Comparing FaaS TLQ Pipeline Performance Based on CPU Architecture and Cold/Hot State

Comparing Implementation Decisions Using AWS

Patrick Hern
UW Tacoma
Tacoma, WA USA
phern@uw.edu

Cameron Gregoire
UW Tacoma
Tacoma, WA USA
cgrego2@uw.edu

Angela Farin
UW Tacoma
Tacoma, WA USA
angelaf7@uw.edu

Tomoki Kusunoki
UW Tacoma
Tacoma, WA USA
tkondo2@uw.edu

ABSTRACT

This project explores the optimization of data processing workflows within the TLQ paradigm (Transformation, Loading, and Querying) in a serverless cloud environment. Utilizing Python for data transformation and bash scripts for function coordination, we constructed a pipeline leveraging various AWS services. Our investigation primarily focused on the performance implications of two critical design trade-offs: the choice of CPU architecture (Intel vs. ARM) and container start conditions (hot vs. cold starts). The comparative analysis revealed notable differences in execution times and efficiency between the CPU architectures. Furthermore, our findings underscore the significant impact of container states on the performance stability and responsiveness of the TLQ pipeline. These insights contribute to understanding the overall efficiency and cost-effectiveness of serverless applications, highlighting key considerations for cloud computing optimization and design.

CCS CONCEPTS

Computing methodologies → Distributed computing methodologies

Software and its engineering → Software performance

Information systems → Data management systems

KEYWORDS

Serverless, TLQ Pipeline, AWS Lambda, Data Transformation, Data Loading, Data Querying, Python, Bash Scripting, AWS Services, Cloud Computing, Asynchronous Flow Control, Performance Metrics, CPU Architecture, ARM Processors, Intel Processors, Design Tradeoffs, Serverless Architecture, Cloud Performance Analysis, Comparative Analysis, and Cost Efficiency.

ACM Reference format:

Patrick Hern, Cameron Gregoire, Angela Farin, and Tomoki Kusunoki. Comparing FaaS TLQ Pipeline Performance Based on CPU Architecture and Cold/Hot State: Comparing Implementation Decisions Using AWS. *UW Tacoma, Tacoma, WA, USA, 4 pages.*

1 Introduction

This project displays a serverless TLQ data pipeline, implemented through AWS lambda functions. The TLQ application encapsulates data transformation, loading, and querying processes seamlessly within a serverless architecture. Our group's interpretation of the TLQ application involves an integration of Python scripts for data transformation, bash scripts for executing serverless functions, and various AWS services. The project's primary focus lies in evaluating the performance implications of key design choices, including the comparison of Intel and ARM processors, and the assessment of container states. We capture runtime metrics, CPU architecture, and hosting costs; contributing insights into the efficiency and cost-effectiveness of different configurations within the TLQ pipeline.

$$Avg = (\Sigma x_i) / N$$

Average (Mean) Calculation: Used for determining the average performance metrics.

$$\sigma = \sqrt{1/N \ * \ \Sigma(x_i - \mu)^2)}$$

Standard Deviation: Measures the variability of the data set.

$$CV = (\sigma / \mu) \times 100$$

Coefficient of Variation: Indicates the relative variability as a percentage average

1.1 **Research Questions**

In this study, we aim to address two fundamental aspects of serverless computing through our research questions. These questions are designed to dissect the critical factors influencing the performance and efficiency of serverless cloud applications.:

RQ-1: How does the choice of CPU architecture (Intel vs. ARM) affect the performance of serverless cloud applications in terms of runtime, throughput, and cost?

RQ-2: What is the impact of container start conditions (hot vs. cold starts) on the performance and stability of serverless cloud applications across different CPU architectures?

2 **Case Study**

The serverless application implemented by our group is a TLQ Pipeline that uses the Sales Database as a basis for the pipeline. Each step of the pipeline has its own AWS Lambda function so that we can use different Intel and ARM processors. There is a script that sequentially calls each step of the pipeline and records relevant information about the performance.

An interesting consequence of implementing the application as a series of FaaS/Lambda functions is that the user doesn't need to worry as much about the creation and management of the application. However, a notable side effect of this is the necessity to store and pull data from S3 and RDS given the fact that there is no local storage. This does add some price obfuscation into the mix and costs. Lambda itself is very cheap, though RDS tends to be a bit more expensive. However, the availability and modularity of Lambda functions are incredibly useful for fine-grain function resource allocation.

2.1 **Design Tradeoffs**

The implementation of the serverless TLQ pipeline involves a series of design tradeoffs aimed at optimizing performance and cost-effectiveness. Our first primary tradeoff centers around the choice of CPU architecture- a comparative analysis between Intel and ARM processors. This measurement aims to uncover how the selected processor influences runtime metrics within the TLQ pipeline.

We are expecting to see ARM yield better results. Although, Intel processors are known for their strong single-threaded performance and are chosen for their high clock speed; ARM has gained popularity in certain cloud computing scenarios due to their power efficiency and scalability.

Secondly, our design considerations extend to container states- comparing between hot and cold containers. This experiment aims to comprehend the impact of container states on function performance, addressing challenges related to container reusability and overall system responsiveness.

We are also expecting to observe differences in performance metrics between hot and cold container states. Hot containers, being prewarmed, should potentially exhibit faster response times compared to cold containers, which need initialization.

2.2 **Serverless Application Implementation**

The serverless TLQ pipeline application is implemented by using Python scripts for data transformation and bash scripts for serverless functions within AWS lambda- creating a seamless integration of data transformation, loading, and querying processes.

More specifically, our application implementation is broken up into three services. Service 1 calls “postCSVData.py” which loads 10000SalesRecord.csv and converts it to a JSON file, and POSTS the JSON data one line at a time. The data is then transformed and stored in a CSV file in S3 by calling TransformAndStore in AWS Lambda.

Service 2 uses a bash script “loadData.sh” which is responsible for invoking the DataLoader component deployed on AWS. This function loads the CSV data into a MYSQL database hosted on Amazon RDS. The target table for storing the loaded data is identified as “SalesRecordTab.” When using the load service, asynchronous flow is supported, allowing for parallel execution of tasks, and enhancing efficiency during the population of the database.

Service 3 has three bash scripts- 'queryCol.sh,' 'queryColAgg.sh,' and 'queryColFilter.sh' which are executed. These scripts can be run with the ‘time’ command to measure their running time. The scripts are designed to send HTTP POST requests containing query conditions. They invoke the QueryHandler implemented in AWS Lambda. Upon invocation, this lambda function processes the queries against the underlying MySQL database boosted on RDS, which is ‘SalesRecordTab’.

Finally, we can execute the TLQ pipeline and gather data by calling “execPipeline.sh”. This bash script constructs a CSV file containing essential runtime metrics, including hot/cold runtime, and CPU architecture details. Within the bash script, there is a loop initialized to run from 0 to 1, which can be adjusted for extensive testing. Notably, these functions are designed to maintain a clean console environment by redirecting the console output to a null directory, preventing unnecessary clutter.

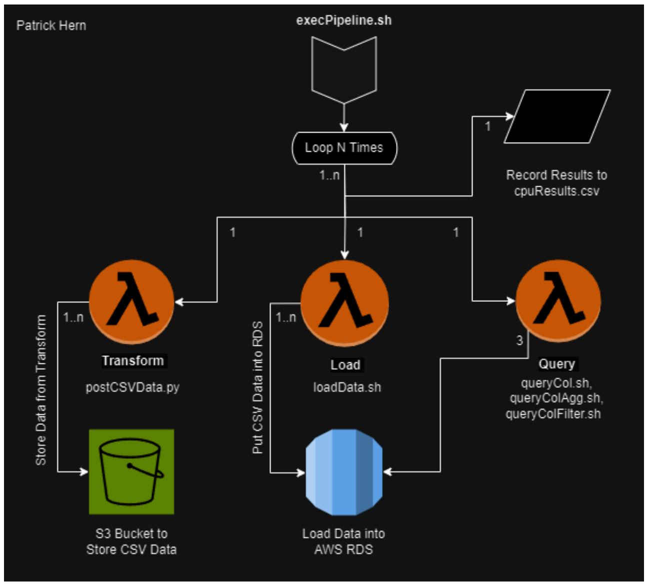


Figure 1: **Diagram of the basic structure of the TLQ Pipeline, created using diagrams.net.**

2.3 Experimental Approach

For our experimental approach, we decided to use a custom BASH script to measure the performance of the pipeline. Each step of the pipeline includes JSON output that comes as a standard with the basic SaaF [2] structure. Given that our primary concerns were with CPU architecture and the Hot/Cold states of the pipeline, we elected to use the JSON output sparingly to avoid complicated and unnecessary data for our experiments. At the time of creating the BASH script for testing the pipeline, we were not aware of the FaaS Runner tool so we elected to use exclusively the client-side test script.

For client-side testing, as has previously been mentioned, there is a script in the GitHub repository called 'execPipeline.sh' which does exactly as it sounds. It runs the TLQ pipeline sequentially, waiting for each service of the pipeline to finish before continuing. The script creates a CSV file that contains the rows: CPU Type, Hot/Cold, Service 1 Runtime, Service 2 Runtime, and Service 3 Runtimes (for each of the three queries), and the total runtime for the pipeline. There is also a loop control variable the user can change to run the pipeline several times without the need to manually call the functions. Each time the script completes, it records the data directly to the CSV file.

To assess the impact of CPU architecture and container state on performance, we conducted 200 test runs—100 using the Intel x86 architecture and 100 using ARM, divided equally between hot and cold starts. This testing spanned both architectures and container states, using a dataset of 10,000 rows.

It is important to note that Lambda's functions must be manually changed from Intel (x86) and ARM processors. This isn't a problem for a real-world situation where you would pick one architecture and stick with it, however, it does make a difference when trying to test the pipeline. That being said, the script uses the JSON return object to extract the CPU's architecture. On top of that, the function memory was at the default of 128 MB.

3 Experimental Results

Our experimental results indicate some significant performance variance between ARM and Intel processors in serverless environments. Notably, in hot start conditions, where functions are frequently invoked, the ARM processors demonstrated a throughput of about 336 records per second compared to Intel's 232. Outperforming Intel in throughput by about 45%, Emphasizing ARM's efficiency in a scenario of consistent operations.

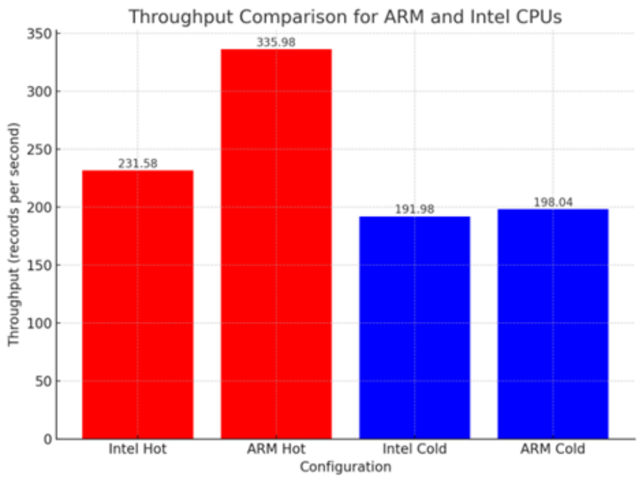


Figure 2: **Throughput performance comparison between ARM and Intel CPUs, showcasing number of records processed under hot and cold conditions.**

In hot start conditions, ARM processors exhibited a much lower standard deviation in performance as compared to intel. Indicating that ARM processors are more consistent in their execution times, providing a more stable and predictable performance. Furthermore, ARM also showed a 9% lower coefficient of variation than Intel in hot starts, further suggesting better predictability and performance variability. This consistency is especially valuable to applications looking for predictable response times.

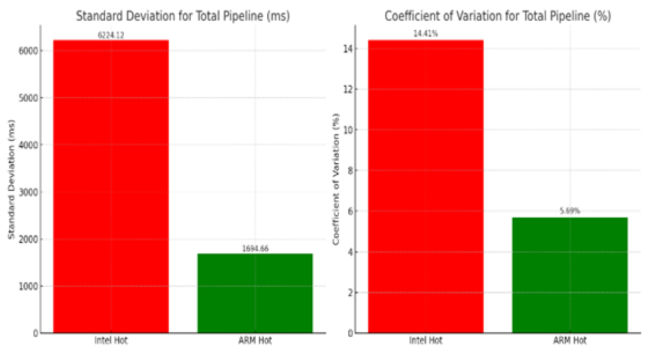


Figure 3: **left- Comparison of the standard deviation in execution times for ARM and Intel CPUs in Hot start conditions. right- Comparison of the coefficient of variation for ARM and Intel CPUs during Hot starts**

Conversely, in cold start conditions, where functions are called infrequently, Intel indicated a slightly more stable environment, with a lower standard deviation and a 1.28% lower coefficient of variation as compared to ARM. While ARM still maintained a higher throughput of 198 records per second to Intels 193. This suggests that Intel is slightly more consistent in scenarios of sporadic function invocation but is also slightly slower.

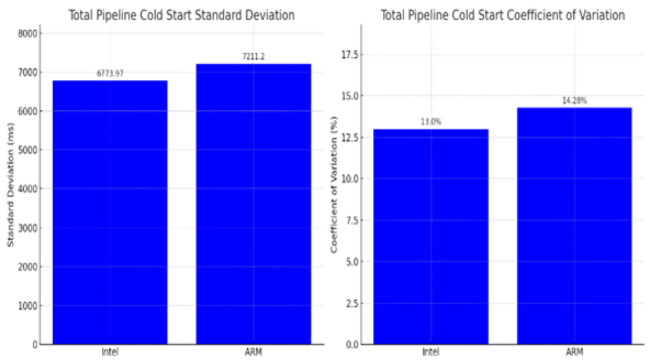
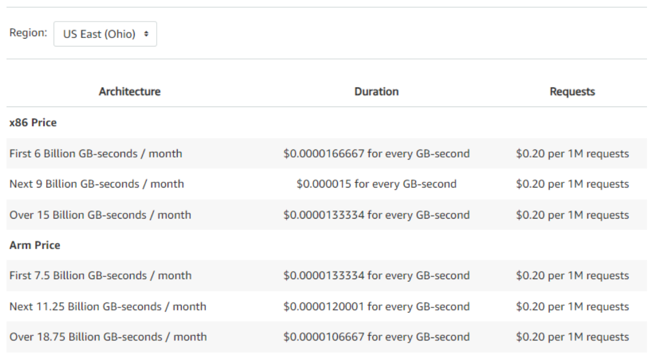


Figure 4: **left- Comparison of the standard deviation in execution times for ARM and Intel CPUs in cold start conditions. right- Comparison of the coefficient of variation for ARM and Intel CPUs during cold starts**

As far as the cost metrics are concerned, AWS has given us some price markers for the 3-4 weeks of debugging, testing, and data collection we've performed.

By far, AWS Lambda was the cheapest component of the pipeline, given that we still have our first 1 million for free working. However, according to the AWS Lambda documentation, every 1 million requests charges $0.20, and for the first 6 billion GB-seconds of throughput, it charges $0.0000167 per GB-second every month for Intel and $0.000013 for ARM [1].

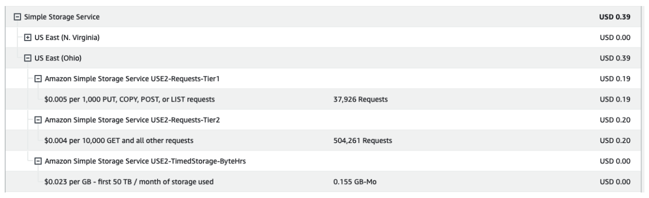


Region: US East (Ohio)

Architecture	Duration	Requests
x86 Price		
First 6 Billion GB-seconds / month	$0.0000166667 for every GB-second	$0.20 per 1M requests
Next 9 Billion GB-seconds / month	$0.000015 for every GB-second	$0.20 per 1M requests
Over 15 Billion GB-seconds / month	$0.0000133334 for every GB-second	$0.20 per 1M requests
Arm Price		
First 7.5 Billion GB-seconds / month	$0.0000133334 for every GB-second	$0.20 per 1M requests
Next 11.25 Billion GB-seconds / month	$0.0000120001 for every GB-second	$0.20 per 1M requests
Over 18.75 Billion GB-seconds / month	$0.0000106667 for every GB-second	$0.20 per 1M requests

Figure 5: **Table from AWS documentation that includes the pricing policy for Intel and ARM CPUs in the US East (Ohio) region.**

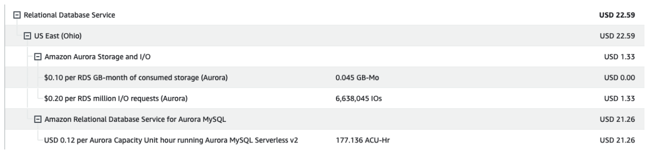
After the 3-4 weeks of using AWS S3, we had $0.39 charged to our AWS account in the form of 40,000 PUT/COPY/POST/LIST requests at $0.19. The last $0.20 came from the GET requests which are charged at $0.004 per request with 500,000 requests called on our end.



Simple Storage Service		**USD 0.39**
US East (N. Virginia)		USD 0.00
US East (Ohio)		USD 0.39
Amazon Simple Storage Service USE2-Requests-Tier1		USD 0.19
$0.005 per 1,000 PUT, COPY, POST, or LIST requests	37,926 Requests	USD 0.19
Amazon Simple Storage Service USE2-Requests-Tier2		USD 0.20
$0.004 per 10,000 GET and all other requests	504,261 Requests	USD 0.20
Amazon Simple Storage Service USE2-TimedStorage-ByteHrs		USD 0.00
$0.023 per GB - first 50 TB / month of storage used	0.155 GB-Mo	USD 0.00

Figure 6: **Table from AWS that specifies the cost of using S3 and with our pipeline for 3-4 weeks.**

Finally, the most expensive part of our TLQ pipeline was the Relational Database Service. The total price came out to $22.59 with $21.26 of those charges coming from the Amazon Relational Database Service for Aurora MySQL. The rest of the $1.33 comes from the $0.20 per million I/O requests for Aurora.



Relational Database Service		**USD 22.59**
US East (Ohio)		USD 22.59
Amazon Aurora Storage and I/O		USD 1.33
$0.10 per RDS GB-month of consumed storage (Aurora)	0.045 GB-Mo	USD 0.00
$0.20 per RDS million I/O requests (Aurora)	6,638,045 IOs	USD 1.33
Amazon Relational Database Service for Aurora MySQL		USD 21.26
USD 0.12 per Aurora Capacity Unit hour running Aurora MySQL Serverless v2	177.136 ACU-Hr	USD 21.26

Figure 7: **Table from AWS that specifies the cost of using Relational Database Service with our pipeline for 3-4 weeks.**

4 Conclusions

Our study on serverless applications, comparing ARM and Intel CPU architectures under different container start conditions, revealed significant performance differences. As Table 1 illustrates, ARM processors demonstrated superior throughput in warm starts, approximately 45% higher than Intel, indicating a marked efficiency in consistent operation scenarios. This suggests the potential for cost savings when using ARM for frequently invoked functions.

CPU Architecture	**Hot Start Throughput (records/sec)**	**Cold Start Throughput (records/sec)**
Intel	~232	~192
ARM	~336	~198

Table 1: **Presents the throughput performance comparison between ARM and Intel CPUs in hot and cold start conditions**.

Intel processors, however, showed greater stability in cold starts, as seen in Tables 2/3. The lower standard deviation and coefficient of variation point to Intel's suitability for applications with irregular usage patterns.

CPU Architecture	**Hot Start Standard Deviation (ms)**	**Cold Start Standard Deviation (ms)**
Intel	~6224	~6774
ARM	~1695	~7211

Table 2: **Shows the standard deviation in execution times for both ARM and Intel CPUs.**

CPU Architecture	**Hot Start Coefficient of Variation (%)**	**Cold Start Coefficient of Variation (%)**
Intel	14.41%	13.00%
ARM	5.69%	14.28%

Table 3: **Compares the coefficient of variation for ARM and Intel CPUs in both hot and cold starts.**

In conclusion, our study's testing across 200 runs with a 10,000 row dataset has led to insights regarding the performance of serverless TLQ pipelines. Addressing RQ-1, we found that ARM processors delivered faster throughput and demonstrated enhanced consistency in execution during hot starts as compared to Intel processors. However, RQ-2 revealed that Intel processors showed marginally better stability in cold start conditions. These results highlight the significance of CPU architecture and container start conditions in cloud applications, suggesting ARM's suitability for sustained performance and Intel's potential in less frequent usage scenarios.

Our analysis focused on performance metrics, future research could expand to include direct cost comparisons, which would provide a more well rounded perspective for decision-making in cloud computing environments. Our findings show the need for a careful consideration of both CPU architecture and container states in using serverless applications while balancing performance and stability.

REFERENCES

[1] Vişan, S. (2006). AWS Lambda Pricing. Amazon Web Services. https://aws.amazon.com/lambda/pricing/

[2] Lloyd, W. J. (2018, November 1). SAAF: Serverless Application Analytics Framework. GitHub. https://github.com/wlloyduw/SAAF/tree/master

[3] JV Roig, Deborshi Choudbury, Josh DeMuth, and Rishi Singla. (2023, October 24). Comparing AWS Lambda ARM vs x86: Performance, Cost, and Analysis. AWS Partner Network Blog. https://aws.amazon.com/blogs/apn/comparing-aws-lambda-arm-vs-x86-performance-cost-and-analysis-2/

[4] Nikita S. (2023, June 27). What are Arm-Based Servers? Comparison with x86, Benefits, and Drawbacks. Cloud Panel. https://www.cloudpanel.io/blog/arm-based-servers/

[5] Taavi Rehemägi. (2021, July 24). Can We Solve Serverless Cold Starts? https://dashbird.io/blog/can-we-solve-serverless-cold-starts/