Mitigating Cold Start Problem in Cloud Computing

Ye Li
TCSS462-562
Research Paper Project

Overview

In serverless computing, when idle, the function does not occupy any resource, and the cloud provider will subsequently allocate the resource, which will be reclaimed after execution. In busy schedules, many requests to the function, and the serverless paradigm replicates the functions and runs them simultaneously, making it cost-effective and more straightforward. However, serverless computing suffers from a cold start problem, which involves a latency between the request arrival and function execution, affecting the response time and workflow. This culminates from the analogy that the function does not occupy any resources.

Apporaches

- Optimizing Cost of Serverless Computing through Function Fusion and Placement
- Improving Serverless Application Performance through Feedback-Driven Function Fusion through a FUSIONALIZE framework
- Eliminating Cold Startup in Serverless Computing with Inter-Action Container Sharing
- HotC: tackling the Cold Start of Serverless Applications by Efficient and Adaptive Container Runtime Reusing
- Using an application-level performance optimization approach called LambdaLite to accelerate the cold start for serverless applications

Comparison

Approach 1 and 2 have similarities in how FUSIONALIZE and Lambda lite operate in a way but differ in their framework despite linking to function fusion

LambdaLite and FUSIONALIZE are both different frameworks but perform the same functions

Conclusion

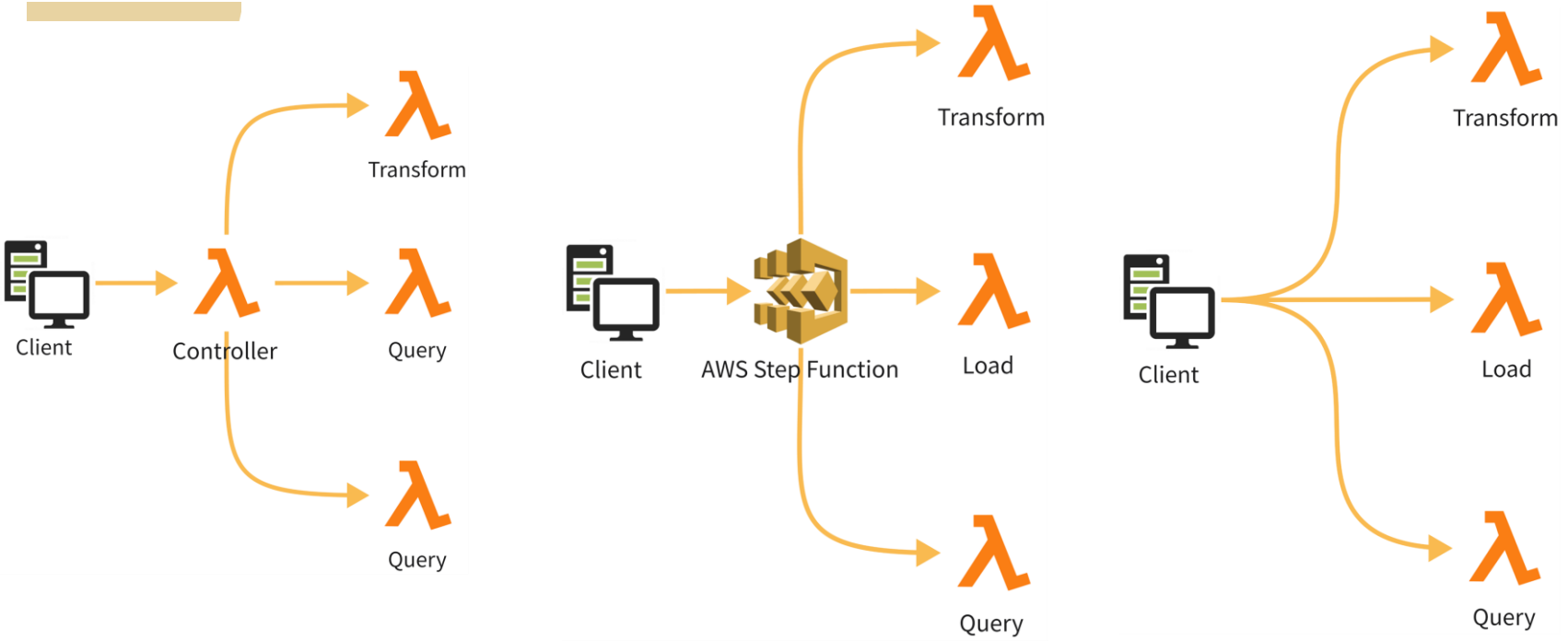
Costless transactions while mitigating cold startups have yet to be achieved, and this inspires research in the future for function placement and function fusion to execute costless transactions. There is a need to improve the total latency of serverless applications in the future.

TCSS 562A Term Project: Serverless Cloud Native Application

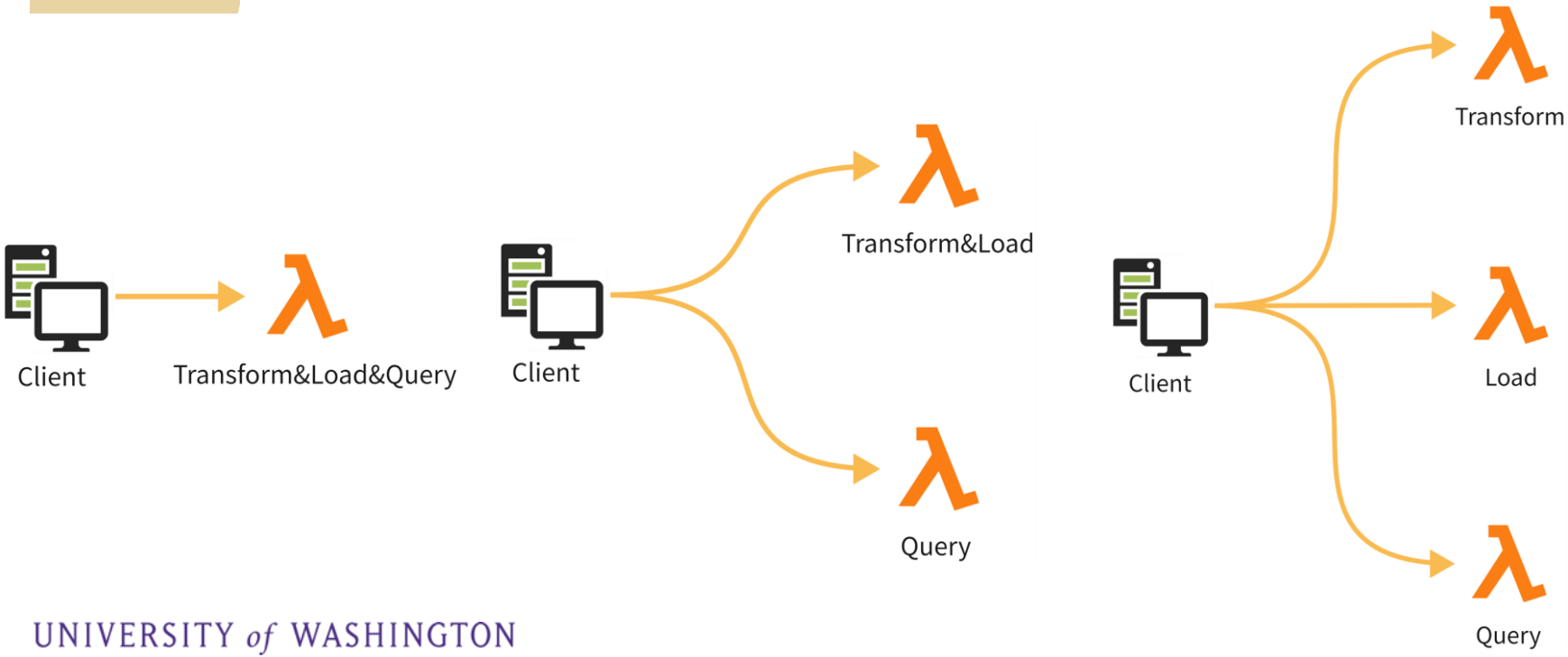
TCSS 562 Team 25:
Yuan Huang, Yifan Xie

UNIVERSITY of WASHINGTON

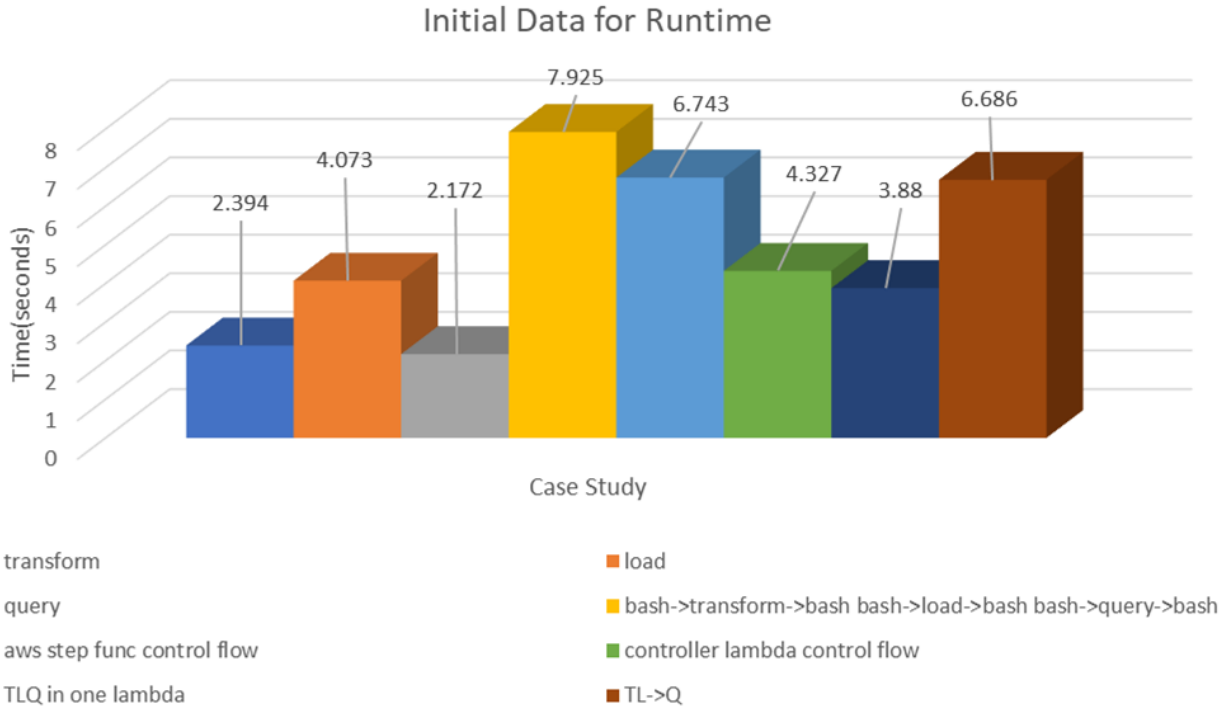
Case Study: Control Flow



Case Study: Composition



Initial Results



Thank you for watching!

UNIVERSITY of WASHINGTON

TERM PROJECT PRESENTATION

RamaSoumya Naraparaju
Sathwika Suddala

INTRODUCTION

The main aim of the project is to develop an TLQ data pipeline on AWS.Firstly, The data is extracted from csv file, transformed then loaded and at last query operations are performed on the data the deployed AWS Lambda functions,then testing and analysis of the opted metrics is done.

The main use case of the project is analysing a TLQ pipeline on different CPU processors and the performance variability based on their runtime, throughput and latency.

The languages used to implement the data pipeline are Java,SQL and Bash.The technologies and tools being used are AWS Lambda, S3 and SQLite.

METHODOLOGY

The code was tested on multiple lambda functions, and by changing the CPU architectures in lambda, the testing was sequential for the initial results.

Testing was done for different architectures, SAAF would be used further for latency, throughput metrics analysis.

And currently working on the 24-hour performance Variability for the analysis of turnaround time, throughput and network latency additionally will compare performance variability in different regions(Ohio vs Virginia).

RESULTS

The Graviton2 ARM64 processor had a 0.34% higher runtime than the Intel x86_64, when all the services were run together in the ohio us-east-2 region.

Individually the ARM64 processor was 16%, 6.04%, 2.07% faster than the Intel Xeon x86_64, for the ETL pipeline services respectively.

Services	Runtime for x86_64	Runtime for Graviton ARM64
Service 1 (Transform)	2.740s	2.351s
Service 2 (Load)	3.906s	3.670s
Service 3 (Query)	1.179s	1.148s
Combined	6.936s	6.960s

Gap Analysis: How Do Serverless Function-as-a-Service Platforms Mitigate Cold Start Latency

Team Number: 15

Team Members: Derek white

What is cold start latency?

- A cold start occurs when a new fresh container environment needs to be created by a cloud provider
- Cold start latency occurs when a serverless platform needs time to prepare resources
- Preparing resources includes – setting up dependencies and creating the runtime environment
- Preparation time leads to delays and slower response times

Why is it worth researching?

- In some cases cold start latency can be significantly longer than a function's run time
- Cold start latency negatively impacts the user experience
- Mitigation of cold start latency can improve response times
- Reducing cold start latency can enhance the user experience

Techniques For Mitigating Cold Start Latency

- Most strategies for mitigating cold start latency fall into two major categories
  1) Decrease the frequency of cold starts
  2) Reduce the preparation time of containers

Modern Research

- **Reinforcement Learning** – A machine learning technique that attempts to learn from usage patterns and tries to reduce the frequency of cold starts
- **Defuse** – A dependency-guided function scheduler that analyzes function invocation patterns. Defuse pre-warms containers based on common trends in usage
- **WLEC** – A container management architecture to reduce cold start time. Builds on an existing S2LRU model. WLEC creates three queues to sort containers by specific metrics
- **Pause Container Pool Manager (PCPM)** – Uses a pause container pool management system to pre-create networks to attach to new containers. The use of pre-created networks mitigates a major bottleneck in container creation

Summary

- Researchers use different methods to test their mitigation strategies
- Researchers use different metrics to measure their results
- The lack of consistency can make comparing strategies challenging
- Some researchers don't test their strategies on live services
- Mitigation strategies have pros and cons
- Fewer cold starts and reduced startup time can lead to increased memory usage

Mitigation Technique	Applied to Open Source Platform	Applied to Commercial Platform	Reduce Startup Time	Reduce Cold Start Frequency	Increase In Memory Usage	Improvement Over Baseline*
Reinforcement Learning	✓	✗	✗	✓	-	✗
Defuse	✗	✗	✗	✓	✗	✓
WLEC	✓	✗	✓	✓	✓	✓
PCPM	✓	✓	✓	✗	✓	✓

*Baseline represents the standard set by existing methods defined by the researchers.

*Baseline may be the number of cold starts or container startup time.

Java vs JavaScript for Serverless Image Processing Pipelines

Team 12: KV Le, Codi Chun, Duy Vu, Carlos Alberto Manrique Ucharico

GitHub Repo: https://github.com/kvietcong/tcss462-project/

Application:

Image Processing

Our Case Studies:

Programming Languages
Hot vs Cold performance

Technologies:

- AWS Lambda: Our code is uploaded here to be run on a function call.
- AWS S3: Our images we manipulate are put here before calling functions and after the processing is done.
- JIMP: Our JavaScript library used to load the image that has no Native Code (Pure JS). (https://www.npmjs.com/package/jimp)

Our group implemented a simple **Image Processing Pipeline** on **AWS Lambda**. It **supports** the following **operations**:

- Greyscale
- Soften (Blur)
- Flip (Vertical and Horizontal)

Our case study was **implementing** our app with **Java** and **JavaScript**. We took best efforts to **maintain similar logic** across the versions to center differences around language rather than implementation.

- To maintain similarity we focused our processing around **manipulating individual pixels** and their RGBA values.

We're also looking into how **"Hot"** and **"Cold" AWS Lambda function calls** can differ in performance and if it's different across languages.

Testing Approach

FaaS runner and Bash Scripts

Our process for testing was quite simple.

- We uploaded a few images to our S3 bucket.
- Prepared **identical SaaF experiment** files for each language to go through **each filter multiple times** for **every image**.
- Collected the **JSON outputs** from the runs and compile them in a **Jupyter Notebook** for interactive processing.

For our **SaaF experiments**, we decided to use many runs and threads, meaning our experiments ran **concurrently.**

- To **simultaneously** get **Hot** and **Cold** calls, we called each experiment multiple times from a **BASH script** with very long **sleep times** between them.

mountains.jpg
565.6KB
3840x2160

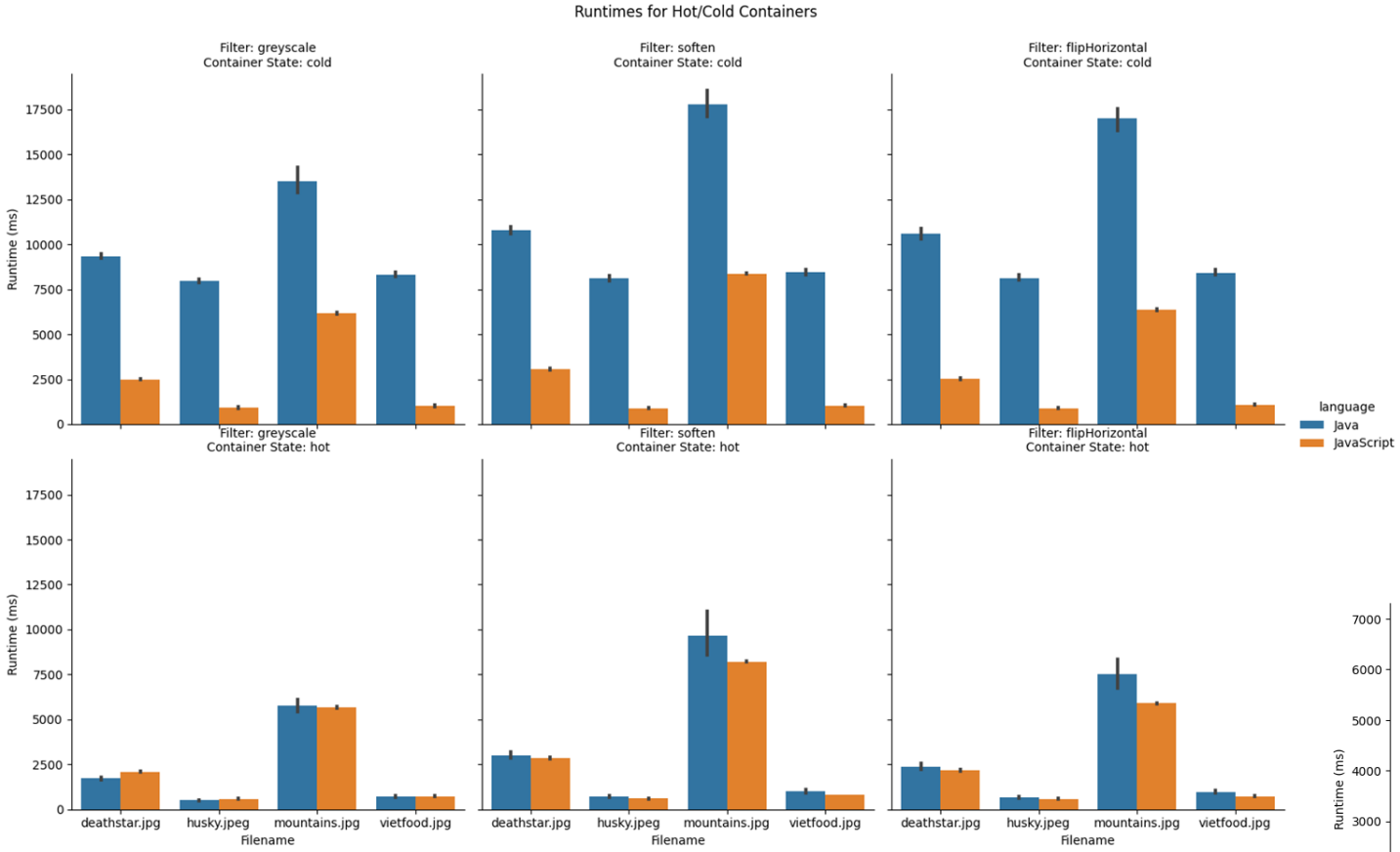
husky.jpeg
16.7KB
256x256

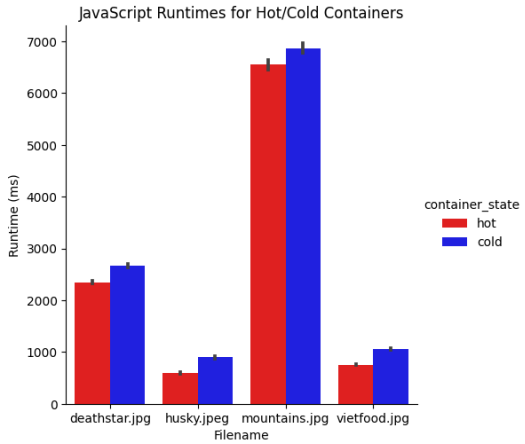
Preliminary Results

Images here are the ones used in our experiments

deathstar.jpg
956.7KB
1920x1200

vietfood.jpg
163KB
509x339





In our preliminary results above we noted the following:

- Java cold starts are brutal. [↑]
- JavaScript and Java performances for hot calls are almost equivalent in runtime. [↑]
- Cold starts are not that impactful for JavaScript but there is a slight difference. [→]

Story Generation Pipeline

Team 7: Anthony Carrillo, Elijah Reyes, Roland Hanson, Showmik Roy

Overview

Use Case

- Our project uses AWS Lambda to implement FaaS and generate a small story using a Markov model, given a text input.

Case Studies

- Comparing performance differences between the x86_64 and arm64 architectures

Implementation

- Our implementation used Python, AWS Lambda, AWS Rest API Gateways, and Bash

Testing

Client

- Locally hosted Ubuntu VM using a two sequential curl Bash script per test
- Communicated via SAAF function handlers and JSON to AWS Lambda

Uniqueness

- Performed using a live Lambda function and using a 6KB portion of a Sherlock Holmes story as input. Called on hot functions.

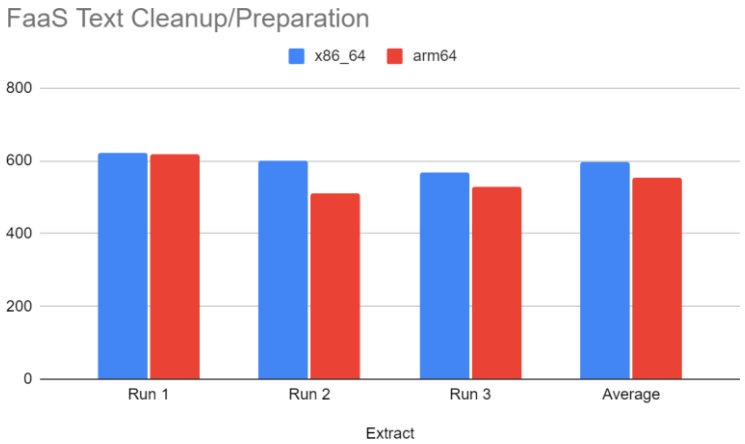
Performance Comparison

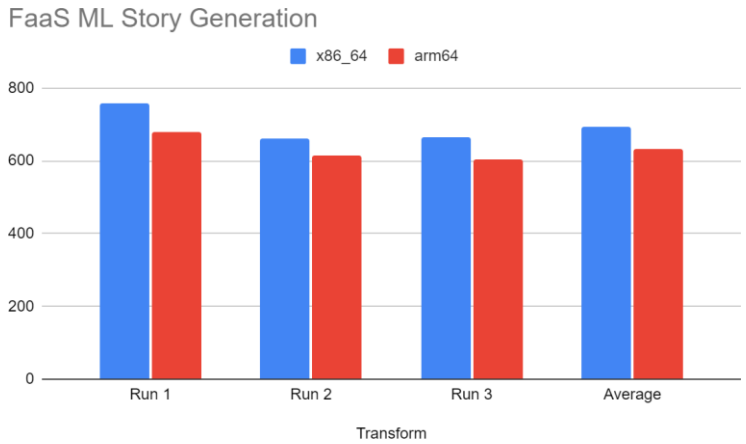
Testing runtime between arm64 and x86_64 architectures (in ms)

	Extract	Transform		Extract	Transform		Extract	Transform		Avg runtime Extract	Avg runtime Transform
x86_64	622	759		599	662		567	664		596	695
arm64	619	680		512	614		527	605		552.6666667	633

Each lambda function was run three times and times were reported from the JSON results
Using the results from the three tests, an average runtime was calculated

Performance Comparison (cont.)





We found that on average **arm64** was **7.54% faster** on the **extract** function and **9.34% faster** on the **transform** function. Additional data will be gathered on larger input.

A FaaS Comparison of Amazon Web Services and Google Cloud Platform using an Image Process Pipeline

TEAM 1:
Jeff Stockman
Rick Morrow
Austin Carter Luu
Mahmoud Elkamhawy

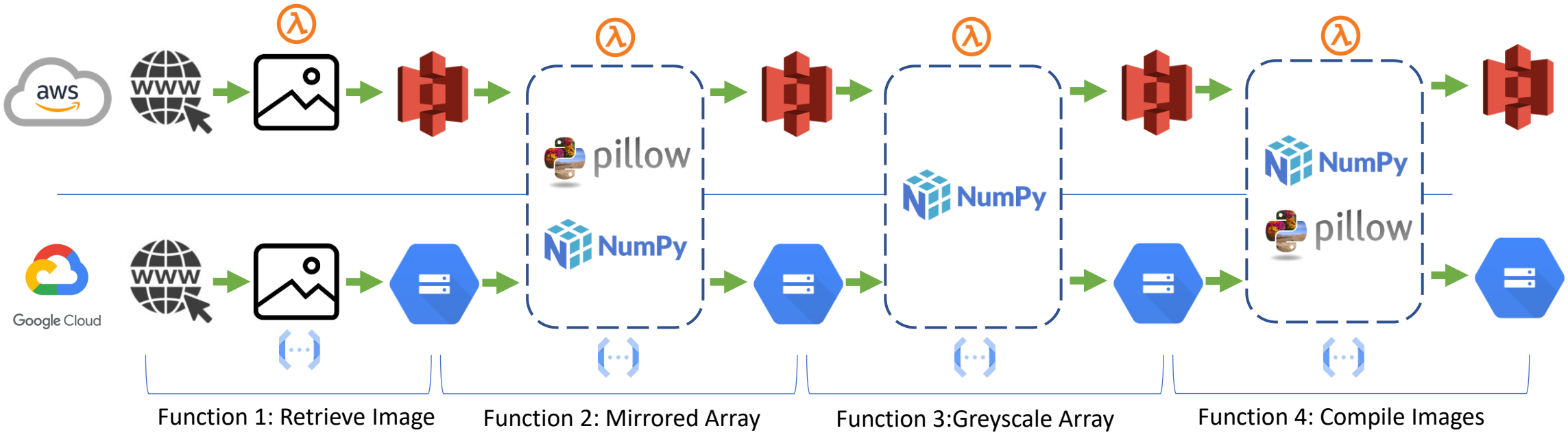
Overview

Use case: Compare performance, accuracy, and cost variations across AWS and GCP FaaS and storage services

Implementation:

Python 3.09; associated AWS/GCP libraries; Pillow & Numpy libraries for image processing

- Function 1: uploads an image from a URL into input storage bucket
- Function 2: retrieves stored image, creates mirrored array & difference array, stores to input bucket
- Function 3: retrieves mirrored array, creates greyscale array & difference array, stores to input bucket
- Function 4: retrieves greyscale/greyscale_difference array & mirrored_difference array, constructs [3] image: greyscale, mirrored, & original image



Testing Approach

Pipeline Composition:

- Asynchronous, sequential pipeline
  - Single Trigger starting with first function
- O(n)^3 Time Complexity meant to be relatively compute and memory-heavy
  - Long-running pipeline will promote performance differences between AWS & GCP

Experimental Design

- Varying Image Size
  - Promotes degrees of performance difference based on computing requirements
- Single Pipeline Deployment
  - One image processed
- Multi-concurrent Pipeline Deployment
  - Up to 500 concurrent images

Evaluation Criteria

- Runtime
  - Average pipeline runtime
  - Average pipeline runtime at scale
- Cost (FaaS + storage)
- Failure rates (e.g. timeouts)

Performance Comparison

AWS:

- Initial Test Runtime: ~10 minutes
- All functions processed in memory; O(n)^3 time complexity
- Function 3 maxes out memory (128MB) at 100%; suspected speed degradation based on collisions and disk caching
- 20% speed improvement using 512MB (Function 3 only)

GCP: Not fully implemented yet

AWS Lambda TLQ (Transform, Load, Query) Data Pipeline

Team 10
Nehaa Vuppala
Chhavi Gupta
Divya Jacob
Nandhini Dhanasekaran

Project Overview

Implementing TLQ pipeline as a set of independent AWS Lambda services and analyzing the performance of the services using different case studies.

Case Studies

- Alternate CPU Architecture

  To analyze the performance of the services using ARM 64 and X86 64 Architecture

- Switchboard Architecture

  Minimizing the number of deployment packages by bundling all source code together into a single Lambda function to check the overall cost and performance

- Performance Variability

  To analyze the performance by measuring the end to end turnaround time of the pipeline in multiple AWS regions.

Language : Java, Shell Script

Technologies: AWS Lambda, S3, SQlite, JSON

Testing Approach

- Developed and tested individual services on different systems
- Merged all 3 services on one environment and tested if all of them working properly together
- Ran the services multiple times by changing the architecture in the same environment
- Recorded the timestamp for each run and compared them with each other

Performance comparison

We used lambda functions and and conducted a case study “CPU Architecture” in which we ran all 3 services multiple times with each run having a different architecture . The goal of the case study was to record timestamp of individual runs & use these values to compare the performance .

TLQ	X86_64	arm64
Service 1(r Transform phase)	4.917s	1.447s
Service 2(Load phase)	0.049s	0.005s
Service 3(Query Phase)	0.01s	0.017s

Java vs JavaScript for Serverless Image Processing Pipelines

Team 12: KV Le, Codi Chun, Duy Vu, Carlos Alberto Manrique Ucharico

GitHub Repo: https://github.com/kvietcong/tcss462-project/

Application:

Image Processing

Our Case Studies:

Programming Languages
Hot vs Cold performance

Technologies:

- AWS Lambda: Our code is uploaded here to be run on a function call.
- AWS S3: Our images we manipulate are put here before calling functions and after the processing is done.
- JIMP: Our JavaScript library used to load the image that has no Native Code (Pure JS). (https://www.npmjs.com/package/jimp)

Our group implemented a simple **Image Processing Pipeline** on **AWS Lambda**. It **supports** the following **operations**:

- Greyscale
- Soften (Blur)
- Flip (Vertical and Horizontal)

Our case study was **implementing** our app with **Java** and **JavaScript**. We took best efforts to **maintain similar logic** across the versions to center differences around language rather than implementation.

- To maintain similarity we focused our processing around **manipulating individual pixels** and their RGBA values.

We're also looking into how **"Hot"** and **"Cold" AWS Lambda function calls** can differ in performance and if it's different across languages.

Testing Approach

FaaS runner and Bash Scripts

Our process for testing was quite simple.

- We uploaded a few images to our S3 bucket.
- Prepared **identical SaaF experiment** files for each language to go through **each filter multiple times** for **every image**.
- Collected the **JSON outputs** from the runs and compile them in a **Jupyter Notebook** for interactive processing.

For our **SaaF experiments**, we decided to use many runs and threads, meaning our experiments ran **concurrently.**

- To **simultaneously** get **Hot** and **Cold** calls, we called each experiment multiple times from a **BASH script** with very long **sleep times** between them.

mountains.jpg
565.6KB
3840x2160

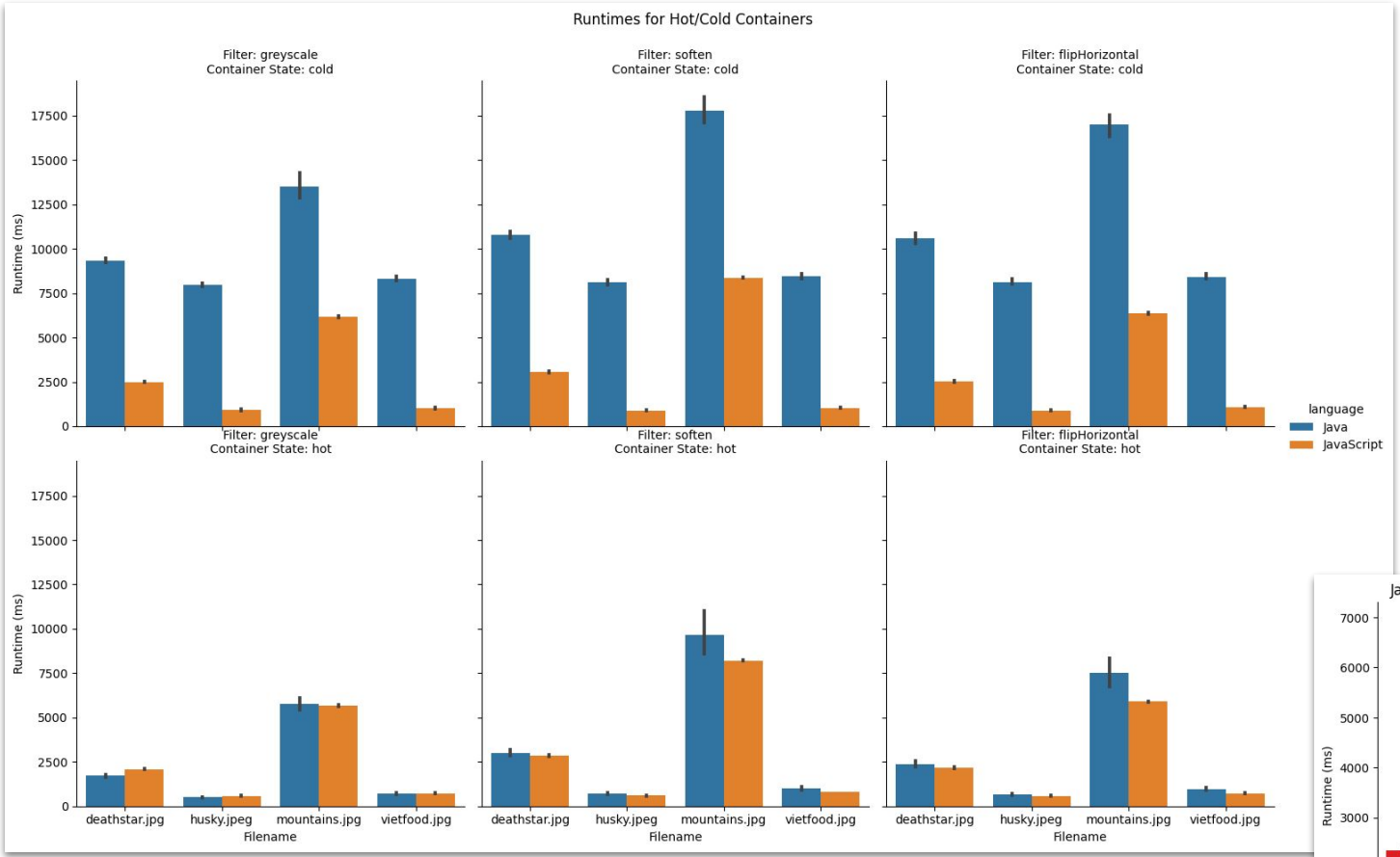
husky.jpeg
16.7KB
256x256

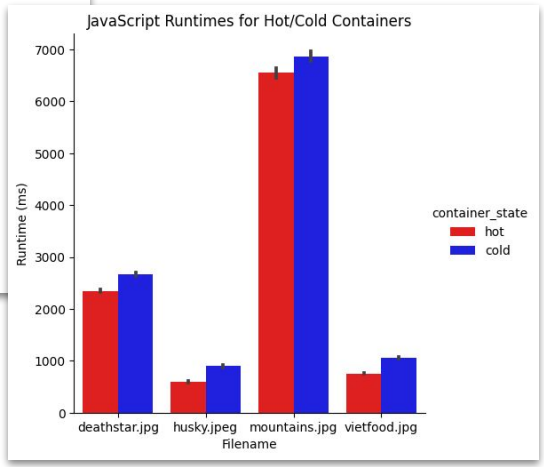
Preliminary Results

Images here are the ones used in our experiments

deathstar.jpg
956.7KB
1920x1200

vietfood.jpg
163KB
509x339





In our preliminary results above we noted the following:

- Java cold starts are brutal. ⬆️
- JavaScript and Java performances for hot calls are almost equivalent in runtime. ⬆️
- Cold starts are not that impactful for JavaScript but there is a slight difference. ➡️

IMAGE PROCESSING PIPELINE

Gurleen Grewal, Nicole Guobadia, Tony Le

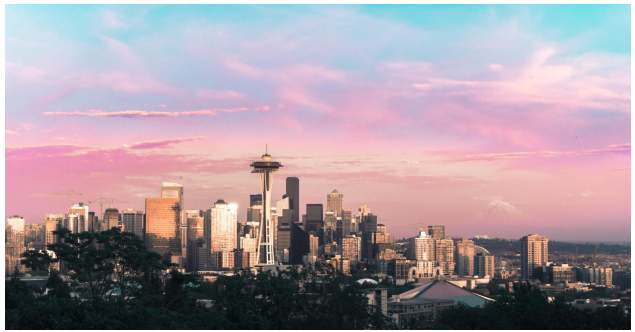
Team 19

PROJECT OVERVIEW

- Analyze the performance variation of serverless image processing functions
  - Grayscale, Soften & Mirror
- Store image in aws S3
- Process image using aws Lambda functions
- Record average workflow round trip time, cold pipeline and warm pipeline performance
- Implement project in Java

TEST APPROACH

- Run 24 hours of sequential image processing calls
- Experiment in 5 regions
  - us-west-2
  - af-south-1
  - ap-northeast-2
  - eu-west-2
  - me-central-1
- Process a singular 115kB jpeg image
- Modify SAAF to handle image processing



INITIAL COLD START RESULTS

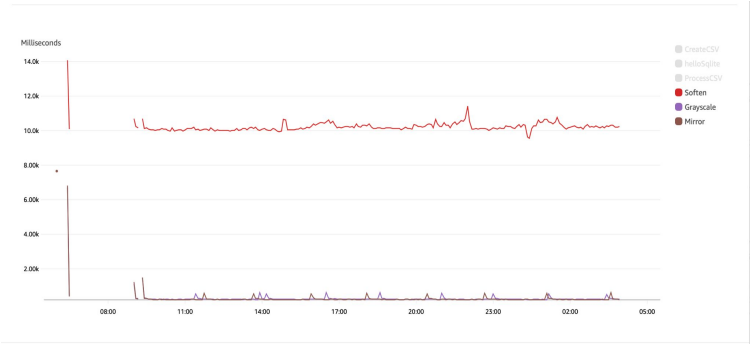


Figure 1 : Average duration in us-west-2

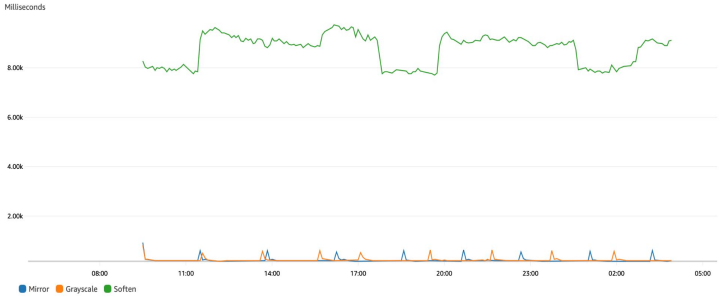


Figure 2 : Average duration in af-south-1

INITIAL COLD START RESULTS

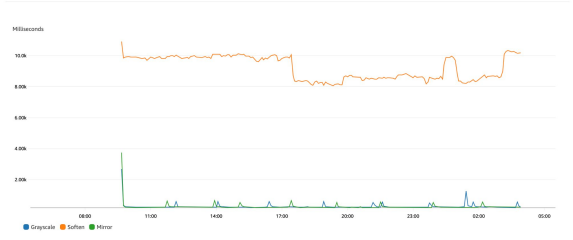


Figure 3 : Average duration in ap-northeast-2

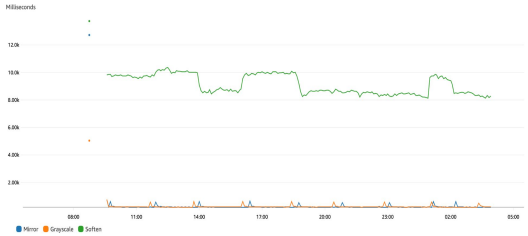


Figure 4 : Average duration in eu-west-2

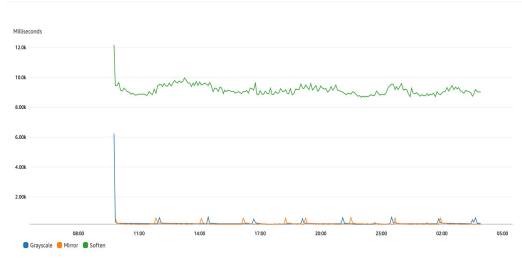


Figure 5 : Average duration in me-central-1

UNIVERSITY of WASHINGTON

TCSS 562 Software Engineering for Cloud Computing

AWS Lambda TLQ Data Pipeline -
Performance Comparison of CPU Architectures across Three Availability Zones in Cold/Warm Start

GROUP 20

Team Members: Angela Mu, Xiaojie Li, Ruigeng Zhang, Yihan Ma

Use Case

Transform-Load-Query data processing pipeline

Case studies:

- 1. Architecture of Intel vs Arm
- 2. Availability zones: us-east-2, us-east-1, ap-east-1
- 3. Freeze-thaw infrastructure lifecycle: cold vs warm

Implementation:

- Language: Java (JDK 11)
- Tools/Technologies: AWS Lambda, Amazon RDS for MySQL, Amazon S3, AWS Step Functions, Amazon EC2

Testing Approaches

- We are using sequential client to test the application.
- HW: On step functions.
- We are using step functions to run the LTQ data pipeline application instead bash script on local computer or ec2 instance.
- SAAF is employed as lambda function which is called by step function.

Performance Comparison (Initial Data)

> Cold v.s. Warm Function (x86-64, us-east-2)

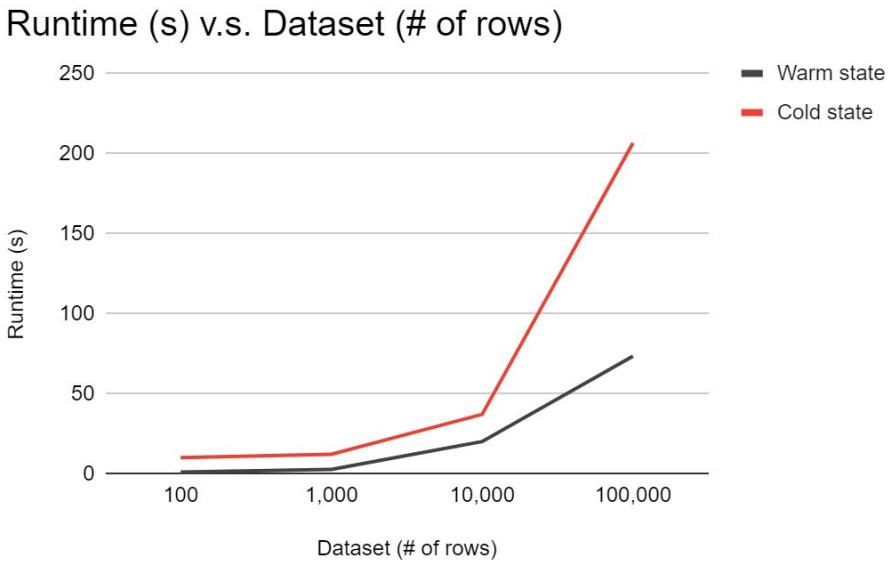
dataset (# of rows)	runtime (s)	throughput (# of rows/s)
100	10.141	9.860960458
1,000	12.23	81.76614881
10,000	37.14	269.2514809
100,000	206.575	484.0856832

Cold Start

dataset (# of rows)	runtime (s)	throughput (# of rows/s)
100	1.127	88.73114463
1,000	2.806	356.3791875
10,000	20.214	494.706639
100,000	73.467	1361.155349

Warm Start

Performance Comparison (Initial Data)



TLQ Pipeline on AWS and GCP

Group 5 - Tsung-Jui Wang, Jinming Yu, Sue Yang, Yafei Li

TLQ pipeline

- Our application is the assigned topic: TLQ pipeline.
- Our case study is the comparison between two different serverless platforms.
  - The two platforms are AWS Lambda and Google Cloud Functions.
  - Both platforms run the same Java code to do the pipeline except some dependent settings for each platform.
  - Both platforms are in us-west(Oregon) region.
- Service 1: Transform the CSV file and upload the result to Cloud storage.
  - S3 bucket for AWS and Cloud Storage for GCP.
- Service 2: Load the CSV to Cloud databases.
  - Aurora MySQL for AWS and Cloud SQL for Google Could.

TLQ pipeline

- Service 3: Filters and aggregations.
  - Inputting 2 filters and get a column output by using WHERE clause in AWS Lambda and Google Cloud Functions.
  - Support 9 types of aggregations, group by by Region, Item Type, Sales Channel, Order Priority and Country, clausing on AWS Lambda and Google Cloud Functions.

Testing Approaches

Testing approaches

- FaaS Runner(https://github.com/wlloyduw/SAAF/tree/master/test)

Laptop

- VM in Ubuntu on Macbook Pro M1 Pro

Using parallel testing.

Performance Comparison

- Service 3 in AWS:
  - Run 50 instances in parallel, using Faas Runner.
  - Average latency: 10065.19 ms
  - Average runtime: 2383.29 ms
  - Average round trip time: 12448.48 ms
- Others in progress …

Programming language project proposal

Use case: Determining the performance of a language's pipeline over a period of time.

Case Study: Java TLQ performance over a 24 hour period.

Details: A Java TLQ pipeline was used to deploy to AWS lambda functions with s3 buckets for data. The criteria were average round trip time performance, warm function performance, and throughput measured in rows of data processed per second.

Testing approaches

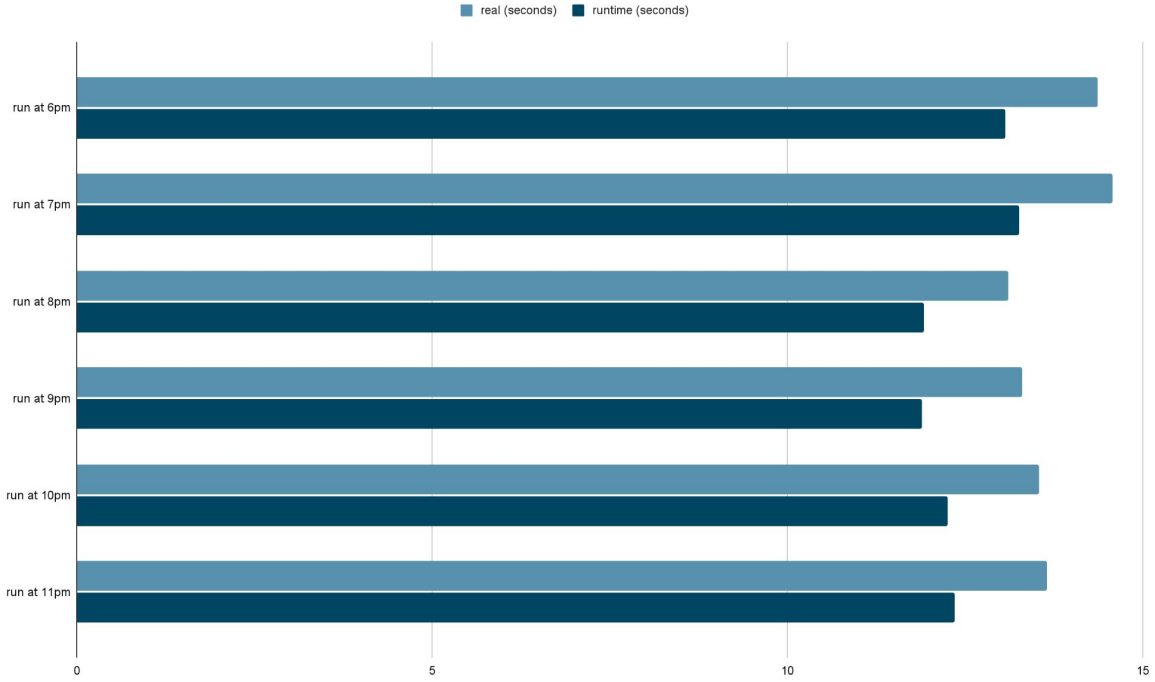
Client type: We will be utilizing a sequential client called from a laptop to lambda for our function calls using s3 buckets for data.

Unique aspect: 24 test runs potentially with regional testing or greater than 24 hour test runs.

SAAF employment: We utilize AWS lambda with API Gateways and s3 buckets for data in the TLQ pipeline.

Initial performance results for a 5 hour period

For our initial test run over a smaller period, the runtime was marginally improved and found its best time at 8pm.



Cloud Performance Variation over Time for Different Availability Zones: TLQ Pipelines

Team Members: Kannika, Jessie, Bao, Stephanie

Overall Project

Use Case is TLQ pipeline

- There are 3 services; Transform data, Load data, and Query data

Case Study

- Perform the TLQ Pipelines with three services; transform, load and query.
- Run 3 services in the different times such as morning (6am), afternoon (12pm), evening (6pm), night (midnight)
- Run 3 services in the different available timezone/area.

Tools and Technologies used

Language

- Java and MySQL

Tools

- Apache Netbeans IDE 11.1, IntelliJ 2.3

Cloud Service

- AWS lambda
- AWS EC2(t2.micro)
- AWS S3
- Amazon RDS Aurora 5.6 database

Testing approaches

- **Running three services separately in the different times.**
- **Running three services separately in the different area.**
- **Analyze the cloud performance variation over time for different availability zones using the TLQ Pipelines with three services; transform, load and query.**
- **Implement an identical processing pipeline with two different backends - cold and warm functions**

Transform Service

This table shows the results of Transform Service testing, running in the us-east-2 area (Ohio), at 5 - 7 am. And the function get "errorMessage": "Java heap space" when we run the 500,000 rows of data.

	100 rows	1000 rows	10,000 rows	50,000 rows	100,000 rows
Runtime (ms)	13,556	14,503	19,359	24,283	27,840
User Runtime (ms)	13,378	14,319	19,218	24,061	27,695
Latency (ms)	178	184	141	222	145

Load Service

This table shows the results of Load Service testing, running in the us-east-2 area (Ohio), at 5 - 7 am. Since, we use the data from the Transform Service, so largest data that we can run with the Load service is 100,000 rows.

	100 rows	1,000 rows	10,000 rows	50,000 rows	100,000 rows
Runtime (ms)	11,978	11,617	11,921	12,218	143,585
User Runtime (ms)	11,822	11,459	11,780	12,019	143,425
Latency (ms)	156	158	141	199	160

Query Service

This table shows the results of Query Service testing, running in the us-east-2 area (Ohio), at 5 - 7 am. Since, we use the data from the Load Service, so largest data that we can run with the Query service is 100,000 rows.

	100 rows	1,000 rows	10,000 rows	50,000 rows	100,000 rows
Runtime (ms)	13,578	14,516	15,103	17,865	21,452
User Runtime (ms)	13,432	14,392	14,888	17,738	21,279
Latency (ms)	146	124	215	127	173

Team 2

Image Processing Lambda vs. Cloud Functions

Jared Pines
Dylan Churchward
Oleg Uvarov
Andrew Moreno-Escareno

Lambda functions vs. Cloud Functions

Use case: Create two image processing pipelines in the cloud, one using AWS lambda functions and S3 and the other using Google Cloud functions and Cloud storage. Both pipelines utilize python and publicly available python image processing libraries, and are made to be as similar as possible during execution

Case study: Analyze the overall runtime to process a single image on both pipelines to determine which pipeline is more efficient time wise. We collect the runtime to process the same image 1000 times (ignoring the cold start data) on each pipeline to compares the average runtimes, standard deviations and coefficients of variation

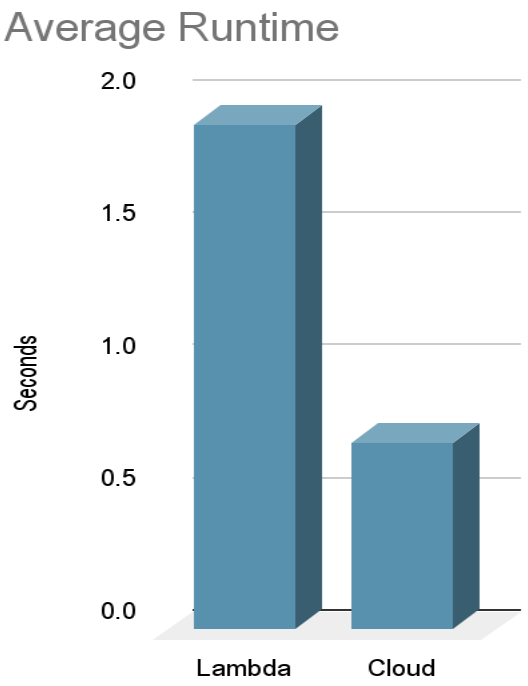
Testing Approach

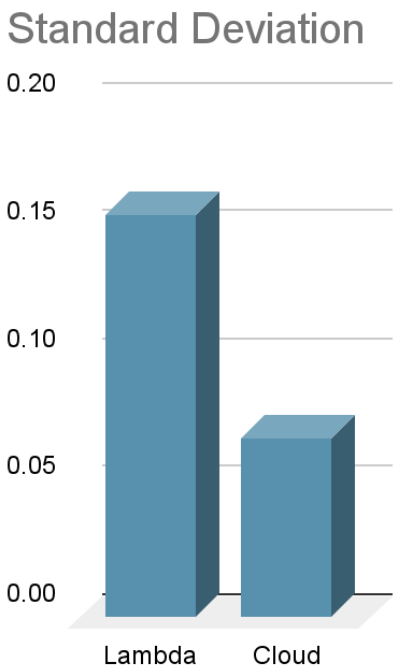
We made 1000 sequential calls to each image processing pipeline using the same image and the same image manipulation parameters, and collected the runtime of each call

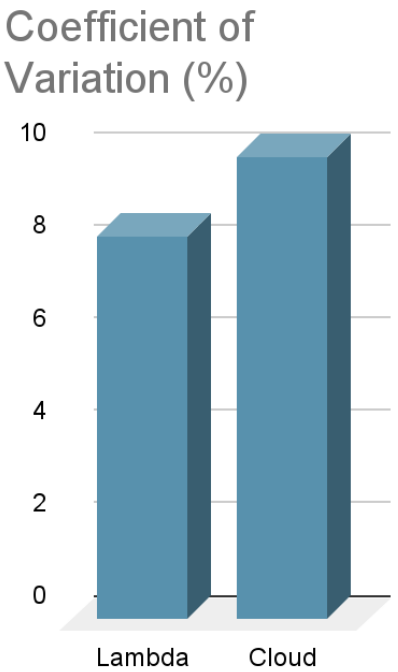
We then use this runtime information to determine the average runtimes, standard deviation, and coefficient of variation of each pipeline

Using our analysis of the runtimes, we can determine which pipeline is faster and more consistent for this specific use case

Results


Average Runtime
Seconds
2.0
1.5
1.0
0.5
0.0
Lambda
Cloud


Standard Deviation
0.20
0.15
0.10
0.05
0.00
Lambda
Cloud


Coefficient of Variation (%)
10
8
6
4
2
0
Lambda
Cloud

Conclusions

For this use case, Google Cloud functions appear to have a much faster average runtime than AWS Lambda functions. So, if you want to process images with FaaS functions, Google is the way to go! The next comparison to do would be the costs of all of the services.

Additionally, while working on our AWS functions we discovered that reading the images from S3 as a byte stream, processing the images in memory (no downloading to /tmp) and writing as a byte stream back to S3 can be about 0.7 seconds faster than reading the images as “files”, saving them to disk and processing them that way. So processing images in memory can greatly reduce I/O and increase speeds.