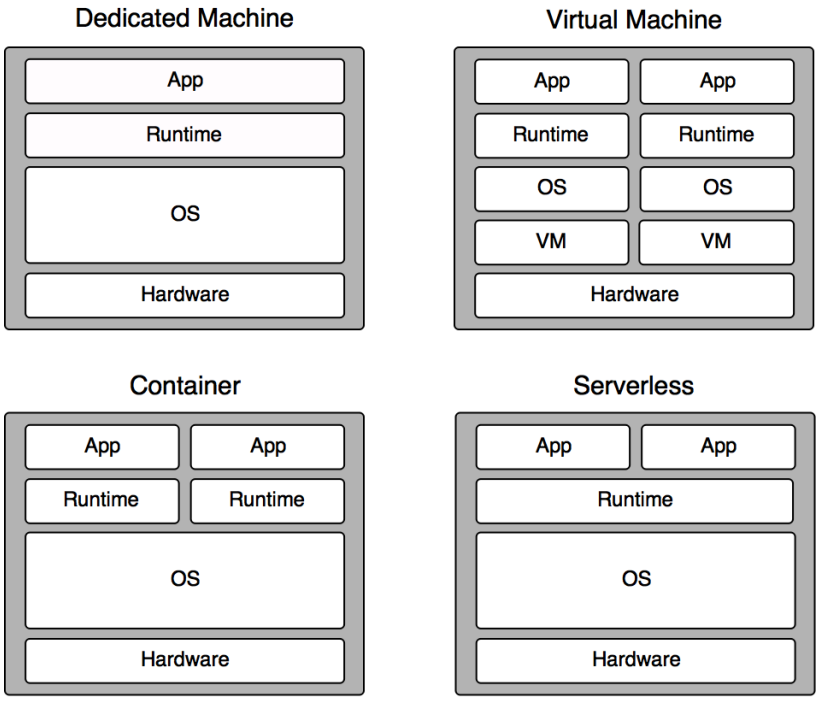


Figure 2.2: Virtualization Architectures

show cold starts on AWS lambda, IBM Bluemix, and Google Cloud Functions executing in the sub 2 second range. However, when utilizing warm functions, or functions ran on already provisioned containers, execution times average under 1 second [17]. Serverless providers handle auto-scaling of concurrent functions based on the volume of requests with many providers having user defined limits between 1000-3000 [16, 18, 19].

2.2 FaaS for High Performance Computing

The work from Spillner et al. [10] published in 2017, evaluates the usability and work-flow of developing serverless applications in four different scientific computing domains. The four applications consisting of mathematics, computer graphics, cryptography, and meteorology where tested using AWS Lambda, Snafu[20] running locally, Python2 running locally. In addition to implementing the four applications, the work also presents a tool for splitting a

traditional monolithic application in to a FaaS application automatically.

In [10], Spillner et al. show the feasibility of serverless computing across a diverse set of scientific computing domains. Furthermore, the research contributes the FaaSification tool, which automates the subdivision of monolithic applications based on function features. While this research focuses on the usability aspects of serverless computing the profiling and feature abstraction of functions for subdividing applications ties into this thesis work. Specifically, in this thesis we look at the performance correlation of a range of functions on different providers in section 4.3.

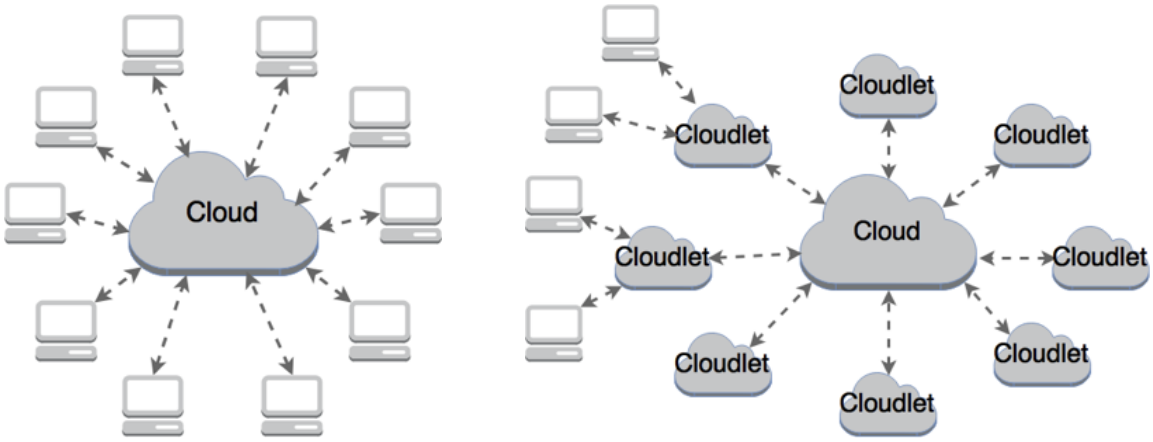
Jonas et al.[21] states that despite many years of availability, scale and elasticity from traditional cloud computing systems are too difficult and that the serverless model could be used instead. Arguing that traditional data processing systems like Apache Hadoop or Spark require decisions on instance 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.

2.3 FaaS to the Edge

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[22, 23, 24]. 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[25, 26, 27]. As seen in Figure 2.2 both VM's and containers incur additional overhead due to runtime and operating system image sizes. Because of this fact, we have seen many publications aimed at minimizing or caching images to speed up provisioning times. In contrast, our research utilizes FaaS architecture for edge computations which keeps provisioning and software overheads to a minimum.



(a) Cloud Architecture (b) Edge Architecture

Figure 2.3: Traditional cloud architecture compared to a hierarchical edge architecture.

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 previous/present research[28]. 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.

Baresi et al. [6] 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. Furthermore, the research shows that heavily constrained computational resources at the edge can be easily overloaded, surrendering any benefits. Baresi et al. outlines much of the foundational work for utilizing FaaS in the context of edge computing, which in many respects is the jumping off point for our work.

2.4 FaaS Benchmark Research

Starting with Amazon's offering of Lambda, we have seen Google, Microsoft, and others offer FaaS platforms. In addition to public cloud offerings, there are a host of open source and potentially privately hosted solutions[29]. As the number of serverless providers expands, the need for performance evaluation tools are at a premium with very few available [30]. Hendrickson et al. [7] present early benchmark work comparing AWS Lambda and AWS Elastic Beanstalk while introducing an open source FaaS platform, Openlambda. Their testing consisted of execution time for heavy and light workloads as well as a Gmail like application workload. This work presents a compelling argument for serverless architecture but doesn't compare performance between providers like our research.

In more recent research from The University of Notre Dame, McGrath et al. [17] present a testing framework for evaluating providers alongside a proposed prototype serverless implementation. The testing framework focuses on the scaling performance of warm and cold

containers in comparison to their prototype. Specifically, the testing framework performs two tests. The first test is a concurrency test in which they incrementally increase the number of concurrent requests while measuring the responses per second. The second is the back off test, where they measure the execution time of each provider while increasing the time between calls. Both of these tests are designed to show the performance differences between executions of functions on warm vs cold containers. In addition to these publications there are also a few online datasets of serverless metrics for the major providers, one of which we will use by Figiela et al. [12].

Chapter 3

PAINLESS SCHEDULING FRAMEWORK

3.1 System Design and Architecture

Painless Scheduling architecture consists of loosely coupled microservices, utilizing REST style communication and multiple extensible plug-able components. Designed for maximum flexibility, Painless Scheduling 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. Painless Scheduling architecture is shown in Figure 3.1 and further details of the components described below.

- **Monitoring Controller** - Is the key component of Painless Scheduling 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

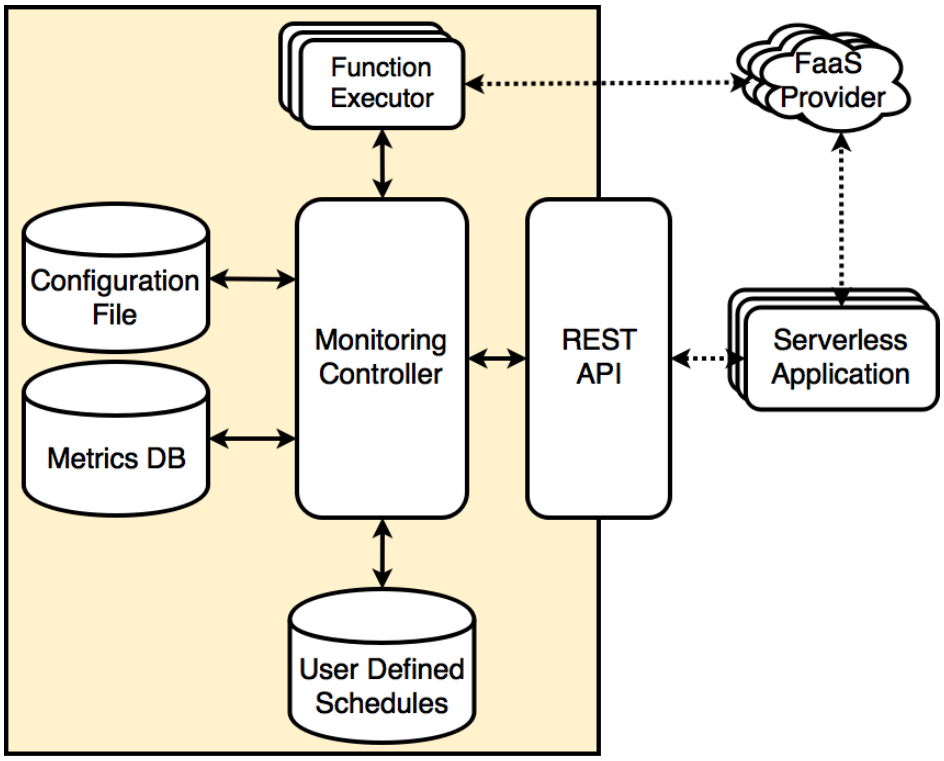


Figure 3.1: Painless Scheduling system architecture diagram

to communicate with the Painless Scheduling Framework. As shown in Table 3.1 the four end-points included in this version are: *Metrics*, *Test*, *AddSchedule*, *Schedule*.

- **Function Executors** - 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.

3.2 Workflow

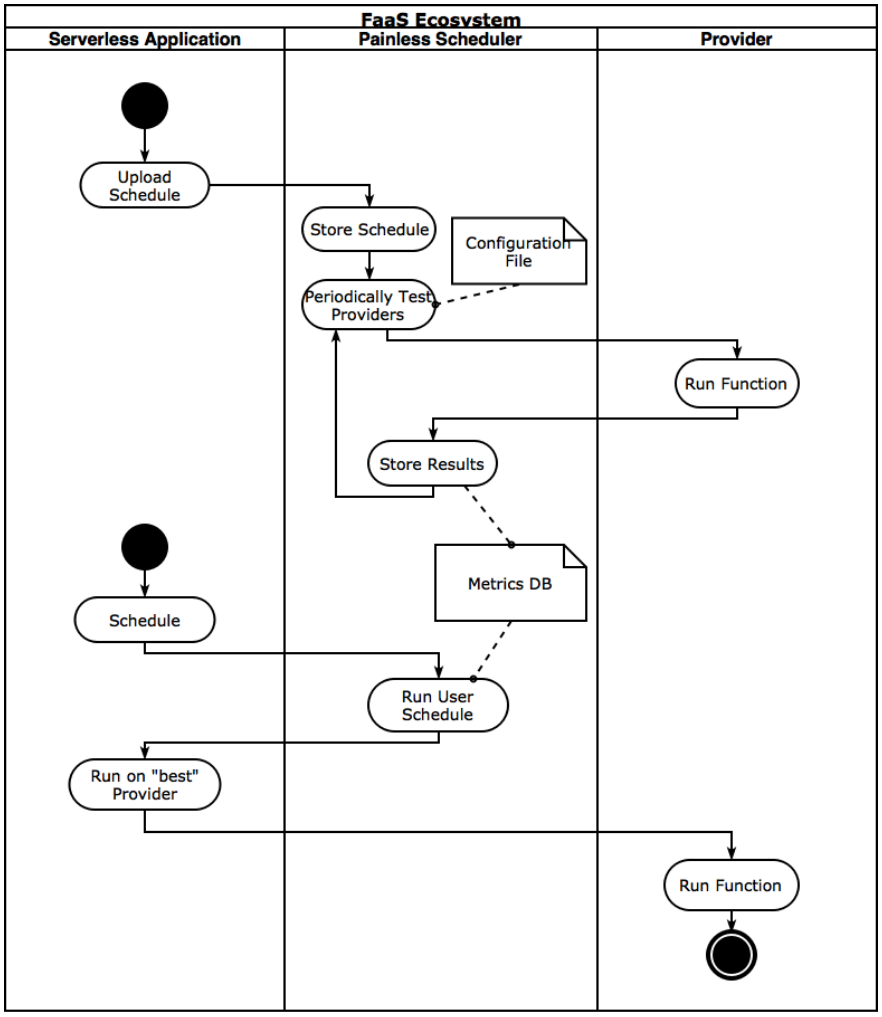


Figure 3.2: Activity Diagram for uploading and scheduling a function in an ecosystem of FaaS providers.

Painless Scheduling framework is a serverless monitoring and scheduling system created to achieve a higher QoS for serverless applications. Shown in Figure 3.2 is the basic workflow of Painless Scheduling 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.

3.2.1 Configuration File

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

Figure 3.3: Example Configuration File

There are two prerequisites for running the work flow shown in Figure 3.2, completing the configuration file and writing a user defined schedule. We will begin by discussing the details included in the configuration file.

The configuration file is loaded at the launch of the Painless Scheduling system and includes user specific information pertaining to each FaaS provider as shown in Figure 3.3. 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.

3.2.2 User Defined Schedules

After the configuration file is completed, the second prerequisite is the creation of a user defined schedule. 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 3.4. 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 Painless Scheduling 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 3.4: Example of a user defined schedule, which utilizes the provider with the lowest average round-trip time.

3.2.3 Painless Scheduling API

Once implemented, the schedule is then submitted to the framework through the REST API using a HTTP/POST request including the source code and an identification string for the schedule. Finally, the schedule can be executed through the REST API passing the schedule's identification string. In response to a schedule request, Painless Scheduling will return the string identifier for the best fit provider. In addition to adding new schedules the REST API exposes a number of other end-points outlined in Table 3.1.

The REST API component of Painless Scheduling was designed to allow easy and standardized integration with serverless applications and other external software. The REST API was built using CherryPy[31] allowing quick and easy interfacing through HTTP. Table 3.1 outlines the API end points and their usage.

Request	Arguments	Return Value
Metrics		JSON Dictionary with metrics for each provider keyed by their providerID
Test	<ProviderID>	
Add Schedule	ScheduleID, Schedule	
Schedule	ScheduleID	ProviderID of the best fit provider

Table 3.1: Painless Scheduling REST API

In addition to the REST API, Painless Scheduling also supplies a plugin interface for the creation of Function Executors. Currently Painless Scheduling includes two Function Executors which handle all the interactions with FaaS providers. Because each cloud provider has unique requirements for invoking and authenticating, functions we designed Function Executors to be user extensible plugins. By leveraging the plugin architecture, Painless Scheduling users can add support for additional FaaS providers or updating current providers as new versions evolve. The two Function Executors included in the prototype support AWS Lambda, IBM Bluemix, and Apache OpenWhisk. It should be noted that IBM Bluemix

runs a version of OpenWhisk which allows for the same Function Executor to handle both providers.

3.2.4 Metrics

Using the results from the Function Executors and the Configuration File, Painless Scheduling records multiple metrics. The first metric is the round trip time for a function to be invoked, executed, and returned to the client. Round trip time can vary because it includes network latency, run-time setup, and execution. Because of the inherent lack of transparency of the underlining software and hardware that functions are being ran on, common metrics are unobtainable. For example, looking at cpu usage from within a function would give little insight to the actual systems usage because most FaaS systems are using containers for isolation. Given that we can't monitor actual system usage of each provider, the round trip execution time has proven to be beneficial in making scheduling decisions, despite its simplicity.

The second metric that Painless Scheduling tracks is the errors or uncompleted function calls per provider. The metric is fairly straight forward, keeping track of the number of failed request per last 100 requests. We believe this to be an important metric for users that want to minimize wasted time from failed requests or have an application that requires extra request stability.

Lastly, Painless Scheduling can measure the cost of using a provider. These values are gathered in the configuration file which is provided by the users upon setup. Currently, these are static values referring to cost per call, and not pulled from the providers. This is a simplified cost model disregarding the fact that cost per call is dependent on function's allocated memory, accumulated calls per month, and the region in which the function is run. Despite its limitations, we believe that cost modeling is an important metric and are

developing plans for an improved model.

Chapter 4

EVALUATION

In the evaluation of Painless Scheduling framework the following three experiments were conducted:

- **Performance Correlation Test** - Comparing the performance of different functions running on each provider.
- **Low Latency Scheduling** - Testing the effectiveness of a user defined schedule optimized for low latency.
- **Low Latency Scheduling with Noisy Neighbor** - Evaluating the performance of a user defined schedule with simulated traffic on the edge provider.

4.1 Providers

To evaluate Painless Scheduling, 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 Ansiable setup. For performance optimization, two changes were made when configuring OpenWhisk. The first change was increasing the limits shown in Table 4.1. 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 4.1: Group variables changed in local OpenWhisk deployment

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 six containerized components as seen in Figure 2.1. Each of the components are run as a Docker container 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 Benchmark Functions

For evaluation purposes, we created three unique benchmark functions which were deployed on all providers. For fairness, each of the functions were deployed with 256 MB of memory using a Python 3 runtime. The first benchmark function created was a factorial function which calculates factorial of 100 fifty times returning the result. The second benchmark

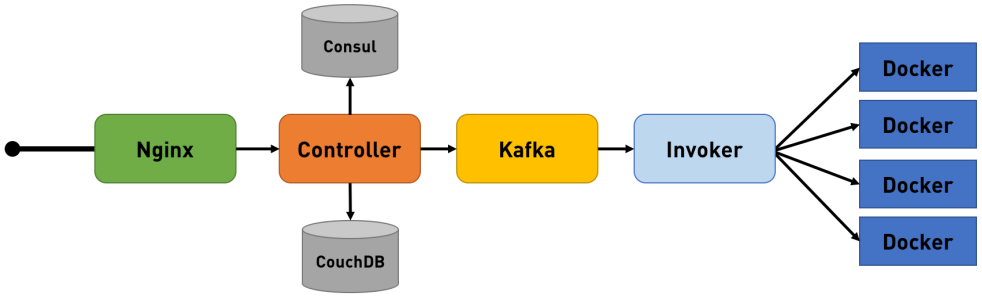


Figure 4.1: OpenWhisk Architecture Diagram. [1]

function, is the result of a Miller-Rabin primality test on one millionth prime (15,485,863). Finally, the last benchmark function is simply an empty function with only a return statement. Using these three benchmark functions deployed on three providers we can begin testing.

4.3 Performance Correlation Test

The performance correlation test was designed to see if we could find a strong correlation between the performance of each provider across different functions. For each provider, we measured the round-trip times of each function. For each function the results were averaged as shown in Figure 4.2.

In addition to the results from each provider on each of the three functions, we conducted ping tests over TCP/IP to each of the regions in which the providers resided. The calculated Pearson Correlation Coefficients for each of the combinations of functions and the ping are shown in Table 4.2

From the Table 4.2 we can conclude that there is a strong correlation between each provider and their round-trip times across different functions. It can also be seen that the raw ping times are not strongly correlated. This shows that network delay is not a good

[1]https://thenewstack.io/behind-scenes-apache-openwhisk-serverless-platform/

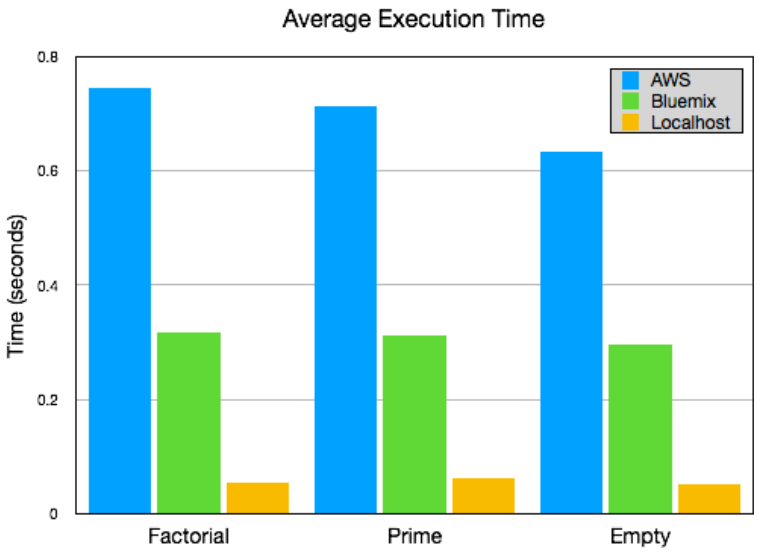


Figure 4.2: Average round trip execution time for each benchmark function running on AWS Lambda, IBM Bluemix, and Openwhisk on localhost.

Correlations

Functions	Factorial	Prime	Empty	Ping
Factorial				
Prime	0.99996			
Empty	0.99885	0.99920		
Ping	0.70863	0.71429	0.74165	

Table 4.2: Pearson Correlation Coefficient values when comparing round trip times of our 3 benchmark functions as well as the ping times of each provider.

indicator of FaaS providers performance. Based on this conclusion we can rule out the need to track ping time in our system and focus on function execution times. It is important to acknowledge that we are not claiming to be able to predict the performance of an arbitrary function, but we can conclude that one service should perform faster than another which we will further illustrate in the remaining testing.

4.4 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 3.4, and in summary picks the provider with the lowest sum of the last 5 function calls. After running Painless Scheduling framework for approximately 400 seconds, testing each provider every 5 seconds, it is clear that the local OpenWhisk provider offers superior performance.

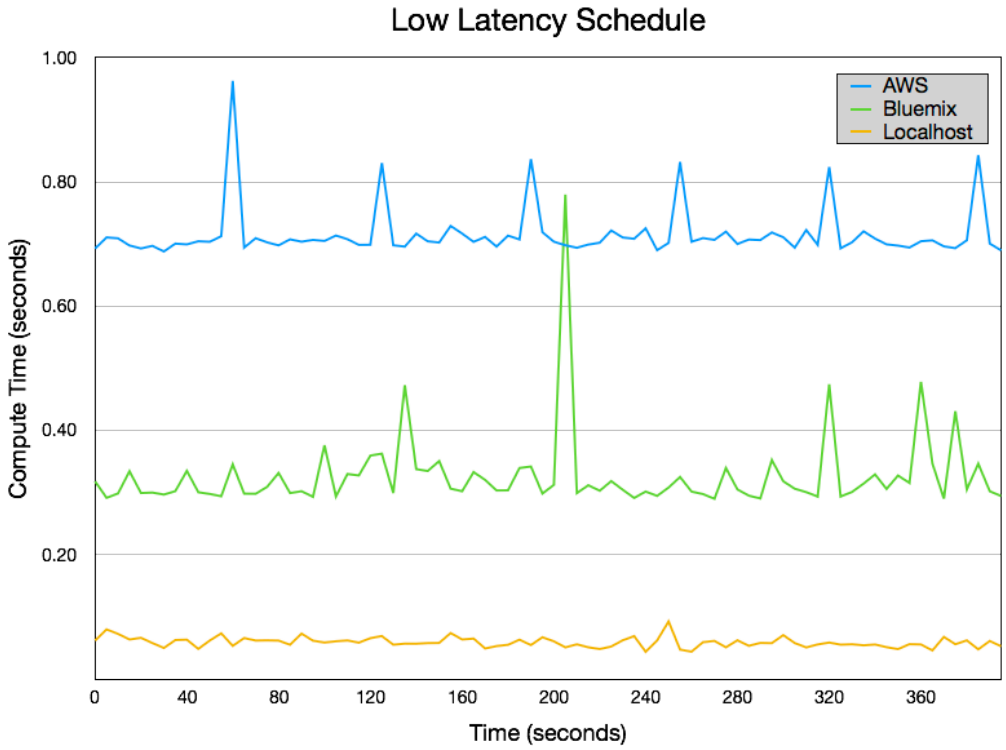


Figure 4.3: The round trip times of each provider running Miller-Rabin primality test on one millionth prime (15,485,863).

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 4.3 compares the results from the experiment looking at the sum of round-trip times, variance, and standard deviation for each provider from the low

	AWS	Bluemix	Localhost	User Schedule
Total Round-Trip (seconds)	57.26446	26.13867	4.72654	4.72654
Variance	0.00184	0.00413	0.00007	0.00007
Standard Deviation	0.04292	0.06430	0.00824	0.00824

Table 4.3: Analysis of Low Latency test for each of the providers.

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[6, 25, 7] but the question still remains how the edge provider performs with additional traffic.

4.5 Low Latency Scheduling with Noisy Neighbor

The low latency scheduling test with noisy neighbor is a continuation of the previous test, low latency schedule. 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 Painless Scheduling is running. This test was performed for approximately 130 seconds while taking measurements every 5 seconds.

As we can see from Figure 4.4, the background color corresponds to the chosen provider based on the provided schedule. Painless Scheduling 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 3.4. Based on this schedule and the frequency of testing providers, Painless Scheduling takes 5 seconds to switch providers. Additionally, Painless Scheduling 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.

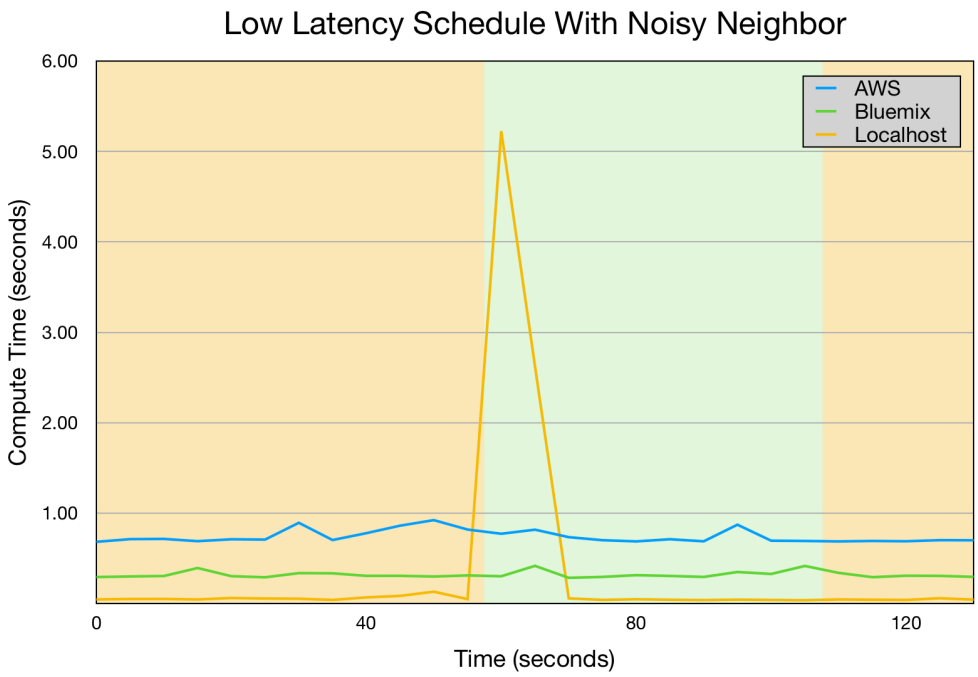


Figure 4.4: The round trip times of each provider with the introduction of 1000 function invocation requests sent to the local provider. Background color represents the best provider based on the user defined schedule.

	AWS	Bluemix	Localhost	User Schedule
Total Round-Trip (seconds)	20.03395	8.61793	9.15323	4.26473
Variance	0.00527	0.00131	1.19856	0.01927
Standard Deviation	0.07263	0.03614	1.09479	0.13881

Table 4.4: Analysis of Low Latency with Noisy Neighbor test for each of the providers

Further analysis was carried out on the low latency schedule with noisy neighbor results, shown in Figure 4.5. Each provider's total execution time is displayed as a baseline and compared to the low latency schedule. In addition, Figure 4.5 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 Painless Scheduling vs AWS Lambda 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 Painless Scheduling framework could be achieved too. In conclusion, our results show that Painless Scheduling framework can provide superior QoS based on application demands, independent of resource congestion.

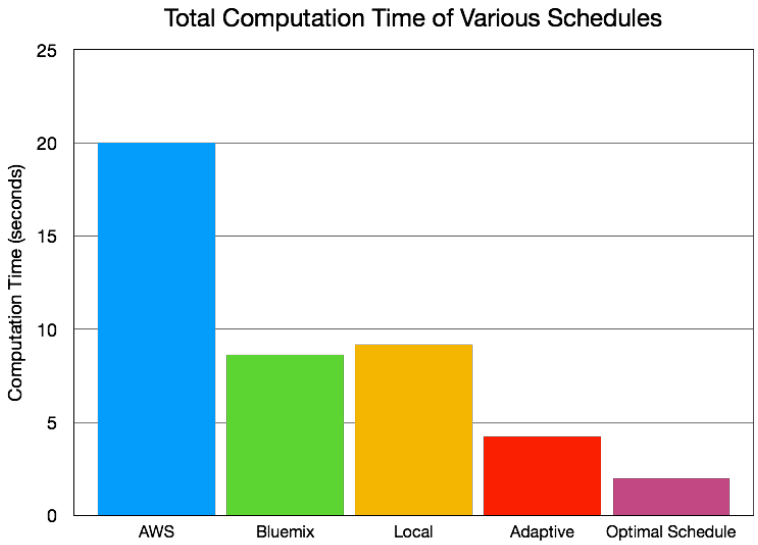


Figure 4.5: Total round trip times for each provider compared with our adaptive schedule and the optimal schedule.

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 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 Painless Scheduling 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.

5.2 Future Work

5.2.1 Painless Scheduling Enhancements

Given the promising results of Painless Scheduling and the gaining popularity of FaaS, we have identified a number of areas for future research. First, we will discuss future enhancements to the Painless Scheduling framework. In our research, the allocated function memory, provider region, and client's location were constant across all tests to isolate providers performance and reduce complexity. However, when deploying a function, these parameters need to be taken into consideration as they will affect cost and performance during invocation. In future development, each parameter could be added to the configurations file to allow for a more comprehensive evaluation of scheduling options. In addition to adding more configuration options, we would like to integrate Painless Scheduling as a plugin for the Serverless framework to increase supported providers through a unified interface [13]. Finally, we see a need to add storage providers like Amazon S3 and IBM Cloud Object Storage to Painless Scheduling . By adding storage providers, the framework can track FaaS providers execution time when accessing different object stores. These enhancements to the Painless Scheduling framework would drastically improve its value and versatility for scheduling serverless applications.

5.2.2 Load Balancing

Taking into consideration that FaaS was introduced only 4 years ago, and that its use in an edge computing context came much later, its usability on the edge is an open research question. As our research shows, resource constrained edge providers can become overloaded resulting in a significant reduction in QoS. To minimize this effect, we believe that further research in load balancing techniques across providers/regions should be done.

BIBLIOGRAPHY

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica *et al.*, "Above the clouds: A berkeley view of cloud computing," Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Tech. Rep., 2009.

[2] P.-F. Hsu, S. Ray, and Y.-Y. Li-Hsieh, "Examining cloud computing adoption intention, pricing mechanism, and deployment model," *International Journal of Information Management*, vol. 34, no. 4, pp. 474–488, 2014.

[3] B. Evans. Microsoft throws down big challenge to amazon: can you top 18.6 billion in cloud revenue? [Online]. Available: https://www.forbes.com/sites/bobevans1/2018/02/01/microsoft-throws-down-big-challenge-to-amazon-can-you-top-18-6-billion-in-cloud-revenue/

[4] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE, 2015, pp. 171–172.

[5] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015, pp. 386–393.

[6] L. Baresi, D. F. Mendonça, and M. Garriga, "Empowering low-latency applications through a serverless edge computing architecture," in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2017, pp. 196–210.

[7] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," *Elastic*, vol. 60, p. 80, 2016.

[8] A. Handy. (2014) Amazon introduces lambda, containers at aws re:invent. [Online]. Available: https://sdtimes.com/amazon/amazon-introduces-lambda-containers

[9] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano *et al.*, "Infrastructure cost comparison of running

web applications in the cloud using aws lambda and monolithic and microservice architectures," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE, 2016, pp. 179–182.

[10] J. Spillner, C. Mateos, and D. A. Monge, "Faaster, better, cheaper: The prospect of serverless scientific computing and hpc," in *Latin American High Performance Computing Conference*. Springer, 2017, pp. 154–168.

[11] G. Adzic and R. Chatley, "Serverless computing: economic and architectural impact," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 884–889.

[12] K. Figiela and M. Malawski. Grafana. [Online]. Available: http://cloud-functions.icsr.agh.edu.pl/dashboard/db/description

[13] Serverless. [Online]. Available: https://serverless.com

[14] Api gateway. [Online]. Available: https://serverless.com/event-gateway/

[15] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 162–169.

[16] (2018) Aws lambda limits. [Online]. Available: https://docs.aws.amazon.com/lambda

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

[18] (2018) Ibm bluemix limits. [Online]. Available: https://console.bluemix.net/docs/openwhisk/openwhisk_reference.html

[19] (2018) Google cloud functions limits. [Online]. Available: https://cloud.google.com/functions/quotas

[20] J. Spillner, "Snafu: Function-as-a-service (faas) runtime design and implementation," *arXiv preprint arXiv:1703.07562*, 2017.

[21] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 445–451.

[22] O. Salman, I. Elhajj, A. Kayssi, and A. Chehab, "Edge computing enabling the internet of things," in *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*. IEEE, 2015, pp. 603–608.

[23] M. T. Beck, M. Werner, S. Feld, and S. Schimper, "Mobile edge computing: A taxonomy," in *Proc. of the Sixth International Conference on Advances in Future Internet*. Citeseer, 2014, pp. 48–55.

[24] W. Hu, Y. Gao, K. Ha, J. Wang, B. Amos, Z. Chen, P. Pillai, and M. Satyanarayanan, "Quantifying the impact of edge computing on mobile applications," in *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM, 2016, p. 5.

[25] G. A. Lewis, S. Echeverría, S. Simanta, B. Bradshaw, and J. Root, "Cloudlet-based cyber-foraging for mobile systems in resource-constrained edge environments," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 412–415.

[26] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan, "Just-in-time provisioning for cyber foraging," in *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 2013, pp. 153–166.

[27] D. Willis, A. Dasgupta, and S. Banerjee, "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, 2014, pp. 43–48.

[28] J. A. Stankovic, "Stability and distributed scheduling algorithms," *IEEE transactions on software engineering*, no. 10, pp. 1141–1152, 1985.

[29] S. Dorodko. Faas: Function hosting services and their technical characteristics. [Online]. Available: https://blog.zhaw.ch/icclab/faas-function-hosting-services-and-their-technical-characteristics/

[30] A. Iosup, G. Andreadis, V. Van Beek, M. Bijman, E. Van Eyk, M. Neacsu, L. Overweel, S. Talluri, L. Versluis, and M. Visser, "The opendc vision: Towards collaborative datacenter simulation and exploration for everybody," in *Parallel and Distributed Computing (ISPDC), 2017 16th International Symposium on*. IEEE, 2017, pp. 85–94.

[31] A minimalist python web framework. [Online]. Available: http://cherrypy.org