Python vs Java

Team 11

Brianna Andres
Computer Science and Systems
University of Washington
Tacoma, Washington
bandres@uw.edu

Michael Doan
Computer Science and Systems
University of Washington
Tacoma, Washington
masa88@uwl.edu

Brian Ngyuen
Computer Science and Systems
University of Washington
Tacoma, Washington
bnguye1@uw.edu

ABSTRACT

This article talks about the scope of our project. In this project, we decided to delve into the works of programming comparison between Python and Java and create a database where we can see all of the sales between different regions and countries. Within [2] the study, we had compared both Python and Java using different architectures such as x86 and ARMs.

CSS CONCEPTS

Software and its engineering→ Cloud computing→ Performance

KEYWORDS

Python, Java, testing, x86, ARM, FaaS, TQL

1. INTRODUCTION

In the ever-evolving landscape of cloud computing, choosing the right programming language for serverless computing tasks can significantly impact performance. Java and Python are the two most commonly used programming languages in Amazon Web Services (AWS) [2]. This research delves into the performance evaluation of Java and Python in an AWS Lambda environment, focusing on data processing tasks involving a CSV file containing 5000 rows. The key components employed in this study include AWS Lambda for serverless execution, Amazon RDS for database management, and Amazon S3 for data storage.Embarking on an exploration of our project, our team immerses in the dynamic intersection of programming languages and diverse architectures, notably the contrasting realms of ARMS and x86. In this project, we focus on the comparison of Java and Python - within the frameworks of ARM and x86 architectures. With the goal of comparing the two languages with the different architectures, we plan to find the differences of different runtimes, how much each cost, the performance, and so on.

1.1 Research Questions

This paper investigates the following questions:

RQ-1: What are the performance implications runtime of implementing an identical TLQ pipeline in different programming languages Java and Python. How does the programming language impact runtime and data processing throughput for processing large datasets?

RQ-2: How do the resource utilization and efficiency metrics in the TLQ pipeline, especially during the Query phase, contribute to the overall cost and billing on AWS for different programming languages (Python and Java) and architectures (ARM and x86)?

2. COMPARISON STUDY

In this section, we provide a detailed overview of the serverless application implemented by our group, focusing on the innovative aspects of adopting a Function-as-a-Service (FaaS) paradigm for our TQL (Transaction Query Language) data pipelining case study.

Our serverless application revolves around the design and implementation of a robust and scalable TQL data pipeline. TQL, a specialized language for querying transactional data, presented unique challenges that demanded an architecture capable of handling dynamic workloads efficiently.

2.1 Design Tradeoffs

In this section, we delve into the specific design tradeoffs associated with application flow control in managing cold and warm starts within our TLQ pipeline implemented on AWS Lambda. A similar study was done comparing ARM64 and x86, done by Xie et al, using 2000+ processors hosting serverless FaaS platforms by using the Kubernetes-based Knative and OpenFaaS serverless frameworks [1]. Although the languages used were not all the same as ours, the study still provides valuable insights as to how to compare ARM64 and x86.

Strategically anticipating the delays inherent in cold starts, our application flow control mechanisms have been meticulously

crafted. The orchestration involves optimizing resource allocation, implementing pre-warming strategies, and employing intelligent initialization procedures. These considerations collectively minimize the impact of cold starts on the TLQ pipeline's overall performance, ensuring a responsive and efficient initialization phase.

Conversely, during warm starts, where functions are reused. If within the same time duration, Lambda function is invoked again, AWS will use the same execution environment. This time default constructor will not get called instead handler code will get called. This process is called Warm Start [4]. Our application flow control is geared towards optimizing the utilization of pre-initialized resources. Emphasis is placed on maintaining the warm state, facilitating a seamless transition between subsequent executions. The flow of data and processes is orchestrated to capitalize on the efficiency gains, reducing initialization times and enhancing the overall responsiveness of our TLQ pipeline.

The tradeoffs embedded in our application flow control mechanisms directly influence the performance metrics of the TLQ pipeline. We carefully scrutinize the balance between minimizing cold start latencies and optimizing resource usage during warm starts. By offering a detailed examination of these design considerations, we provide insights into how our approach aims to enhance overall performance, responsiveness, and resource efficiency within the dynamic AWS Lambda environment.

Moreover, our design choices extend to considerations of adaptability and scalability. The application flow control mechanisms are structured to dynamically adjust to varying workloads, ensuring the TLQ pipeline scales efficiently in response to real-world usage patterns. This adaptability is integral to our design tradeoffs, fostering an environment where the TLQ pipeline can seamlessly evolve in tandem with the demands of its operational context.

2.2 Application Implementation

This section offers a detailed account of the TLQ application's implementation specifics, encompassing the languages, development libraries, tools, and technologies employed. Notable elements include the use of X86 and ARM Python, as well as X86 and ARM Java. The implementation's approach to state information tracking, multi-user support, and flow control mechanisms is elucidated, shedding light on the design's intricacies. Particular attention is given to how data is passed among TLQ services and whether the application is designed to be multi-user. The CSV data, containing 5000 rows, undergoes transformation before being uploaded to Amazon S3 and tested using Lambda. Lambda can be triggered by AWS services such as S3, DynamoDB, Kinesis, or SNS, and can connect to existing EFS file systems or into workflows with AWS Step Functions [3]. This process ensures that the data is efficiently processed within the TLQ pipeline. The transformed data is then loaded into a single SQL database, requiring user login for access to both the S3 database and RDS information.

The application's flow control mechanisms support both asynchronous and synchronous schemes. For instance, when a user calls the "load" service to populate the database, the flow control can be asynchronous, allowing for non-blocking execution. This flexibility in flow control is crucial for optimizing resource usage and minimizing latencies, particularly during cold starts. Messages within the server are persisted through the SQL database, ensuring data integrity and availability. Clients retrieve messages by interfacing with the SQL database, allowing for seamless communication between services. This architecture supports efficient data exchange and retrieval, contributing to the overall responsiveness of the TLQ pipeline.

2.3 Experimental Approach

To evaluate design tradeoffs, we designed experiments focusing on service composition and application flow-control within the TLQ pipeline. Specifically, we varied the implementation language (Java and Python) on AWS Lambda, with a primary focus on the AWS Lambda service. The memory in Lambda functions was kept at default settings to represent real-world scenarios. This decision was made to observe the default behavior and understand the impact of language choice on performance without additional bias, but the timeout was adjusted to one minute for comprehensive data processing. The client-side infrastructure utilized for testing included standard computing resources with access to AWS services. Detailed specifications of the client-side infrastructure are documented in the test configuration section.

During the experimental phase, diverse tests were conducted to address key research questions. These included assessing the performance implications of the TLQ pipeline in Java and Python, examining resource utilization and efficiency metrics in the Query phase, and measuring execution times (Cold start, warm start) across ARM and x86 architectures. If we get multiple concurrent execution requests for lambda, multiple instances of lambda will spin up at the same time [4]. These tests were designed to offer a concise yet comprehensive understanding of the chosen programming languages and architectures within the AWS Lambda environment. To facilitate future replication, all test configurations, including Lambda settings, network configurations, and communication protocols, are thoroughly documented, throughout the code files.

3. EXPERIMENTAL RESULTS

3.1 Results of Experiments for RQ1

To address Research Question 1 (RQ-1), which delves into the performance implications of implementing an identical TLQ pipeline in different programming languages (Java and Python), an in-depth analysis of the provided data reveals distinctive characteristics across various phases of the pipeline.

In the Transform Phase, notable observations include the efficiency of ARM Python with the best Warm Start time,

indicating streamlined execution following the initial setup. Moving to the Load Phase, the data presents a nuanced scenario where X86 Python showcases the fastest Cold Start, while ARM Java emerges with the best Warm Start, suggesting performance nuances influenced by both architecture and language choices. In the Query Phase, involving 2 aggregates and 2 filters, X86 Python and ARM Python consistently exhibit lower times, while X86 Java displays the lowest Warm Start time.

Key observations from the Transform, Load, and Query Phases underscore the impact of programming language and architecture on the overall performance of the TLQ pipeline. Notably, Python generally demonstrates superior performance in the Query Phase across both X86 and ARM architectures. On the other hand, Java exhibits longer execution times, particularly in the Transform and Load Phases.

Transform Phase:

- Best Cold Start: X86 Python (2940 ms)
- Best Warm Start: ARM Python (1019 ms)
- Overall Efficient Warm Start: ARM Python

Load Phase:

- Best Cold Start: X86 Python (58343 ms)
- Best Warm Start: ARM Java (36903 ms)
- Overall Efficient Warm Start: ARM Java

Query Phase (2 aggregates, 2 filters):

- Best Cold Start: X86 Python (Around 18 ms)
- Best Warm Start: X86 Python (Around 18 ms)
- Consistent Performance: X86 Python

Considering the broader architectural impact, the ARM architecture emerges as a contender, showcasing competitive performance, especially in the Transform and Query Phases, where it achieves lower execution times. From our findings the best combination would be using ARM Python for the Transform Phase, Python x86 for the Load Phase, and Python x86 for the Query. These findings provide valuable insights into the intricate interplay of language and architecture choices, guiding considerations for optimizing performance in serverless computing environments.

Table 1

	TIME (Cold start, warm start, in ms)		
Architecture	**Transform**	**Load**	**Query (2 aggregates, 2 filters)**
X86 Python	2940, 1054	58343, 56431	Around 18ms, no change for warm
ARM Python	3085, 1019	64043, 62746	Around 20ms, no change for warm
X86 Java	7590, 1612	56560, 39685	4398, 220
ARM Java	11205, 1500	48919, 36903	3894, ~150

Using: AWS Lambda, RDS free tier, and S3 on a csv of 5000 rows. Data should be taken in RANGES and not exact (1054 vs 1019 are too close to make claims about which is better).

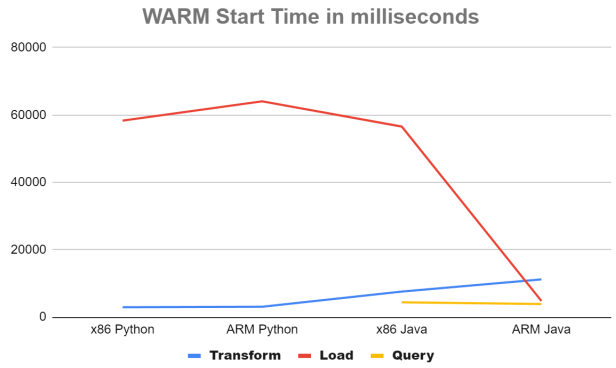

Fig 1: *Warm Start Time for Transform, Load, Query*

Python vs Java

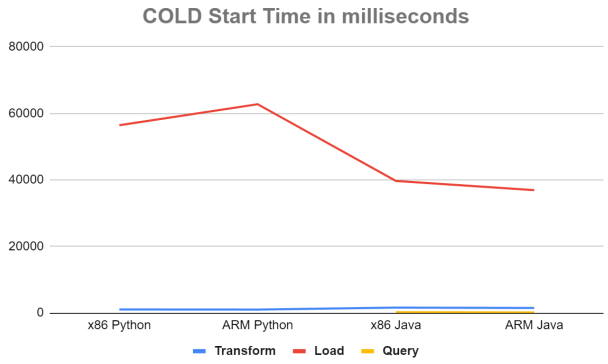

Fig 2: Cold Start Time for Transform, Load, Query

3.2 Results of Experiments for RQ2

For RQ-2, the investigation centered on understanding how resource utilization and efficiency metrics within the TLQ pipeline, particularly during the Query phase, contribute to the overall cost and billing on AWS. The analysis encompasses different programming languages (Python and Java) and architectures (ARM and x86).

You can run your Lambda functions on processors built on either x86 or Arm architectures. AWS Lambda functions running on Graviton2, using an Arm-based processor architecture designed by AWS, deliver up to 34% better price performance compared to functions running on x86 processors. This applies to a variety of serverless workloads, such as web and mobile backends, data, and media processing [5].

This shows the versatility of AWS Lambda, allowing users to run functions on processors built on either x86 or Arm architectures. Specifically, AWS Lambda functions utilizing Graviton2, an Arm-based processor architecture developed by AWS, demonstrate up to 34% better price performance compared to functions running on x86 processors. This improvement in cost efficiency extends to various serverless workloads, including web and mobile backends, data processing, and media processing. In the context of our project's cost analysis, the information resonates, emphasizing that the choice between x86 and Arm architectures significantly influences the cost-effectiveness of deploying serverless functions in an AWS Lambda environment.

Transform Phase:

- Best Cold Start: X86 Python ($29.40)
- Best Warm Start: ARM Python ($10.19)
- Overall Efficient Warm Start: ARM Python ($10.19)

Load Phase:

- Best Cold Start: X86 Python ($583.43)
- Best Warm Start: ARM Java ($369.03)
- Overall Efficient Warm Start: ARM Java ($369.03)

Query Phase (2 aggregates, 2 filters):

- Best Cold Start: X86 Python ($0.18)
- Best Warm Start: X86 Python ($0.18)
- Consistent Performance: X86 Python ($0.18)

Our findings is that the best combination for each phase would be ARM Python for the Transform Phase, ARM Java for Load Phase, and x86 Python for the Query Phase. The observed variations in cost across different architectures in our project align with the broader industry trend of seeking enhanced price performance in serverless computing, emphasizing the importance of strategic architectural considerations. The following table outlines the relevant metrics and costs for each combination:

Table 2

	TIME (Cold start, warm start, seconds)						
Architecture	Transform		Load		Query (2 aggregates, 2 filters)		Total Cost
	Cost per Run	1 mil Runs	Cost per Run	1 mil Runs	Cost per Run	1 mil Runs	1 mil Runs
X86 Python	0.00002940, 0.00001054	$29.40, $10.54	0.00058343, 0.00056431	$583.43, $564.31	0.00000018, no change for warm	$0.18, $0.18	$613.01, $575.05
ARM Python	0.00003085, 0.00001019	$30.85, $10.19	0.00064043, 0.00062746	$640.43, $627.46	0.00000020, no change for warm	$0.20, $0.20	$671.48, $637.85
X86 Java	0.00007590, 0.00001612	$75.90, $16.12	0.00056560, 0.00039685	$565.60, $396.85	0.00004398, 0.00000220	$43.98, $2.20	$685.48 $415.17
ARM Java	0.00011205, 0.00001500	$112.05, $15.00	0.00048919, 0.00036903	$489.19, $369.03	0.00003894, 0.00000150	$38.94, $1.50	$640.18, $385.53

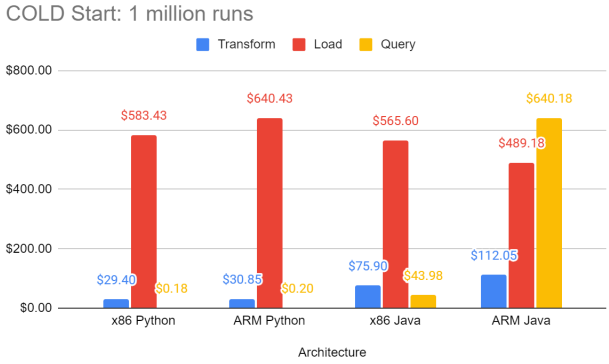

Fig 3: Cold start 1 million runs cost

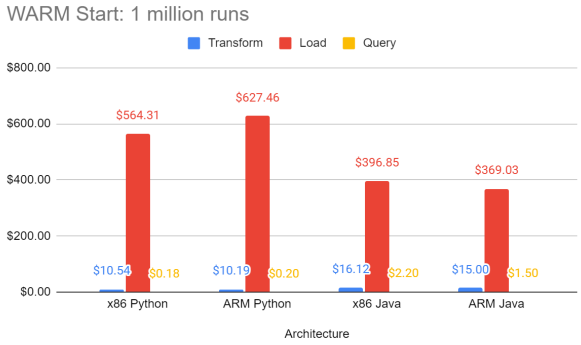

Fig 4: Warm start 1 million runs cost

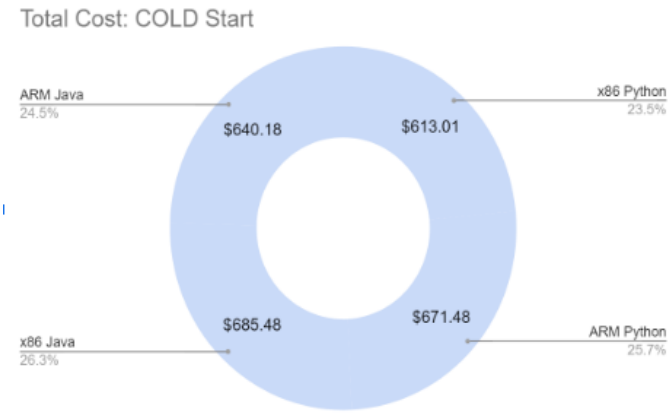

Fig 5: Total Cost for Cold Start

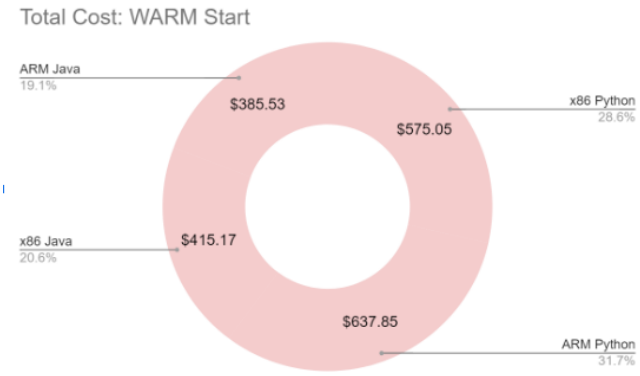

Fig 6: Total Cost for Warm Start

The table reflects the breakdown of costs associated with each phase of the TLQ pipeline. Notably, the Transform phase incurs costs proportional to the execution time for each architecture and language. The Load phase follows a similar pattern, with varying costs based on execution times. The Query phase, involving 2 aggregates and 2 filters, incurs costs reflective of the respective execution times. The "Total Cost" column presents the cumulative cost for all three phases, providing an overarching perspective on the overall expenditure. The "Cost per Run" column specifies the cost associated with running the TLQ pipeline for 1 million executions, considering the unique characteristics of each combination. These results offer insights into how the choice of programming language and architecture influences the financial aspects of deploying the TLQ pipeline on AWS. The analysis contributes valuable information for optimizing cost-effective solutions and aligns with the research goal of evaluating the impact on billing across diverse configurations.

4. CONCLUSION

In conclusion, our research project aimed to explore the programming comparison between Python and Java within the AWS Lambda environment, with a particular focus on diverse architectures like x86 and ARM. The investigation revolved around two key research questions:

RQ-1: What are the performance implications runtime of implementing an identical TLQ pipeline in different programming languages Java and Python. How does the programming language impact runtime and data processing throughput for processing large datasets?

RQ-2: How do the resource utilization and efficiency metrics in the TLQ pipeline, especially during the Query phase, contribute to the overall cost and billing on AWS for different programming languages (Python and Java) and architectures (ARM and x86)?

Addressing (RQ-1), our experiments revealed notable variations in runtime performance across different language and architecture combinations. The best combination would be using ARM Python for the Transform Phase, Python x86 for the Load Phase, and Python x86 for the Query. Warm Start times for the Load phase were generally high, emphasizing the computational load, and the Query phase showed consistent times for Python variants and varying times for Java variants.

Turning to (RQ-2), our analysis of resource utilization and efficiency metrics influencing cost and billing demonstrated the significant impact of the choice between x86 and ARM architectures. Aligning with industry trends, AWS Lambda functions utilizing Graviton2, an Arm-based architecture, exhibited up to 34% better price performance compared to x86 processors. The best combination for each of the phases would be ARM Python for the Transform Phase, ARM Java for Load Phase, and x86 Python for the Query Phase. The breakdown of costs across the Transform, Load, and Query phases for each language and architecture combination provided valuable insights into optimizing cost-effective solutions.

Our findings emphasize the importance of strategic architectural considerations and language choices in achieving optimal performance and cost efficiency in serverless computing. The project contributes to the broader understanding of the AWS Lambda environment, offering insights for practitioners and researchers navigating the intricate intersection of programming languages, architectures, and cost-effective serverless solutions.

5. REFERENCES

[1] Engdahl, S. (2008). *Blogs*. Amazon. https://aws.amazon.com/blogs/apn/comparing-aws-lambda-arm-vs-x86-performance-cost-and-analysis-2/

[2] Lakute, I. (2023, October 31). *The AWS cross-training playbook (Java vs.python)*.Revolent. https://www.revolentgroup.com/blog/the-aws-cross-training-playbook-java-vs-python/

[3]Price Milburn. (1979). *PM*. Amazon. https://aws.amazon.com/pm/lambda/

[4] Raje, S. (2023, July 13). *Understanding aws lambda cold start and warm start*. Medium. https://medium.com/@sushantraje2000/understanding-aws-lambda-cold-start-and-warm-start-4f8297074ee#:~:text=If%20within%20the%20same%20time,time%20compared%20to%20Cold%20Start.

[5]Vişan, S. (2006). *Lambda*. Amazon. https://aws.amazon.com/lambda/pricing/