PES
UNIVERSITY

Dissertation on

Ensuring QoS for Serverless Computing

Submitted in partial fulfillment of the requirements for the award of degree of

**Bachelor of Technology
in
Computer Science & Engineering**

Submitted by:

Saahitya E 01FB16ECS322

Under the guidance of

Internal Guide
Dr K V Subramaniam
Professor,
PES University

January – May 2020

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

FACULTY OF ENGINEERING

CERTIFICATE

This is to certify that the dissertation entitled

Ensuring QoS for Serverless Computing

is a bonafide work carried out by

Saahitya E 01FB16ECS322

In partial fulfillment for the completion of eighth-semester project work in the Program of Study Bachelor of Technology in Computer Science and Engineering under rules and regulations of PES University, Bengaluru during the period Jan. 2020 – May. 2020. It is certified that all corrections/suggestions indicated for internal assessment have been incorporated in the report. The dissertation has been approved as it satisfies the 8th-semester academic requirements in respect of project work.

Signature	Signature	Signature
Dr. K V Subramaniam	Dr. Shylaja S S	Dr. B K Keshavan
Professor	Chairperson	Dean of Faculty

External Viva

Name of the Examiners	**Signature with Date**
1. ____________________________	____________________________
2. ____________________________	____________________________

DECLARATION

I hereby declare that the project entitled **Ensuring QoS for Serverless Computing** has been carried out by us under the guidance of Dr. K V Subramaniam, Professor and submitted in partial fulfillment of the course requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** of **PES University, Bengaluru** during the academic semester January – May 2020. The matter embodied in this report has not been submitted to any other university or institution for the award of any degree.

01FB16ECS322 **Saahitya E**

Sahitya

ACKNOWLEDGEMENT

I would like to express my gratitude to Dr. K V Subramaniam, Center for Cloud

Computing and Big Data, PES University for his continuous guidance, assistance and encouragement throughout the development of this project.

I take this opportunity to thank Dr. Shylaja S S, Chairperson, Department of Computer Science and Engineering, PES University, for all the knowledge and support I have received from the department. I am deeply grateful to Dr. M. R. Doreswamy, Chancellor, PES University, Prof. Jawahar Doreswamy, Pro-Chancellor – PES University, Dr. J Suryaprasad, Vice-Chancellor, PES University for providing to us various opportunities and enlightenment every step of the way. I am thankful for Prof. Preet Kanwal and Prof. Sangeeta V for coordinating the project related activities.

I wish to thank my mother and father for their unwavering support and encouragement.

ABSTRACT

Since the introduction of AWS Lambda in 2014, the adoption of serverless technologies has increased significantly. Average weekly invocations on Serverless platforms of cloud providers have shown a 209% increase in over the last 12 months[1].

Serverless platforms providing performance guarantees to enterprise customers is important, but is indifferent to the importance of resource utilization by the platform and overall economic viability.

We quantify latencies of the functions that were run including individual component-wise latency of the functions when running the workload. Modification to improve QoS by implementing a simple heuristic that modified cpu shares of runtimes environments was made and effects were observed when the same workload was run again.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Overview

There have been lots of new trends and technologies in cloud computing over the past decade. One such important and compelling technology is serverless computing, also called FaaS(Function-as-a-Service), which abstracts away deployment and back-end operations away from the developer allowing him/her to focus on the coding. Serverless computing involves developers writing applications as a collection of stateless functions which the serverless platforms deploy automatically.

Serverless computing is differentiable from other cloud computing paradigm like IaaS(Infrastructure as a Service) in the following ways [2] :-

1) **Decoupled computation and storage**. The storage and computation scale separately and are provisioned and priced independently. In general, the storage is provided by a separate cloud service and the computation is stateless.

2) **Executing code without managing resource allocation**. Instead of requesting resources, the user provides a piece of code and the cloud automatically provisions resources to execute that code.

3) **Paying in proportion to resources used instead of for resources allocated**. Billing is by some dimension associated with the execution, such as execution time, rather than by a dimension of the base cloud platform, such as size and number of VMs allocated

Serverless is clearly an increasingly popular paradigm due to the flexibility it offers(ability to write functions in a variety of languages), ease of use(no need to allocate VM's manually i.e. operation work) and very low overall cost(20 cents per million requests for functions that consume 128 Mb of memory).

Yet, serverless computing faces some major challenges and limitations that prevent it from mass adoption. One such major limitation that this project addresses is the nonexistence of performance guarantees for the function invocations.

Quality of service (QoS) is the description or measurement of the overall performance of a service such as cloud computing, particularly as seen by users of the service.

As part of this project, we try to build a prototype for a controller to ensure QoS for serverless platforms.

1.2 Scope

The scope of the project involves identifying what factors primarily influence the QoS(Quality of Service) of a function that is run on the serverless platform Apache OpenWhisk. It involves using Linux cgroup technology intelligently to ensure QoS(Quality of Service) for the various functions that are run on the serverless platform by allocating system resources appropriately. QoS can be measured by different metrics, but the scope of the project limits looking at QoS only from the lens of latency of function invocation.

The project also involves identifying the functions to run, the workloads characteristics like request distribution, request rate, set of functions, the tools to record system metric data and function latency data from the platform and devising experiments to run to show the impact of implementing the above quantitatively.

The scope does not involve ensuring QoS by dynamically allocating more compute or system resources such as increasing the number of VMs to scale the number of invokes, etc. Also the use of prewarmed containers is not included in the scope.

1.3 Objective

Using serverless platforms generally means that cloud customers don't have control over infrastructure management which makes it hard for developers to ensure performance. But for cloud developers to use serverless platforms in their core business applications and in more use cases, the serverless platforms have to provide SLA(Service Level Agreement) that agrees upon a QoS(Quality of Service) for an application as is common in other cloud computing offerings. SLA's and QoS's often serve an important regulatory purpose. Server uptime for defense applications and data security for hospitals is very important as they involve matters of life and death. Hence there is a need for serverless platforms to ensure QoS for functions and the option for cloud customers to specify QoS level required for different functions.

This study focuses on implementing a mechanism to ensure QoS for different functions of different users running on a serverless platform and quantitatively measuring the impact of implementing such a mechanism by running experiments that simulate real-world use case scenarios so that serverless platforms can have more robust functionality and thus be used in more critical applications.

CHAPTER 2

RESEARCH BACKGROUND

2.1 Literature Survey

Kumar[3] showed the effects of contention on AWS Lambda and OpenFaas, two Serverless Platforms. Microbenchmarks and applications were developed to quantify the baseline costs of both the serverless platforms with the traditional orchestrator - Kubernetes in an isolated system and co-located workload.

Kishore et al[4] discusses the resource manager for Apache Cassandra - a distributed No-SQL database by modifying the number of CPU cores an Apache Cassandra node is allocated depending on system metrics like Native-Transport-Requests(no of requests to the node), etc. Regression tree using the system metrics as features is used to predict node's performance and modify the resource allocation of the node to ensure QoS is maintained.

Suresh et al[5] is a paper that describes a function level scheduler that modifies CPU shares for a function based on a simple heuristic of observing latency degradation. The work presents FnSched, a function-level scheduler designed to minimize provider resource costs while meeting customer performance requirements. FnSched works by carefully regulating the resource usage of colocated functions on each invoker, and autoscaling capacity by concentrating load on few invokers in response to varying traffic.

Reza et al[6] is a paper that describes a model predictive resource controller that describes an elaborate effort to control QoS in a serverless platform by using the ARIMA model to predict future QoS violations and then use PSO to optimize resources such as CPU and RAM.The solution makes appropriate resource allocation decisions by predicting the future rate of events coming to the system as well as considering the QoS enforcements requested by each function. ARIMA(Auto Regressive Integrated Moving Average) model is a time series forecasting model. The ARIMA model explains a given time series based on its own past values, that is, its own lags and the lagged forecast errors, so that the ARIMA model can be used to forecast future values.

Lo et al[7] describes a feedback-based controller called Heracles that enables the collocation of latency-critical service and best effort tasks. The controller dynamically manages multiple hardware and software isolation mechanisms, such as CPU, memory, and network isolation, to ensure that the latency-sensitive job meets latency targets while maximizing the resources given to best-effort tasks.

The paper goes on to describe a datacenter specific approach that the engineers at Google used to improve server utilization in Google datacenters. A critical insight described in the paper is preventing interference between the LC(latency-critical) tasks and BE(best-effort) tasks to ensure SLO is maintained for LC tasks. The isolation techniques mentioned are cgroup isolation, LLC(Low-Level Cache) isolation, power isolation, network traffic isolation. The paper goes into detail describing the implementation of the Heracles controller and further describes the way the controller was evaluated.

CHAPTER 3

Serverless Computing

3.1 Overview of Serverless Platforms

Serverless computing is a relatively new paradigm and major serverless platforms have different architectures and other various differences. The major serverless platforms are :-

1. **AWS Lambda** is currently the industry leader for serverless platforms and was created all the way back in 2015.It uses a different virtualization technology called Firecracker to execute code snippets.

2. **Azure Functions** like AWS Lambda offers function invocations at a very cheap price. But the main difference is that Azure functions tries to maintain availability by using web workers to call the functions during inactivity to decrease the difference between cold and warm start.

3. **Google Cloud Functions** only allows functions to be written in node js in a marked contrast to other platforms that allows functions to be written in many different programming languages to increase flexibility.

4. **Apache OpenWhisk** originally was made by IBM and then open sourced as an Apache OpenSource project. OpenWhisk is the most important open source serverless platform.

The next section describes ApacheOpenWhisk in slight detail explaining the architecture for the platform and the common terminologies.

3.2 OpenWhisk

3.2.1 OpenWhisk Architecture

The architecture of Apache OpenWhisk is important to understand the problem that is ensuring QoS for Serverless Computing and the other aspects of the solution proposed. So a brief overview of the architecture of the Apache OpenWhisk serverless platform is necessary.

The important components of the Apache OpenWhisk platform are[9] :-

1. **Nginx**

   This open source web server exposes the public-facing HTTP(S) endpoint to the clients. It is primarily used as a reverse proxy for the API.

2. **Controller**

After a request passes through the reverse proxy, it hits the Controller, which acts as the gatekeeper of the system. Written in Scala, this component is responsible for the actual implementation of the OpenWhisk API. It performs the authentication and authorization of every request before handing over the control to the next component. Think of this as an orchestrator of the system which will decide the path that the request will eventually take.

3. CouchDB

The state of the system is maintained and managed in CouchDB. The credentials, metadata, action source code, activation records,etc are stored in CouchDB. It is used to verify credentials of function invokee and used to initialize the runtime container environment.

4. Kafka

Apache Kafka is used for building the pipeline queue between controller and invoker. Kafka buffers the messages sent by the Controller before delivering them to the Invoker. When Kafka confirms that the message is delivered, The Controller immediately responds with the Activation ID.

5. Invoker

The Invoker tackles the final stage of the execution process. Based on the runtime requirements and the quota allocation, it uses an action container that acts as the unit of execution for the chosen Action. The Invoker copies the source code from CouchDB and injects that into the Docker container. Once the execution is completed, it stores the outcome of the activation in CouchDB for future retrieval. The Invoker makes the decision of either reusing an existing “hot” container, or starting a paused “warm” container, or launching a new “cold” container for a new invocation.

6. **Action Container**: This is the container that is used to execute the action.This container is made dynamically by the invoker when or reused by the invoker to invoke an action.

3.2.2 Important OpenWhisk Terminology

Action:

Actions are stateless functions (code snippets) that run on the OpenWhisk platform. Actions encapsulate application logic to be executed in response to events

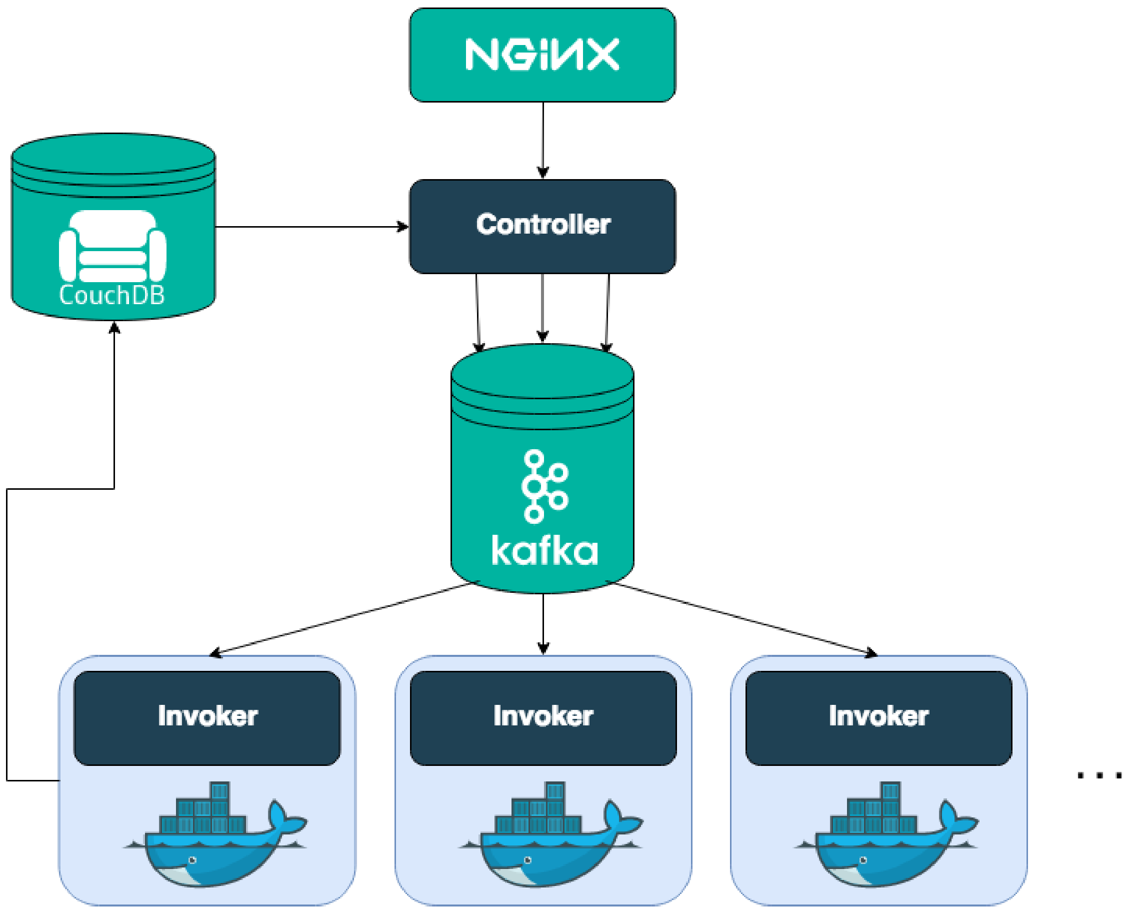


Fig 1: Various components of Apache OpenWhisk[21]

Activation:

Activation is a record of the action execution that is stored in the CouchDB database.It is usually referred with a corresponding activation id that is used to track function requests in OpenWhisk.

Cold Start:

When an action is invoked and an action container has to be made to run the action, the invocation is called a cold start. Cold start involves starting the docker container, initializing it with the source code increasing the latency and hence the number of cold invocations must be decreased.

Warm Start:

When an action is invoked and the action is executed in an already running or paused action container, then the invocation is called a warm start. Warm starts are the ideal invocation type in serverless platforms as they have a much lower latency than corresponding cold start.

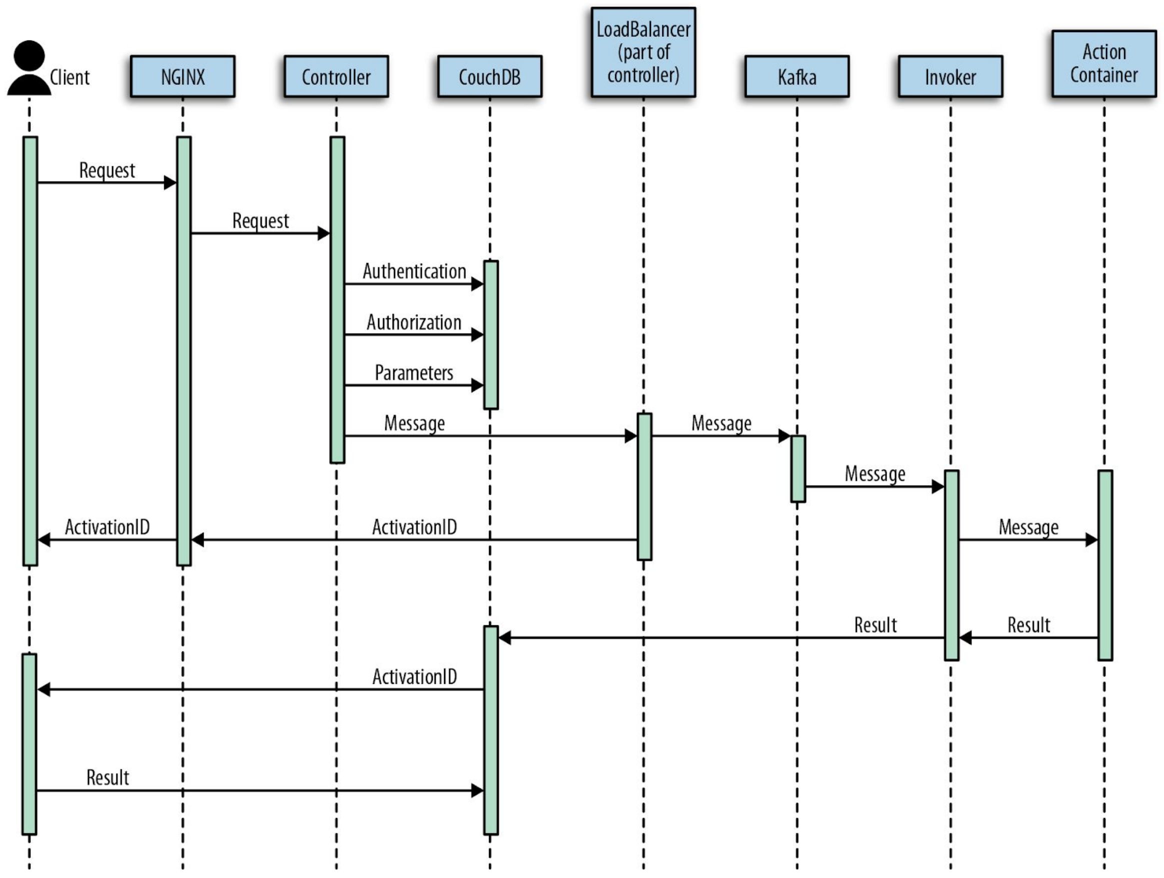


Fig 2: The Flow of an action execution in Apache OpenWhisk[23]

CHAPTER 4

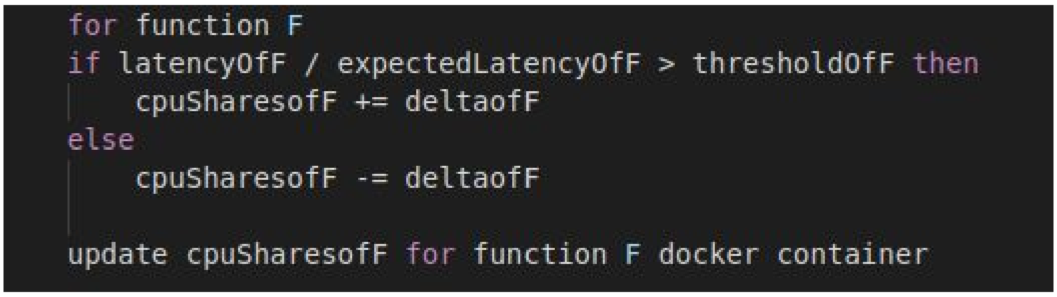
METHODOLOGY

4.1 Proposed Approach

We aim to make sure that the for function executions that are run on a serverless platform QoS is ensured. We plan to approach this problem by first confirming that the single most important factor related to achieving QoS(Quality of Service) is indeed cpu-shares that are allocated to a runtime container in OpenWhisk as considered before the start of the project.

We do this by manually changing the cpu-shares for a function runtime container through the docker CLI before running the function through the OpenWhisk system and observing QoS by measuring latency for the function execution[10,11].

Once it was verified that the above approach showed significant results, a simple heuristic/model/algorithm to change the cpu-shares of the various runtime containers will be chosen.



```
for function F
if latencyOfF / expectedLatencyOfF > thresholdOfF then
    cpuSharesofF += deltaofF
else
    cpuSharesofF -= deltaofF

update cpuSharesofF for function F docker container
```

Fig 3: Pseudocode of the heuristic used to control QoS

Then a simple prototype will be developed by modifying the OpenWhisk source code and using the system metrics and/or latency data from the system to show a simple proof-of-concept.

This prototype's performance in ensuring QoS will be evaluated by running experiments with various workloads that consist of real-world use case applications. Various parameters and measurements will be considered to show the quantitative significance of implementing the prototype.

4.2 CPU shares

CPU shares provide the tasks in a cgroup with a relative amount of CPU time. This means that cgroups with higher CPU shares have a higher proportion of CPU time compared to cgroups with lower CPU shares. This technology can be used by us to control action invocation latency for the action by allocating different amounts of CPU shares for the action containers and ensuring QoS. But it is important to note that CPU share is a soft limit and hence is not applicable when there is no CPU contention.

4.3 High Level Architecture

The main components of the proposed solution are:-

1. **Metric Processor:** The metric processor involves collecting data from the OpenWhisk application and cleaning and modifying the data in the right format. This component gets metric data such as system data, latency data, span data from the OpenWhisk application.
2. **Algorithm/Heuristic/Model:** This component predicts the action(number of cpu shares to allocate for a function) to be taken to ensure QoS and passes the updated cpu shares to the resource manager.
3. **Resource Manager:** This component updates the number of cpu shares for the function runtime container asynchronously.

4.4 Heuristic Overview

The heuristic used in the QoS controller is a simple heuristic that changes the cpu shares of the particular action by lambda based on the current QoS achieved by the function. First the ratio between the current latency and the expected latency for the action is calculated, then if this ratio is greater than the threshold mentioned, then cpu shares for that particular action are increased by delta for the action, else the cpu shares for the same action is decreased by delta. This updated cpu shares for the action are updated for the containers running the action by the resource manager.

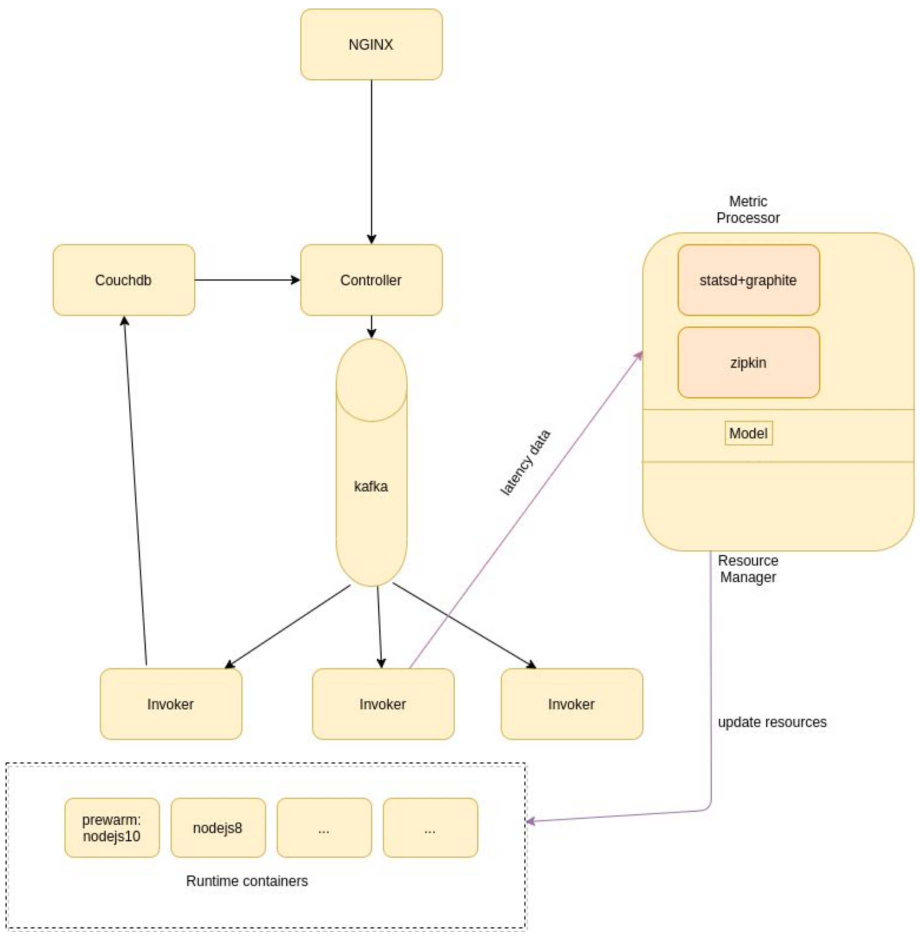


Fig 4: Proposed high level changes to OpenWhisk to include QoS controller

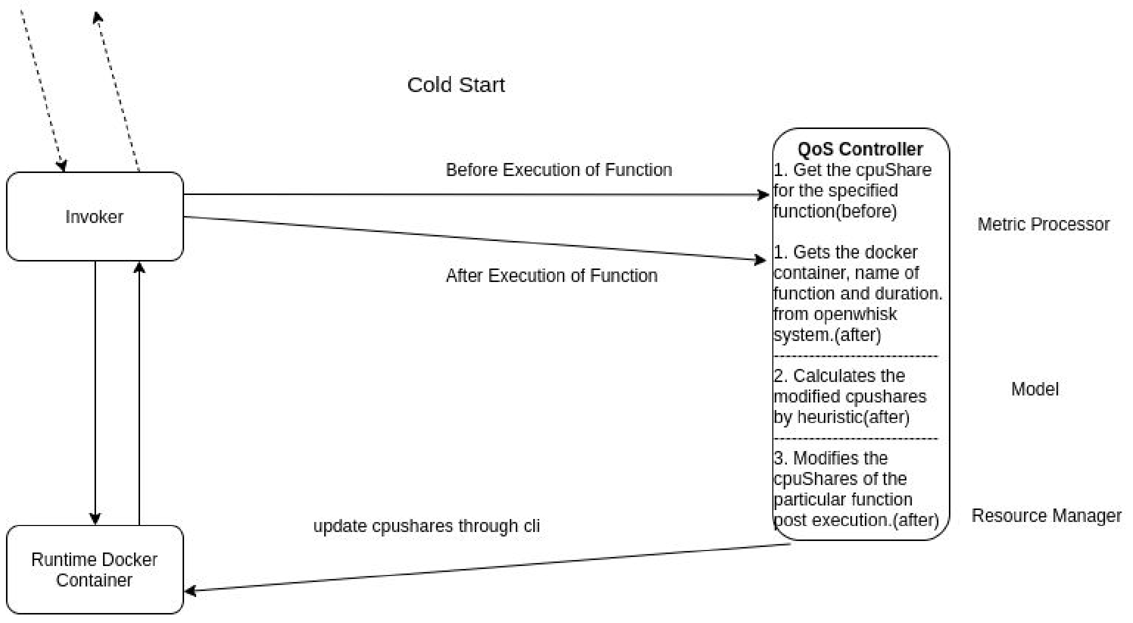

Cold Start
Invoker
Before Execution of Function
After Execution of Function
QoS Controller
1. Get the cpuShare for the specified function(before)
1. Gets the docker container, name of function and duration. from openwhisk system.(after)
2. Calculates the modified cpushares by heuristic(after)
3. Modifies the cpuShares of the particular function post execution.(after)
Metric Processor
Model
Resource Manager
update cpushares through cli
Runtime Docker Container

Fig 5: Additional flow of execution to use QoS controller on a cold start

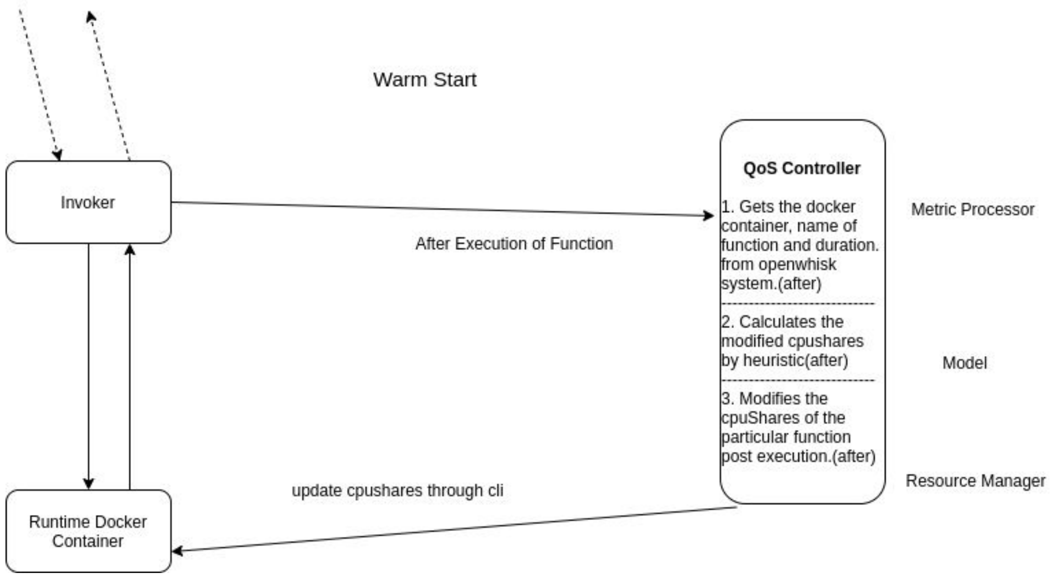

Warm Start
Invoker
After Execution of Function
QoS Controller
1. Gets the docker container, name of function and duration. from openwhisk system.(after)
2. Calculates the modified cpushares by heuristic(after)
3. Modifies the cpuShares of the particular function post execution.(after)
Metric Processor
Model
Resource Manager
update cpushares through cli
Runtime Docker Container

Fig 6: Additional flow of execution to use QoS controller on a warm start

4.4 Flow of Execution in QoS Controller for a request

There are two distinct cases when a request comes into the OpenWhisk system. One of the cases is a **cold start** where the OpenWhisk system has to start the container before initializing code in it and executing and a **warm start** where the action container is reused.

In the case of a warm start the action container already exists and we follow a strategy of updating the CPU shares of the action container after the function invocation.This means that after the function invocation, we send a request to the QoS controller to update the CPU shares of the action container with the current invocation latency.The Model component in the QoS controller uses a heuristic to see how much to update CPU shares by looking at the latency. Finally the Resource Manager component updates the CPU shares of the action container by using Docker CLI.

On the other hand in a cold start, an action container has been created for the action we need to assign the container with the CPU shares for the action, thus the OpenWhisk system requests the CPU shares for the appropriate action from the QoS controller just after the action is invoked. We also have to do an update after the end of the function invocation to use the current request's QoS and appropriately modify CPU shares. The flow of control closely follows that of a warm start.

CHAPTER 5

ENVIRONMENT REQUIREMENTS

5. 1 Hardware Requirements

OpenWhisk can be run in many different setups such as a distributed setup with different OpenWhisk components running in different machines or VMs. But the OpenWhisk setup which was used for running the experiments described in this report was the OpenWhisk local single-machine setup.

The experiments were conducted on the Azure Cloud, specifically on an Azure D-Series virtual machine with 16 GB RAM and 128 GB of premium SSD Memory. The CPU on which the VM is running is an Intel® Haswell 2.4 GHz E5-2673 v3, 8 core processor with max IOPS of 6400 and frequency of 2.3 GHz. The SSD chosen supported IOPS of 500 and a throughput of 100 MB/sec.

The reason for provisioning 128 GB storage was because docker container images and gradle build caches for building OpenWhisk, and the OpenWhisk executables consumed a lot of space.

5.2 Software Requirements

Apache OpenWhisk is a serverless, open-source cloud platform that executes functions in response to user events.OpenWhisk manages the infrastructure, servers and scaling using Docker containers.

Apache OpenWhisk was initially developed as a serverless platform by IBM Cloud and was subsequently open-sourced to become an Apache project.

Apache OpenWhisk was chosen because it was an open source project and facilitated local development compared to other serverless platform offerings from the industry like AWS Lambda, Azure Functions, etc. It also supports a huge variety of programming languages to write functions to run in.

Zipkin is a distributed tracing system. It helps gather timing data of requests in the form of a collection of time that is spent in different components of the system. Features include both the collection and lookup of this data. Interesting data is summarized, such as the percentage of time spent in service, and whether or not operations failed. Zipkin embeds a tracing instrumentation library in OpenWhisk that when configured, asynchronously reports request data to the Zipkin server.

The Zipkin server is run as a docker container when the experiments are run to record the span data(the component specific timing) of the requests and subsequently, this data is used to get metrics like number of cold starts, average time in various components, etc.

Docker is an open-source containerization platform. Docker enables developers to package applications into containers—standardized executable components that combine application source code with all the operating system (OS) libraries and dependencies required to run the code in any environment.

Docker makes it easier, simpler, and safer to build, deploy, and manage containers. It’s essentially a toolkit that enables developers to build, deploy, run, update, and stop containers using simple commands and work-saving automation.

In this scenario, OpenWhisk components like Controller, Invoker, Kafka, runtimes, etc are run as separate docker containers. This is done to reuse existing opensource technologies like CouchDB, zookeeper, Kafka and to easily scale the OpenWhisk application.

The Docker CLI is also used to change the CPU shares of a runtime container to effectively ensure QoS of a function execution in the OpenWhisk platform.

SAR(System Activity Report) is a command that is used to collect, report & save CPU, Memory, I/O usage in Unix-like operating systems. SAR command produces the reports on the fly and can also save the reports in the log files as well.

SAR command is used to collect and monitor CPU utilization and memory utilization statistics when the experiments are run with different workloads. I have used SAR to check the CPU utilization of the system to monitor. The command that i ran was “sar -u 1” that showed the cpu utilization rate and other cpu statistics every 1 second

Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. It began as a simple wrapper around Werkzeug and Jinja and has become one of the most popular Python web application frameworks. Flask offers suggestions but doesn't enforce any dependencies or project layout. It is up to the developer to choose the tools and libraries they want to use.

In this scenario, Flask is used to build our prototype application where the OpenWhisk platform sends latency data to a Flask application endpoint and that is used to take any action regarding modifying cpushare data.

CHAPTER 6

WORKLOADS

The goal of our project was to ensure QoS in serverless platforms, but one big drawback in the serverless research area is the nonexistence of a single standard mature serverless benchmark.

I explored various open-source benchmarks before zeroing in on FunctionBench[12,13]. Other benchmark suites that were explored were not sufficiently computationally intensive. FunctionBench unlike the few other benchmarks had real-world scenario application and was coded in python which was easy to use and run.From the FunctionBench four functions to run were selected that resembled real-world use case scenarios that were modified to run on the OpenWhisk framework.

The function being used in the experiment run were :-

- **Function A - Image resizing**: Pulls an image from the URL passed as a parameter and resizes it to the appropriate dimensions as specified in the parameters to the function.

- **Function B - Generating names**: Uses the characters of the string as starting letters passed as parameters to generate names using a pytorch RNN model.
- **Function C - Model training**: Trains the model to predict review sentiment scores of a text review from the amazon fine food review dataset.First the text is vectorized using a TF-IDF Vectorizer which becomes the input to a regression model, a Logistic Regression model that calculates the sentiment scores of texts from amazon fine food review dataset.
- **Function D - pyaes benchmark**: This function performs private key encryption and decryption,It is a pure-Python implementation of the AES block-cipher algorithm in CTR mode

The 3 workloads that were defined were:-

- **Workload 1** - Only Function A is run
- **Workload 2** - Functions A, C run together
- **Workload 3** - Functions B, D run together

The experiments were run with different workload characteristics :-

- The function requests following both a uniform distribution and a Zipfian distribution for the application without controlling QoS and with controlling QoS using the resource controller. The Zipfian distribution is used with an alpha parameter of 1.3.
- The function following different request rates denoted by number of requests per minute
- The threshold of the function that denoted the ratio of actualLatency to expectedLatency below which the function QoS is acceptable
- The delta of the function is the value by which CPU shares are updated for every action request..

CHAPTER 7

RESULTS

We focused our experiments on demonstrating that by using a QoS controller prototype in the OpenWhisk serverless platform, there is a significant improvement in QoS. On a large machine in any public serverless platform, it is possible that multiple functions are being executed in parallel and the solution, QoS controller prototype must ensure that sufficient performance isolation is provided to the functions, hence progressively the experiments use more functions while running.

We also ran the workloads with different request distributions(uniform and Zipfian) to see how the solution performs in different circumstances. Web request distributions closely match Zipfian distribution and measuring how the solution performs on it is important.

Before starting the experiments the functions were run as warm invocations two times during the idle period and the average of the latencies was taken to get the expected latencies.

The expected latency that was considered for the functions are:

- A: 2000 msec
- B: 24000 msec
- C: 2900 msec
- D: 25000 msec

The experiments were conducted looking at the metrics to compare performance between uncontrolled and controlled Apache OpenWhisk. Here uncontrolled Apache OpenWhisk refers to the original Apache OpenWhisk without the QoS controller running to modify the CPU shares. While controlled Apache OpenWhisk refers to the Apache OpenWhisk with the QoS controller running that modified CPU shares to maintain QoS levels.

The experiment looked at such metrics as time spent in the Kafka queue, the number of cold starts, the total number of violations, the percentile latencies(75, 90, 99) and the probability distribution plots of the requests to get a sense of how they performed. Here total number of violations refers to those requests that didn't meet the relaxed latency expectation that is expected latency times threshold.

The first experiment that was run was just running the Workload 1 with an uniform request distribution. This experiment was run to demonstrate the improvement in a basic setting where only one function is being invoked and when the distribution of requests is uniform. The experiment was run with a uniform request distribution with function invocation rate of 60 invocations per minute with a threshold of 1.1 and delta of 10. Here using 1.1 as threshold means we are allowing a 10% relaxation in latency target.The metrics observed are shown and tabulated below.

	controlled	uncontrolled
Total number of cold starts	4	4
Average time spent in kafka queue(micro secs)	5298.92	5383.43
Total number of violations	108	165

Fig 7: Experiment 1: Various metrics measured on QoS controlled and uncontrolled system

Latency (milliseconds)	controlled	uncontrolled
99	3959.23	4385.399
90	2277.0	2434.20
75	2207.0	2316.0

Fig 8: Experiment 1: Percentile latencies measured on QoS controlled vs uncontrolled system.

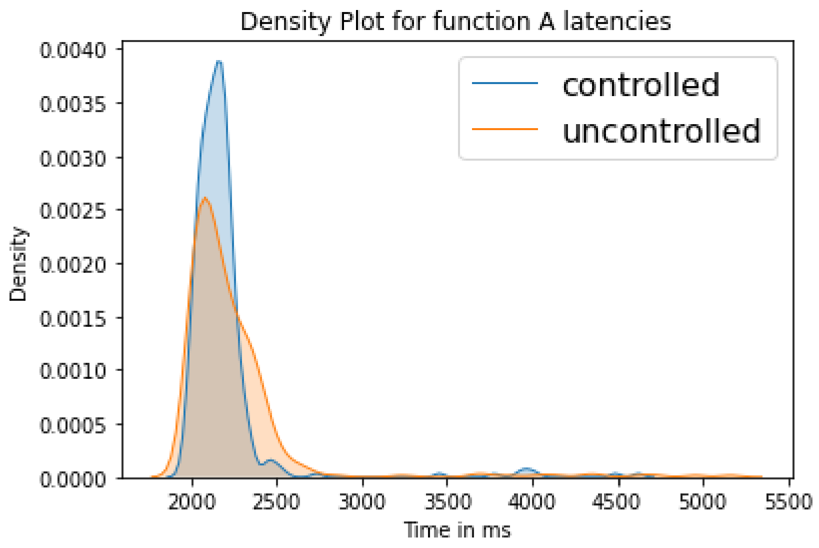


Fig 9: Experiment 1: Latency distribution plot for QoS controlled and uncontrolled system.

We can clearly look at the total number of violations and the percentile latencies to understand that QoS controlled Apache OpenWhisk indeed helps ensure QoS for the requests. There are 30% less violations in the controlled Apache OpenWhisk compared to the uncontrolled OpenWhisk.

The second experiment that was run was using the same workload but with a zipfian distribution. We ran the experiment with function A having a slightly more relaxed threshold of 1.15.

	controlled	uncontrolled
Total number of cold starts	**7**	**4**
Average time spent in kafka queue(micro secs)	**4466.15**	**5367.74**
Total number of violations	**180**	**216**

Fig 10: Experiment 2: Various metrics measured on QoS controlled and uncontrolled system

Latency (milliseconds)	controlled	uncontrolled
99	3766.21	5716.499
90	2696.1	2566.7
75	2447.75	2428.75

Fig 11: Experiment 2: Percentile latencies measured on QoS controlled vs uncontrolled system.

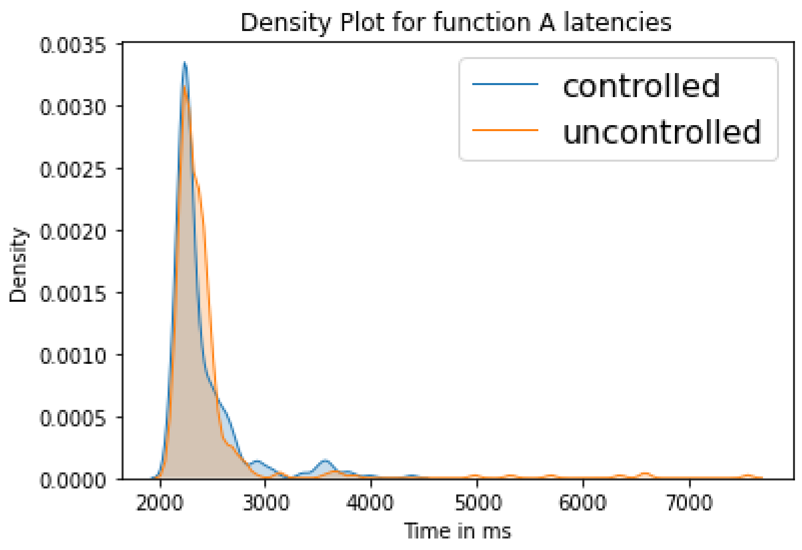


Fig 12:Experiment 2: Latency distribution plot for QoS controlled and uncontrolled system.

Looking at only the total number of violations we can see there is a decrease in the number of violations for the controlled OpenWhisk. But this is not reflected when we look at 90 or 75 percentile latencies.

The next experiment tries to run Workload 2 that consists of two functions - functions A and C with a uniform distribution. Here the function A is run at the rate of 60 requests per minute and has a threshold and delta of 1.15 and 10 respectively. The second function B is run at the rate of 5 requests per minute and a threshold and delta of 1.5 and 10 respectively.

	controlled	**uncontrolled**
Total number of cold starts	**37**	**45**
Average time spent in kafka queue	**9186.55**	**10190.1**
Total number of violations	**226**	**240**

Fig 13: Experiment 3: Various metrics measured on QoS controlled and uncontrolled system

Latency (milliseconds)	controlled	uncontrolled
99	3771.07	3991.35
90	2774.7	2928.5
75	2647.25	2651.0

Fig 14: Experiment 3: Percentile latencies for function A measured on QoS controlled vs uncontrolled system.

Latency (milliseconds)	controlled	uncontrolled
99	33004.2	33430.14
90	31631.7	32939.1
75	31438.0	32602.75

Fig 15: Experiment 3: Percentile latencies for function C measured on QoS controlled vs uncontrolled system.

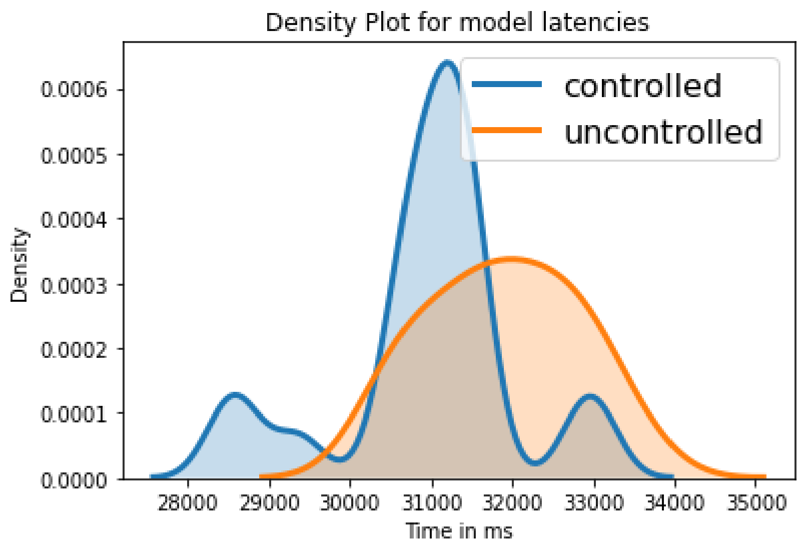


Fig 16:Experiment 3: Latency distribution plot of function C for QoS controlled and uncontrolled system.

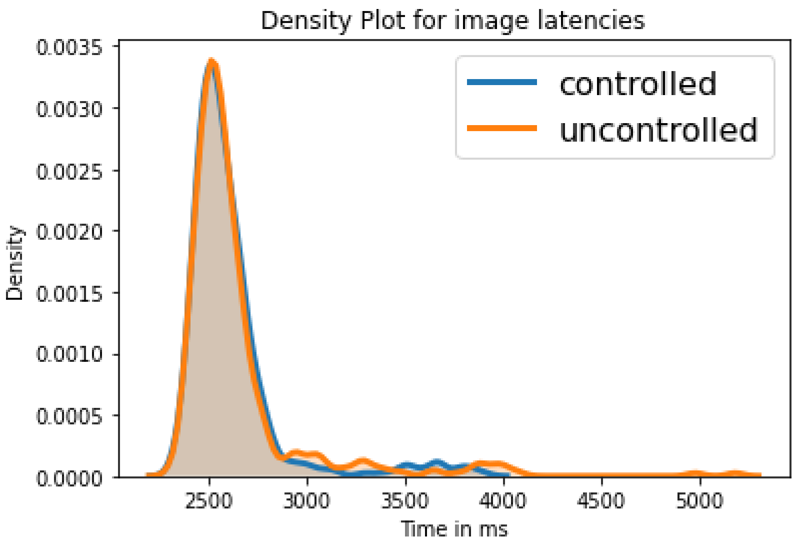


Fig 17:Experiment 3: Latency distribution plot of function A for QoS controlled and uncontrolled system.

I tried running the experiment with the workload 3 with a uniform distribution and function invocation rate of 60 invocations per minute and 5 invocations per minute for functions B and D respectively, but the OpenWhisk system became overloaded with requests. This could be due to the fact that the functions A and C could be less computationally intensive than functions B and D.

So I ran the experiment with 30 invocations and 1.34 invocations per minute(a relatively lighter load) of functions B and D respectively with a threshold of 1.2 and 1.8 and delta of 10 and 50 respectively.

	controlled	uncontrolled
Total number of cold starts	22	21
Average time spent in kafka queue	3988.05	3940.1
Total number of violations	184	188

Fig 18: Experiment 4: Various metrics measured on QoS controlled and uncontrolled system

Latency (milliseconds)	controlled	uncontrolled
99	5388.69	4458.519
90	4247.2	4249.8
75	4071.75	4119.75

Fig 19: Experiment 4: Percentile latencies for function B measured on QoS controlled vs uncontrolled system.

Latency (milliseconds)	controlled	uncontrolled
99	37827.24	34671.20
90	37194.2	34175.79
75	36075.0	33768.5

Fig 20: Experiment 4: Percentile latencies for function D measured on QoS controlled vs uncontrolled system.

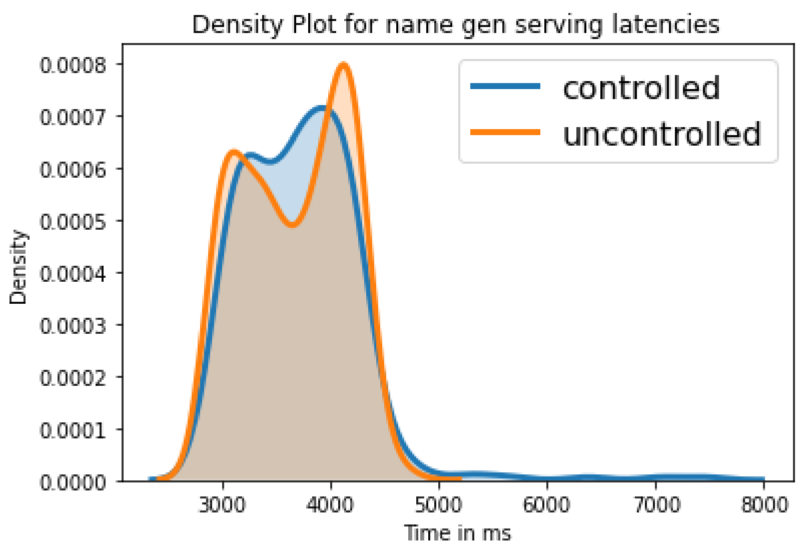


Fig 21:Experiment 4: Latency distribution plot of function B for QoS controlled and uncontrolled system.

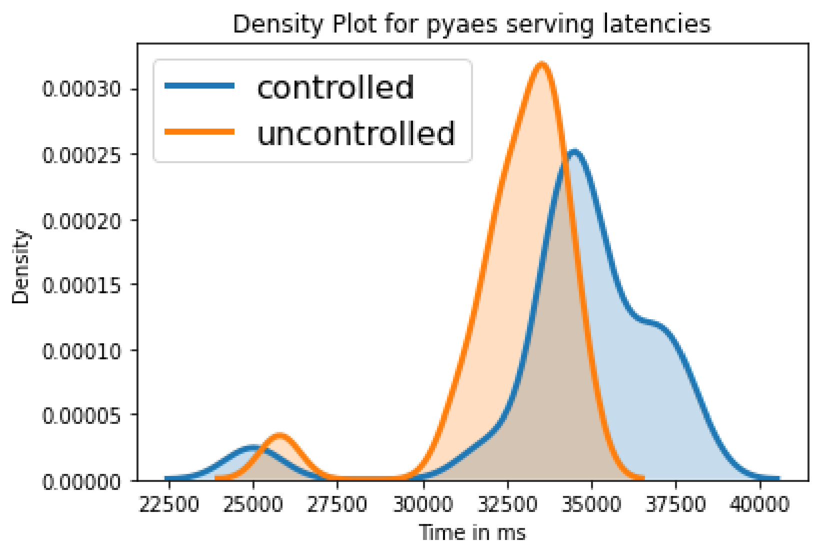


Fig 22:Experiment 4: Latency distribution plot of function D for QoS controlled and uncontrolled system.

Looking at the results of this we see that relaxing the threshold on the function D does help decrease the total violations as CPU shares for function D decreases as it latency easily satisfies the threshold, and hence uses less CPU shares, increasing the CPU time for the function B

The next experiment that was run was increasing the delta for the function to see the effect it would have on the above experiment. The delta for function B used is 20 and delta for function D is 50.

	controlled	uncontrolled
Total number of cold starts	23	21
Average time spent in kafka queue	4139.84	3940.1
Total number of violations	173	188

Fig 23: Experiment 5: Various metrics measured on QoS controlled and uncontrolled system

Latency (milliseconds)	controlled	uncontrolled
99	5183.039	4458.519
90	4232.7	4249.8
75	4016.25	4119.75

Fig 24: Experiment 5: Percentile latencies for function B measured on QoS controlled vs

uncontrolled system.

Latency (milliseconds)	controlled	uncontrolled
99	37000.93	34671.20
90	36623.1	34175.79
75	35784.5	33768.5

Fig 25: Experiment 5: Percentile latencies for function B measured on QoS controlled vs uncontrolled system.

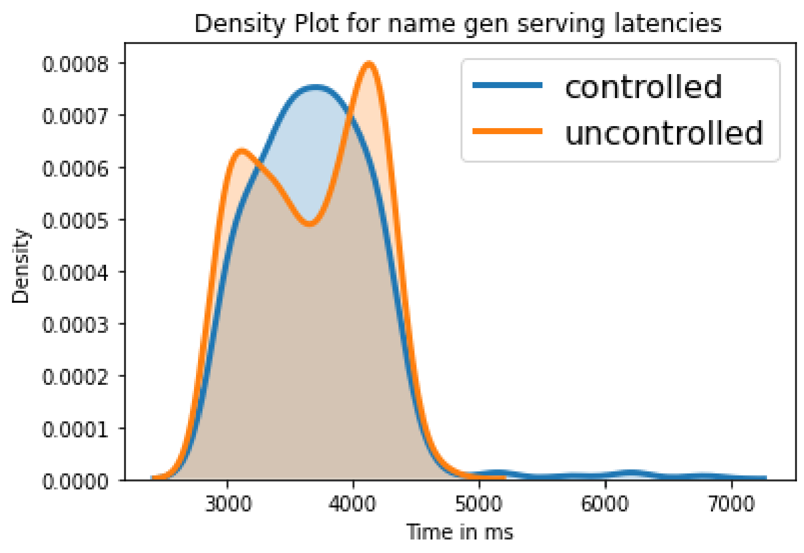


Fig 26:Experiment 5: Latency distribution plot of function B for QoS controlled and uncontrolled system.

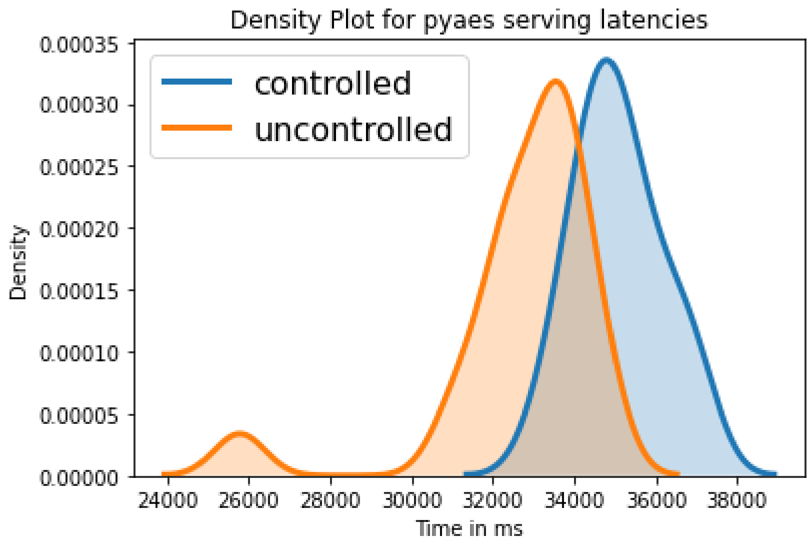


Fig 27:Experiment 5: Latency distribution plot of function D for QoS controlled and uncontrolled system.

Looking at the results we see one marked difference, we see that the total number of violations has decreased by 6 % just by increasing the delta value for function B by 10.

CHAPTER 8

CONCLUSIONS

We ran the above experiments to check the quantitative impact of using a simple heuristic to ensure QoS of the functions invocations in a serverless platform. In observing the above results, there is clearly an increase in QoS level when you use a QoS Controller vs using OpenWhisk that is without a QoS controller.

We see that using a QoS controller shows improvement over most metrics such as average time spent in kafka queue and percentile latencies in most cases. But **using a QoS controller shows a decrease in total number of violations in every single experiment**. But we see that some metrics like total number of cold starts do not necessarily correlate to better QoS level for the function. We also see the effect of changing the delta value (CPU share updation value), which causes the total number of violations to drop in the last experiment.

Outside of the tabulated results, we notice that running the experiments under high load(high request rate) or setting very low threshold for the functions running, often does not show significant improvement when using the QoS controller to ensure QoS due to resource constraints.

Also noticed was that when the QoS controller operates with a threshold, calculating the total number of violations by using a slightly higher threshold greatly decreases the total number of violations as there are many requests that are at the edge of the threshold.

In conclusion, implementing a QoS controller using the concept of updating CPU shares is suitable and feasible for ensuring QoS in a serverless platform. The QoS controller gives the added advantage of specifying QoS level required to the cloud user making it more flexible.

CHAPTER 9

FUTURE WORK

1. Run on bigger machines with CPUs with more cores to measure performance.
2. Use system metrics in augmentation with latency data and request data to ensure QoS.
3. Figuring out the significance of dynamically provisioning system resources like extra VMs, extra memory to ensure QoS for serverless platforms
4. Figuring out whether using the Real-Time scheduler(RT) rather than the Completely Fair scheduler(CLS) to ensure QoS for serverless platforms is feasible and significant or use other ways of restraining CPU.
5. Figuring out quantitative significance of using request timeouts to improve the overall QoS of the functions.

CHAPTER 10

REFERENCES

1. https://newrelic.com/resources/ebooks/serverless-benchmark-report-aws-lambda-2020

2. E. Jonas et al. Cloud programming simplified: A Berkeley view on serverless computing. Technical report, EECS Department, University of California, Berkeley

3. P. Kumar, QoS and Efficiency for FaaS Platforms, Thesis paper

4. Y. Kishore, N. H. V. Datta, K. V. Subramaniam and D. Sitaram, "QoS Aware Resource Management for Apache Cassandra," *2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW)*, Hyderabad, 2016, pp. 3-10.

5. Amoghavarsha Suresh, Anshul Gandhi: FnSched: An Efficient Scheduler for Serverless Functions. WOSC@Middleware 2019(Workshop of Serverless Computing)

6. HoseinyFarahabady, MohammadReza et al. “A QoS-Aware Resource Allocation Controller for Function as a Service (FaaS) Platform.” *ICSOC* (2017).

7. D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, Portland, OR, 2015, pp. 450-462.

8. Apache. OpenWhisk. https://github.com/apache/openwhisk.

9. An Architectural View of Apache OpenWhisk, https://thenewstack.io/behind-scenes-apache-openwhisk-serverless-platform/

10. Jeongchul Kim and Kyungyong Lee, 'Function Bench: A Suite of Workloads for Serverless Cloud Function Service', IEEE International Conference on Cloud Computing 2019, 07/2019

11. Docker Runtime options with Memory, CPUs, and GPUs, https://docs.docker.com/config/containers/resource_constraints/

12. Understanding Linux Container Scheduling, https://engineering.squarespace.com/blog/2017/understanding-linux-container-scheduling

13. Function Bench functions: https://github.com/kmu-bigdata/serverless-faas-workbench

14. AWS. Lambda. https://aws.amazon.com/lambda/

15. What is serverless computing? — serverless definition. https://www.cloudflare.com/learning/serverless/what-is-serverless/.

16. The Rise of Serverless Computing, https://cacm.acm.org/magazines/2019/12/241054-the-rise-of-serverless-computing/fulltext

17. StatsD + Graphite, https://github.com/kamon-io/docker-grafana-graphite

18. Zipkin, https://zipkin.io/, https://github.com/openzipkin/zipkin

19. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01

20. https://docs.scala-lang.org/tutorials/scala-for-java-programmers.html

21. https://www.toptal.com/scala/concurrency-and-fault-tolerance-made-easy-an-intro-to-akka

22. https://github.com/apache/openwhisk/blob/master/docs/about.md

23. https://www.oreilly.com/library/view/learning-apache-openwhisk/9781492046158/ch01.html