PES UNIVERSITY

Project Work :

Project Title : Ensuring QoS in Serverless Computing
Project ID : PW20KVS06M
Project Guide : Dr KV Subramaniam
Project Team : Saahitya E (01FB16ECS322)

Report

Problem Statement

- Serverless platforms must provide QoS guarantees as basic requirement to attract customers to use serverless computing.

- Platforms must also look to balance resource utilization with good latency to achieve economic viability.

- Ensuring QoS for business customers (like latency, throughput, errors) is especially important for serious business application scenarios as businesses dont control the deployment related aspects of their application.

PES UNIVERSITY

Problem Statement

Technicalities involved :-

- Different functions have different characteristics(like latency critical, asynchronous, compute intensive, etc).
- Serverless platforms must manage system resources among different functions that are running in the system.
- What policies for resource allocation? How to implement resource management in Apache OpenWhisk(cgroups?)

PES UNIVERSITY

Literature Survey Summary

QoS Aware Resource Management for Apache Cassandra

Kishore,2016 IEEE 23rd International Conference on High Performance Computing Workshops

- **Helped provide the high level system architecture of the solution in terms of defining various components like metric processor, resource manager**

Heracles: Improving Resource Efficiency at Scale

David Lo, ISCA 2015: Annual Symposium on Computer Architecture, June 2015

- **Defined the collocation of different types of jobs and resource management using cgroup technologies**

FnSched: An Efficient Scheduler for Serverless Functions

A Suresh, WOSC '19: Proceedings of the 5th International Workshop on Serverless Computing

- **Defined a heuristic to change resource allocation and workloads**

QoS and Efficiency for FaaS Platforms

Pranav Kumar, MSc Thesis, May 2019

Defined some of statistics to measures in the experiments and workloads

Literature Survey

QoS and Efficiency for FaaS Platforms

Pranav Kumar, MSc Thesis, May 2019

- Defines good set of workloads that the author used to benchmark performance.
- Describes statistics from benchmarking against baseline and gives good set of empirical data from his experiment

- BUT, the author doesn’t talk about implementation. Uses only performance metrics to improve latency.

Literature Survey

QoS Aware Resource Management for Apache Cassandra

Kishore,2016 IEEE 23rd International Conference on High Performance Computing Workshops

- Describes resource provisioning using internal system metrics for a Cassandra cluster based on the specified latency agreement which led to increase in utilization of resources.

PES UNIVERSITY

FnSched: An Efficient Scheduler for Serverless Functions

A Suresh, WOSC '19: Proceedings of the 5th International Workshop on Serverless Computing

- Describes a scheduler that changes cpu-shares for the container **based only upon** the overall latency degradation for a function call.

- Also describes how to scale-in and scale-out the number of invokers based upon load on the serverless platform.

Literature Survey

A QoS-Aware Resource Allocation Controller for Function as a Service (FaaS) Platform

MohammadReza, ICSOC, Oct 2017

- Paper describes good algorithm design that the author used and the resource metrics he optimized for ensuring QoS.
- Also defines a QoS detriment metric using QoS violations only.
- BUT, the author doesn’t talk about implementation.

Literature Survey

Heracles: Improving Resource Efficiency at Scale

David Lo, ISCA 2015: Annual Symposium on Computer Architecture, June 2015

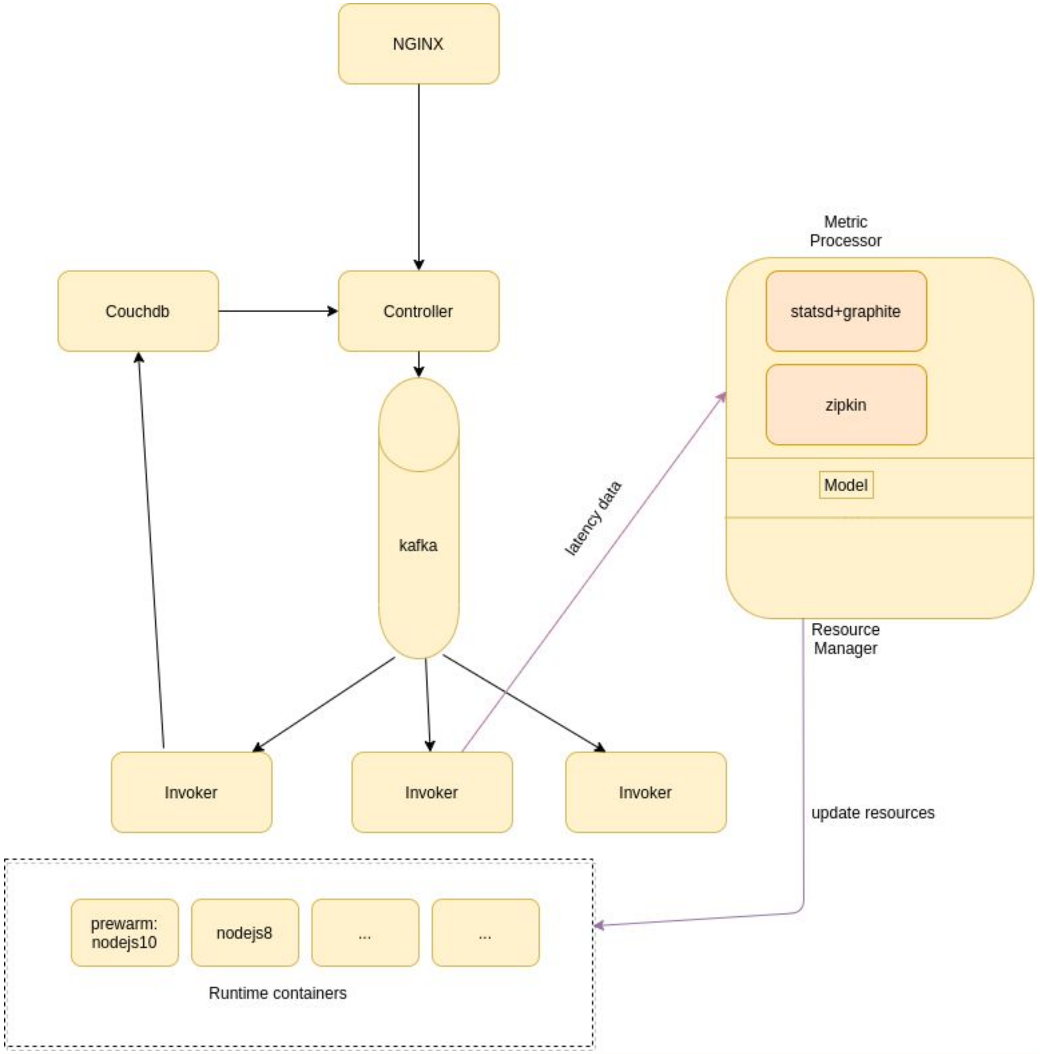
- Tries to ensure that the latency-sensitive job meets latency targets while maximizing the resources given to best-effort tasks.

- BUT, Solution is not for FaaS it is for tasks in server
  Also implementation details very close to hardware and uses other technologies like overclocking, cache isolation

Proposed Solution

- First benchmark each component of Apache OpenWhisk.
- A component will be responsible for monitoring resource metrics and one for allocating system resources.
- It will use some policy/heuristic to determine how to appropriately modify resource allocation to minimize QoS violations and/or optimize other parameters.
- Use cgroups to restrict resource allocation.

OpenWhisk System Architecture



Components of OpenWhisk

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.

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.

CouchDB

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

Components of OpenWhisk

Kafka

Apache Kafka is typically used for building pipeline 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.

Invoker

The Invoker tackles the final stage of the execution process. Based on the runtime requirements and the quota allocation, it spins up a new Docker 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 retrievals. 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.

Modules

Metric-processor:

- Reads the metrics from sources
- processes the data
- Feeds the processed data to a model

Model or Algorithm:

- Predict the action to be taken to ensure QoS

Resource-manager:

- Either directly changes resource allocation with docker client api or
- sends specific information to the controller to change the resource allocation to a component.

Design Constraints, Assumptions & Dependencies

Design constraints

- preferably use small amount of memory in runtime.
- must take small amount of time to calculate and update cpu-shares and not significantly increase the latency to start the specified function.

Assumptions:

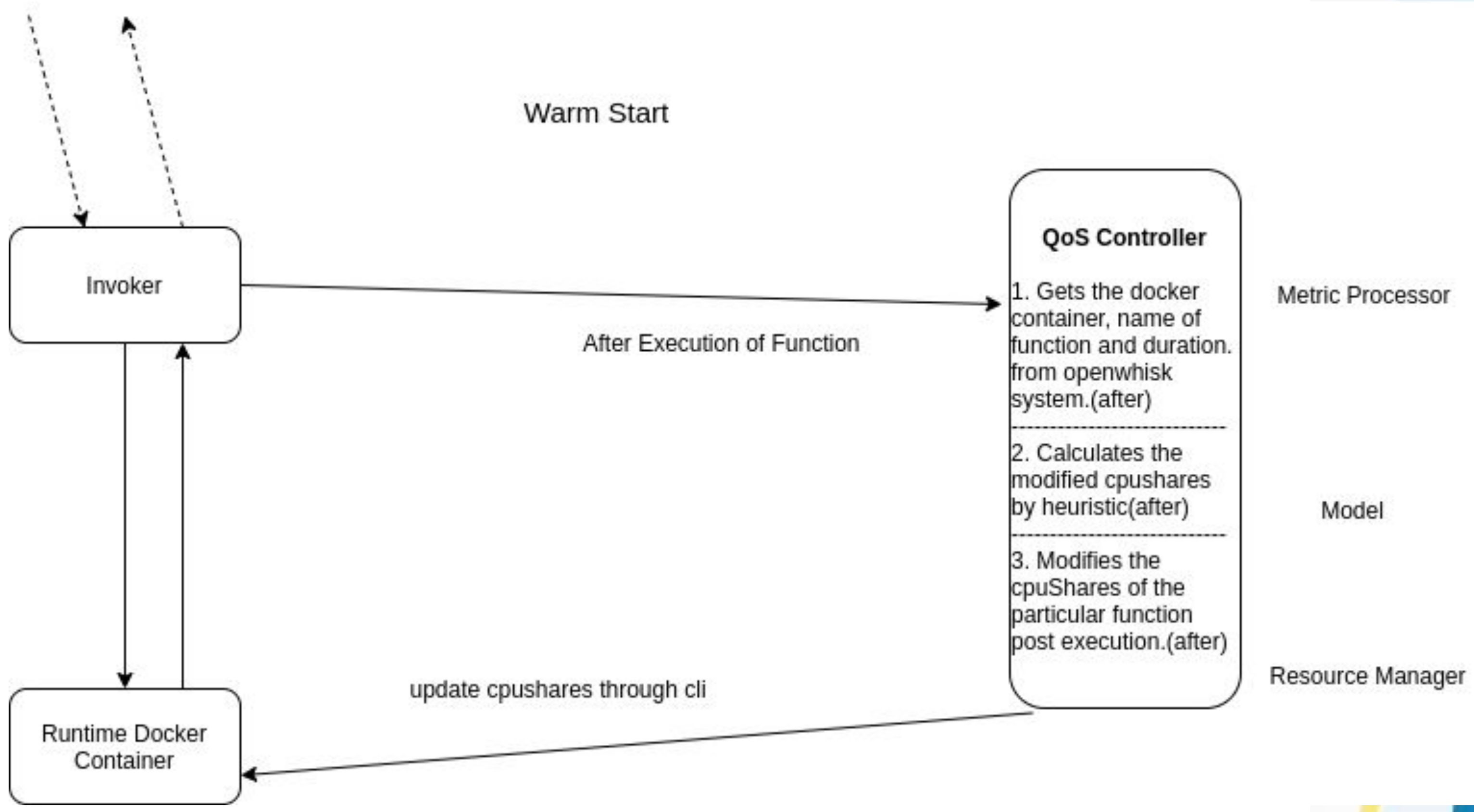
- Single machine setup for openwhisk
- Realistic workloads

Design Description

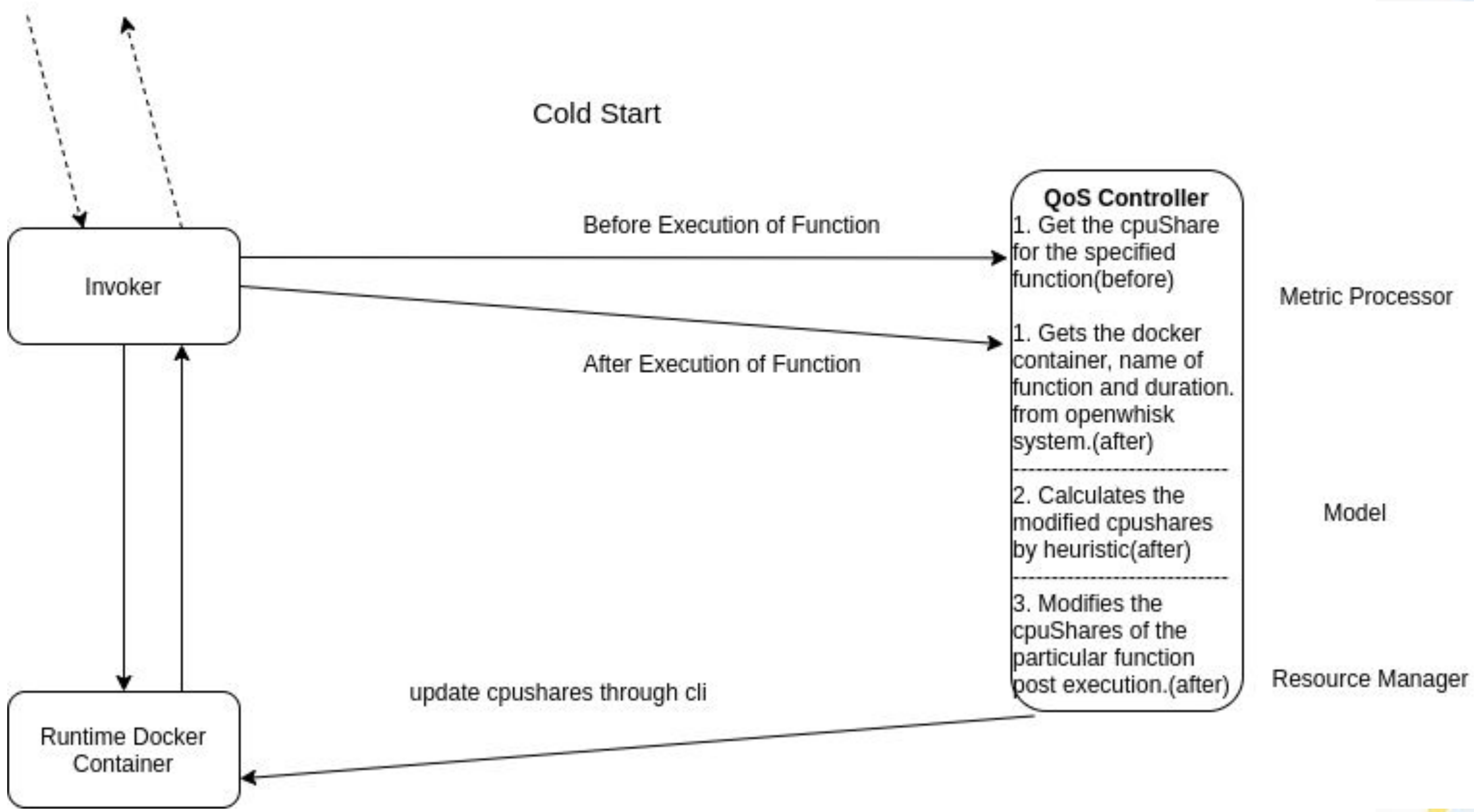
Cpu shares

- specifies the relative share of cpu time available to tasks in cgroup(docker container)
- QoS is proportional to cpu share
- It is single largest contributing factor

Design Description



Design Description



Implementation Details

Metric-processor:

- Technologies used: openzipkin, scala, python3, Flask server

Simple heuristic:

- Technologies used: python3
- each function starts of with 512 cpuShares

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

update cpuSharesofF for function F docker container
```

Resource-manager:

- Technologies used: python3, docker commands

Project Results

Results

Applications being run:

- **Function A:Name generation**:Uses the first letters from string to generate names with a rnn model.
- **Function B:Image resizing**: Pulls an image from the url and resizes it to the appropriate dimensions
- **Function C:Model training**: Trains the model to predict review sentiment scores of a text review from the amazon fine food review dataset.
- **Function D:pyaes benchmark**: performs private key encryption and decryption, implementation of AES block cipher algorithm

Applications are taken from serverless-faas-workbench and modified to work with openwhisk.

Base/Expected Latencies

- Latency of one invocation of the function without any other function invocations running

Actions	Expected latencies(msec)
Image Resizing	2000
Model training	24000
Name generation	2900
Pyaes benchmark	25000

Project Results

The 3 workloads that were run were:-

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

Project Results

Experiment 1(Workload 1):

- **Image resizing application at the rate of 60 requests per minute**

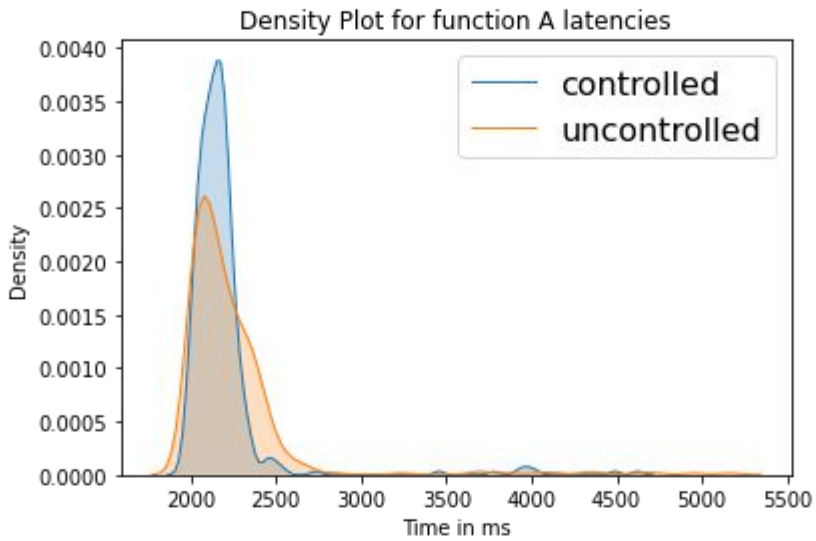
Project Results

	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

latency	controlled(msec)	uncontrolled(msec)
99	3959.23	4385.399
90	2277.0	2434.20
75	2207.0	2316.0

PES UNIVERSITY

Project Results



Project Results

Experiment 2(Workload 1):

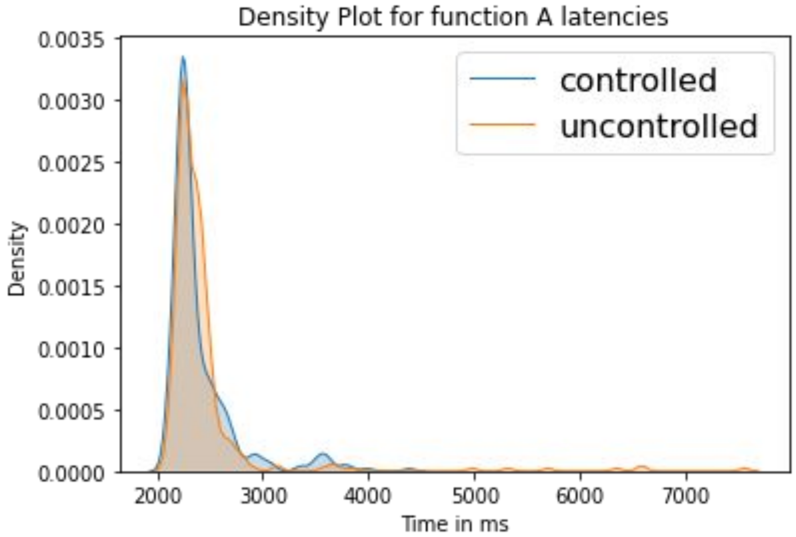
- **Image resizing application with a zipfian distribution.**

Project Results

	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

latency	controlled(msec)	uncontrolled(msec)
99	3766.21	5716.499
90	2696.1	2566.7
75	2447.75	2428.75

Project Results



Experiment 3(Workload 3):

- Model training application at the rate of 12 requests per minute with threshold
- Image resizing application at the rate of 60 requests per minute

Project Results

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

Project Results

Comparison of tail latencies between original platform and modified platform.

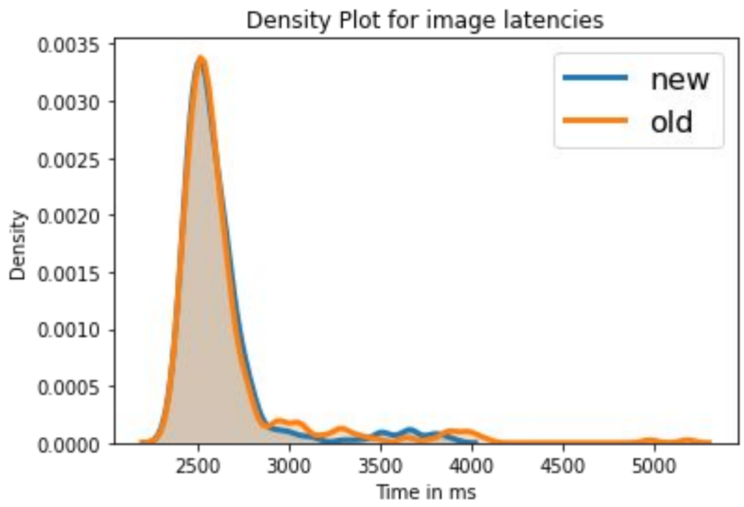
- image application latencies

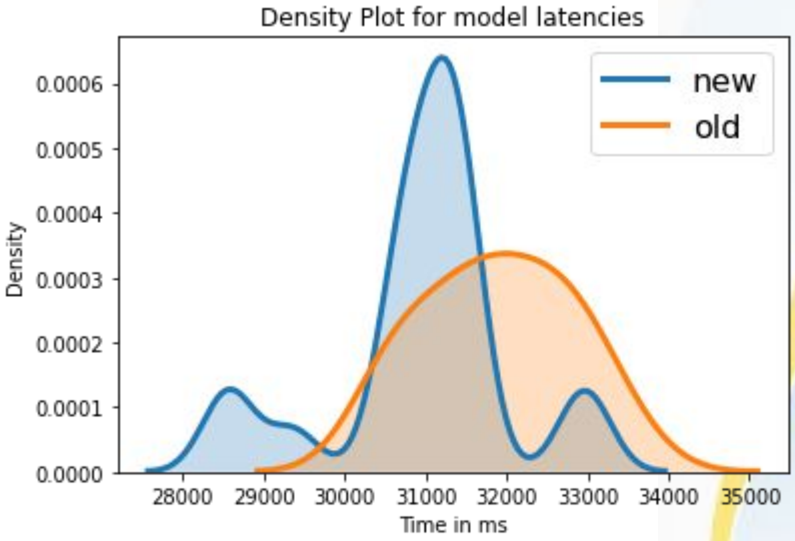
latency	controlled(msec)	uncontrolled(msec)
99	3771.07	3991.35
90	2774.7	2928.5
75	2647.25	2651.0

- model application latencies

latency	controlled(msec)	uncontrolled(msec)
99	33004.2	33430.14
90	31631.7	32939.1
75	31438.0	32602.75

Project Results





Experiment 4(Workload 4):

- Name generation application at the rate of 1.34 requests per minute
- Pyaes application at the rate of 30 requests per minute

- Run on both the original application and with modified application

Project Results

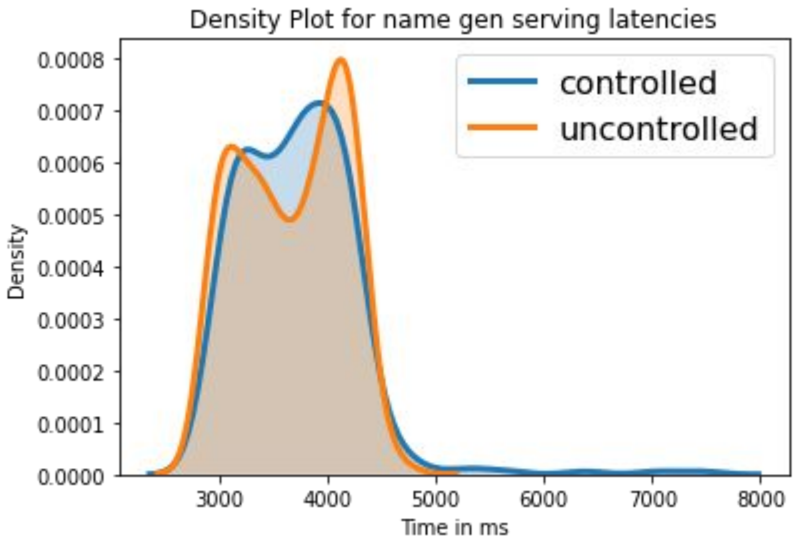
	controlled	uncontrolled
Total number of cold starts	22	21
Average time spent in kafka queue(micro seconds)	3988.05	3940.1
Total number of violations	184	188

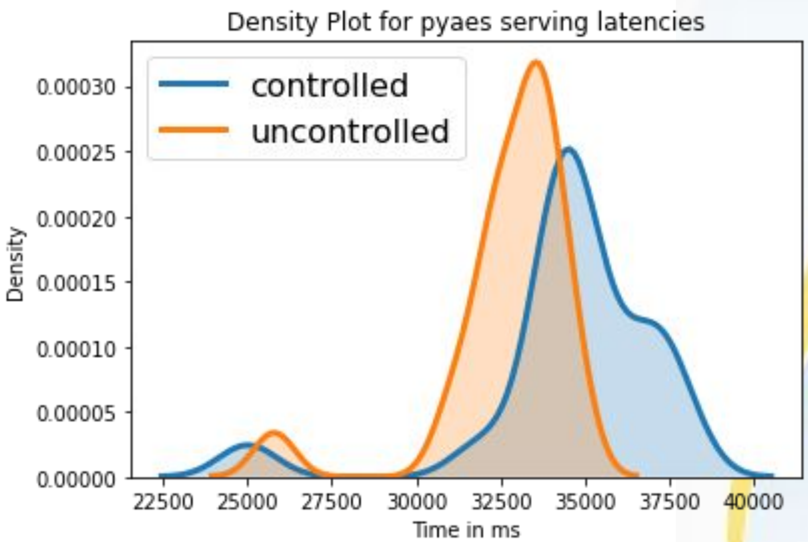
Project Results

latency	controlled(msec)	uncontrolled(msec)
99	5388.69	4458.519
90	4247.2	4249.8
75	4071.75	4119.75

latency	controlled(msec)	uncontrolled(msec)
99	37827.24	34671.20
90	37194.2	34175.79
75	36075.0	33768.5

Project Results





Project Results

Now same experiment is repeated but with delta of 20 for the Name generation function

Experiment 5:(Workload 4)

- **same characteristics are previous experiment but with delta for first function as 20**

Project Results

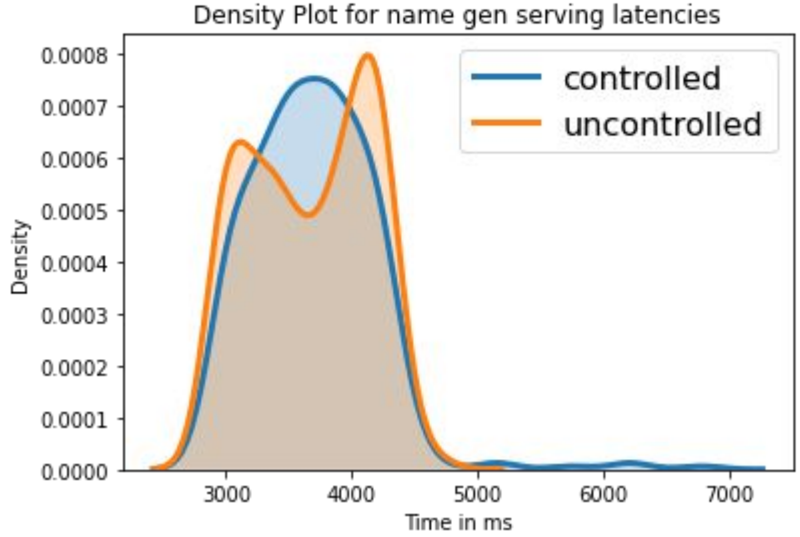
	controlled	uncontrolled
Total number of cold starts	23	21
Average time spent in kafka queue(micro seconds)	4139.84	3940.1
Total number of violations	173	188

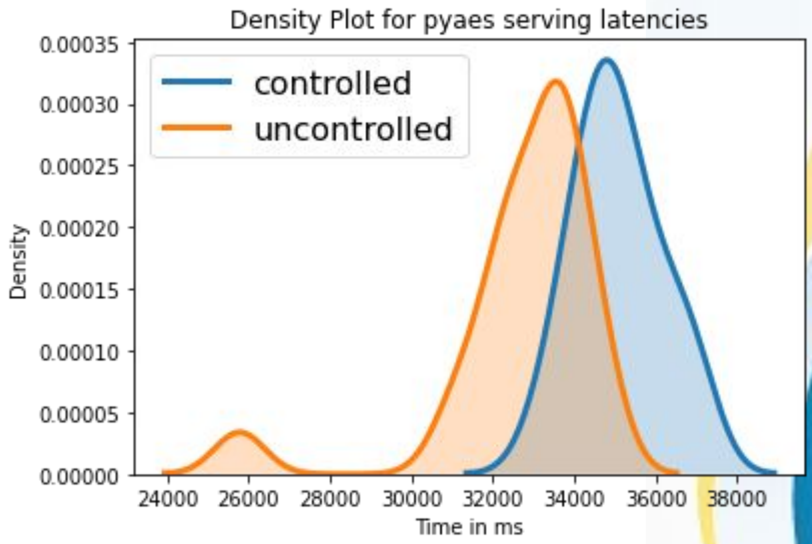
Project Results

latency	controlled(msec)	uncontrolled(msec)
99	5183.039	4458.519
90	4232.7	4249.8
75	4016.25	4119.75

latency	controlled(msec)	uncontrolled(msec)
99	37000.93	34671.20
90	36623.1	34175.79
75	35784.5	33768.5

Project Results





Conclusions

- **Adding a module in Apache OpenWhisk to ensure QoS definitely helps(look at density plots). For statistics like time spent in kafka queue, percentile latencies for most experiments there was a decrease**

- **In every single experiment there was a decrease in the total number of violations observed**

- **There is also the effect of increasing the delta value (CPU share updation value), which causes the total number of violations to drop in the last experiment.**

Project Results

Future Work

- Run on bigger machines to measure performance
- Use system metrics in augmentation with latency data and request data to ensure QoS.
- Figuring out the significance of dynamically provisioning system resources like extra VMs, extra memory to ensure QoS for serverless platforms
- 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.
- Figuring out quantitative significance of using request timeouts to improve the overall QoS of the functions.

Lessons Learnt

Any new technique that you learnt out of this project?

- Reading technical documentation
- Reading and modifying source code of large project.
- Interacting with open source community to ask for help

How has this project made your knowledge better?

- Lots of different technologies
- Learnt a bit of Scala programing language

Any issues that you faced?

- Scala code difficult to understand and modify
- Very very difficult to debug an function running on serverless platforms.
- Finalizing a workload to run.

Thank You

Planned Effort Vs Actual Effort

Week 1-2: Concretize thoughts, complete literature survey, Learn the language and Study of Code

Week 3: Design the architecture of solutions and other high level details, Learn the language and Study of Code

Week 4-8: Work on getting the solutions built after thinking about low level details.

Week 9-$: Testing the application(Took longer to run as delay in deciding workloads, setting up scripts to run experiments)

Start writing the report

PES UNIVERSITY

Technologies / Methodologies

Technologies used :

- Python3 - Flask framework to implement the Resource Manager
- OpenWhisk - open source serverless framework
- Scala(Akka Framework), Java - languages in which various components of OpenWhisk are written in
- Kamon, Graphite, Statsd, Zipkin libraries - tracing and monitoring for metrics
- Docker - containers that run the function invocation
- Ansible - For deployment of Openwhisk on server

Dependencies and Risks

- It is possible that under very high load beyond a limit, there is going to be latency degradation for end users.(hardware is finite)
- Assuming that cloud providers don't provide SLA for their serverless platforms(aws lambda has a SLA https://aws.amazon.com/lambda/sla/) - maybe they just solve it with throwing hardware.
- Implementation might be too hard.
- System developed might not be robust.

  Eg: Updating cpu-shares for a container has a high latency involved, metrics collected are not accurate. & https://gist.github.com/JPvRiel/bcc5b20aac0c9cce6eefa6b88c125e03

Why Your Solution is Better?

- Robust and simple heuristic to modify cpu shares.
- Robust to actions in OpenWhisk and other processes running in the system.
- Change the cpushares of runtime dockers containers to ensure QoS for different functions.