Serverless Computing Performance Analysis: A Comparative Study of Multi-Language Lambda Function Implementations

Brandon Morgan[†]
University of Washington Tacoma
Tacoma, WA
bmorgan1@uw.edu

Aly Badr
University of Washington Tacoma
Tacoma, WA
badraly@uw.edu

Shu-Ren Shen
University of Washington Tacoma
Tacoma, WA
vishen@uw.edu

ABSTRACT

This research investigates the performance characteristics of a Transform-Load-Query (TLQ) data processing pipeline implemented across multiple programming languages using AWS Lambda serverless architecture. By developing parallel implementations in Java and Python, the study systematically examines runtime, data processing throughput, cost implications, and the efficacy of AI-assisted code generation tools. Through a methodology involving 10 repeated tests with varied dataset sizes, the research aims to provide empirical insights into the comparative advantages of different programming languages in serverless computing environments.

The experimental design explores four critical research dimensions: runtime performance, data processing throughput, computational cost analysis, and the potential of generative AI tools like ChatGPT and GitHub Copilot in facilitating cross-language implementation. By systematically analyzing these implementations, the research contributes to the understanding of serverless computing trade-offs and provides practical guidance for developers and architects in selecting appropriate programming languages and development methodologies for data processing pipelines.

The study employs a controlled approach, testing both small and progressively larger datasets to assess the scalability and computational efficiency of the multi-language serverless pipeline. Through this methodical evaluation, the research seeks to uncover insights into runtime, throughput, and cost-effectiveness of different programming language implementations in a serverless framework, while simultaneously examining the role of AI-powered code generation in streamlining development processes.

CCS CONCEPTS

- Computer systems organization → Cloud computing
- Software and its engineering → Serverless architectures
- Performance → Performance analysis and modeling
- General and reference → Empirical studies

Serverless Computing, Multi-Language Implementation, AWS Lambda, Performance Analysis, Data Processing Pipeline, Cloud Computing, Comparative Study, Runtime Performance, AI Code Generation, Throughput Measurement, Computational Efficiency, Java, Python, Generative AI Tools

1 Introduction

This report presents the design and implementation of a serverless Transform-Load-Query (TLQ) pipeline application. The TLQ application processes sales data in a serverless architecture using AWS Lambda and Amazon S3, showcasing a highly scalable and cost-efficient approach to data transformation, storage, and querying. The dataset used for this project consists of sales data samples provided by Wes Lloyd, comprising 5,000 and 10,000 records. This project was implemented using a combination of Java and Python programming languages.
The TLQ pipeline consists of three distinct services:

1. Transform Service: This service ingests raw sales data in CSV format and applies a series of transformations. These transformations include:
   - Adding a column to calculate Order Processing Time, defined as the number of days between the Order Date and the Ship Date.
   - Standardizing the Order Priority column by mapping the values L, M, H, and C to their respective descriptive labels: "Low," "Medium," "High," and "Critical."
   - Adding a Gross Margin column, calculated as Total Profit / Total Revenue and stored as a floating-point value.
   - Removing duplicate records identified by the Order ID.

The transformed data is then saved as a new CSV file in an Amazon S3 bucket for further processing.

2. Load Service: The Load Service reads the transformed CSV data from S3 and loads it into a relational

database. For this implementation, an SQLite database stored within an S3 bucket is used to facilitate efficient querying while maintaining a lightweight and serverless infrastructure.

3. Query Service: This service allows users to query the data stored in the SQLite database based on specific input parameters. Example queries include filtering data by Order Priority, calculating aggregate metrics, or retrieving subsets of data that meet particular criteria. The results are returned in a user-friendly format.

The entire pipeline leverages AWS Lambda for its serverless implementation, ensuring scalability and minimizing infrastructure overhead. The use of Amazon S3 for data storage enables seamless data sharing between services and guarantees high availability. By combining Python and Java, the project demonstrates the flexibility of using multiple programming languages within a serverless ecosystem.

This implementation adheres to the principles of the original TLQ application while introducing unique choices, such as the use of SQLite stored in S3 for database management and the use of both Java and Python for service development.

1.1 Research Questions

This paper investigates the performance, cost implications, and practical considerations of implementing serverless applications in different programming languages, using the TLQ pipeline as a case study. Specifically, the research aims to explore the tradeoffs between Java and Python implementations on AWS Lambda, as well as the feasibility of using generative AI tools for multi-language pipeline development. The following research questions guide our analysis:

RQ-1: How does the implementation language (Java vs. Python) of AWS Lambda functions impact the performance and cost of the TLQ pipeline across its Transform, Load, and Query services, considering factors such as execution time, memory allocation, and AWS Lambda pricing?

RQ-2: How effective are generative AI tools in designing and implementing serverless pipelines across multiple languages, and what are the potential challenges and benefits of using these tools for real-world applications?

2 Case Study

2.1 Design Tradeoffs

In this project, two programming language implementations were tested for the TLQ pipeline: Java and Python. The goal was to assess their tradeoffs in terms of performance, cost implications, and the effectiveness of generative AI tools in code development.

Implementation A: Java

The TLQ pipeline was implemented entirely in Java, leveraging its strong performance in computational tasks and robustness in handling complex workflows. Java's static typing and runtime optimizations were expected to result in better performance, particularly for the Transform service's computationally intensive operations. However, Java's larger deployment package size and higher cold start latency in AWS Lambda could increase costs and affect response time.

Implementation B: Python

The TLQ pipeline was also implemented in Python, chosen for its simplicity, fast development cycles, and compatibility with serverless platforms. Python's lightweight deployment packages were expected to reduce cold start times and hosting costs, making it well-suited for the stateless, event-driven nature of AWS Lambda functions. However, Python's performance in compute-heavy tasks may lag behind Java, particularly for large datasets.

Code Generation with AI Tools

The effectiveness of generative AI tools, such as ChatGPT, in assisting with the implementation of both Java and Python pipelines was evaluated. These tools were used to generate portions of the pipeline's logic and configurations, offering faster prototyping and reducing development time. However, differences in language-specific outputs and the need for manual adjustments to generated code were anticipated to vary between Java and Python.

2.2 Serverless Application Implementation

The TLQ data pipeline was implemented as a serverless architecture designed for flexibility, scalability, and efficiency. AWS Lambda was used to execute the Transform, Load, and Query services, with Amazon S3 providing persistent storage for raw and transformed CSV files and SQLite databases. Each service was implemented in both Python and Java to evaluate the tradeoffs between runtime performance, memory usage, and development complexity. The pipeline employs a stateless architecture, with intermediate data stored in S3 and exchanged between services through JSON payloads containing file location metadata. This design supports synchronous execution, where services process data sequentially to ensure predictable and orderly workflows. Additionally, asynchronous execution is possible, leveraging S3 as an intermediary to allow services to operate independently, enabling parallel processing and scalability for larger datasets or concurrent workloads.

2.3 Experimental Approach

To evaluate the performance and cost trade-offs of our multi-language Transform-Load-Query (TLQ) serverless pipeline, we conducted a series of controlled experiments on AWS Lambda. The primary objective was to compare throughput, runtime, and cost implications of implementing identical functions in Java and

Python, while assessing how various configurations influenced the end-to-end workflow.

Client Setup

We generated test traffic from a local client environment using custom Bash scripts. These scripts automated the invocation of the three-function sequence—Transform (T), Load (L), and Query (Q)—in both Java and Python implementations. By controlling batch executions, and repeating tests systematically, we ensured consistent and reproducible conditions. All code and scripts were maintained in a version-controlled repository to enable future replication and validation.

Server-Side Infrastructure and Configuration

Our serverless application was deployed in the us-east-1 region to maintain cost-effectiveness and reduce variability. Each Lambda function was implemented twice, once in Java and once in Python, executing identical logic on identical datasets. Memory was fixed at 256 MB per function to isolate runtime differences from memory-based effects. Timeouts were set to two minutes as a safeguard against premature termination, though actual runtimes were typically shorter. We used default networking configurations without a custom VPC and deployed all resources within the same region to ensure consistent latency and focus on language-driven differences.

Data Sets and Workloads

To capture a range of operational scenarios, we tested CSV datasets varying from 100 to 100,000 rows, with primary emphasis on mid-sized sets of 5,000 and 10,000 rows. Running each scenario in groups of ten allowed for statistically meaningful results, while using identical datasets for both languages ensured fairness. We evaluated both cold and warm start conditions by adjusting idle times between runs to understand initialization overhead.

Metrics and Data Collections

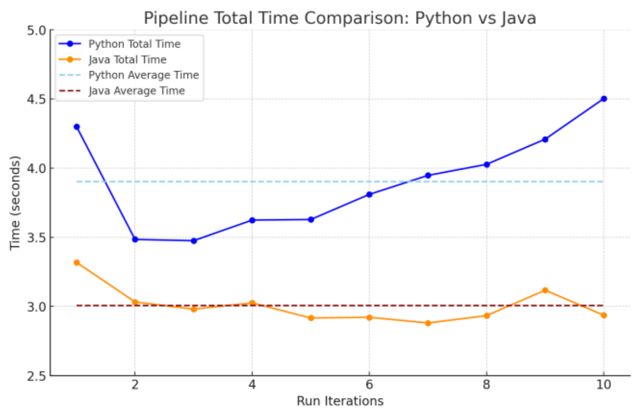
We measured key metrics such as runtime and round-trip time to understand how language choice and configuration affected performance. Metrics were gathered using SAAF's Inspector class, which offered a structured and repeatable mechanism for data collection. Cost implications were estimated by applying AWS Lambda's pricing model to scenarios with 10,000 or 100,000 full pipeline executions, offering insights into the financial impact of scaling these workloads.

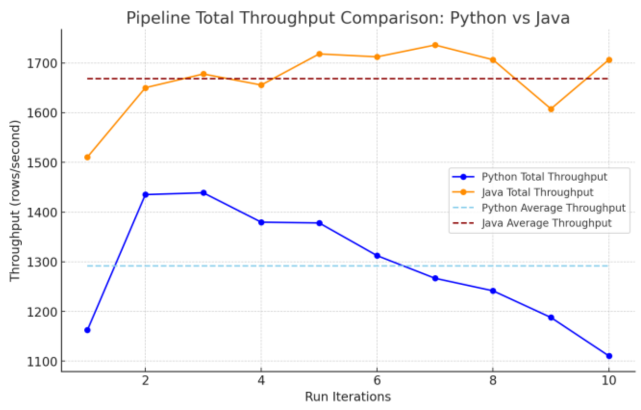
Reproducibility and Scripts

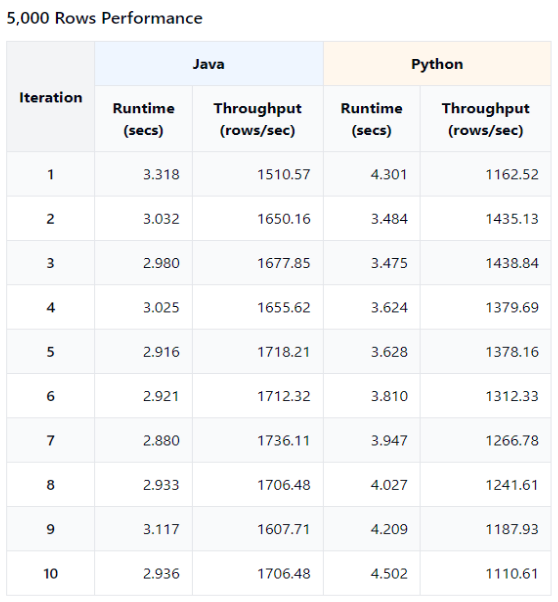
All test procedures were automated and documented, with scripts and detailed instructions available in a version-controlled repository.

3 Experimental Approach

Comparison: 5,000 Rows

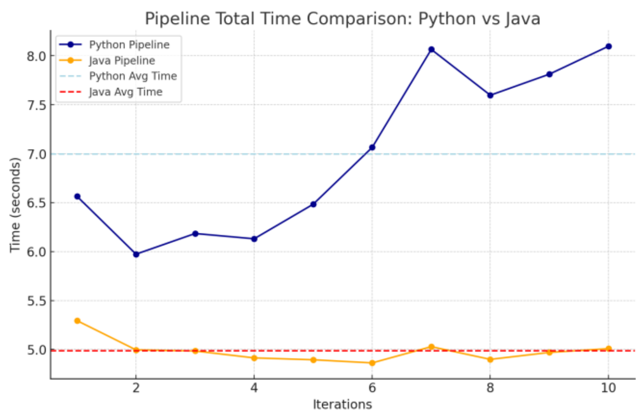


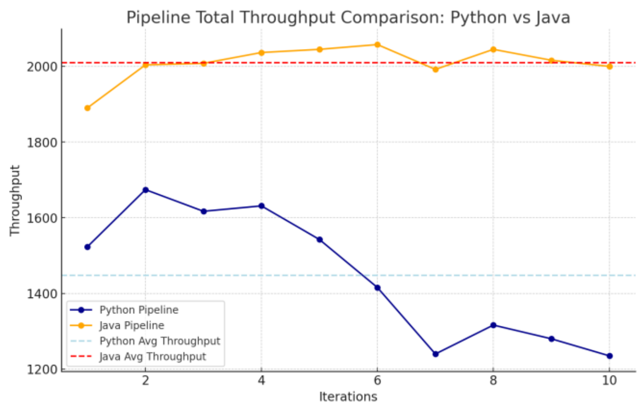


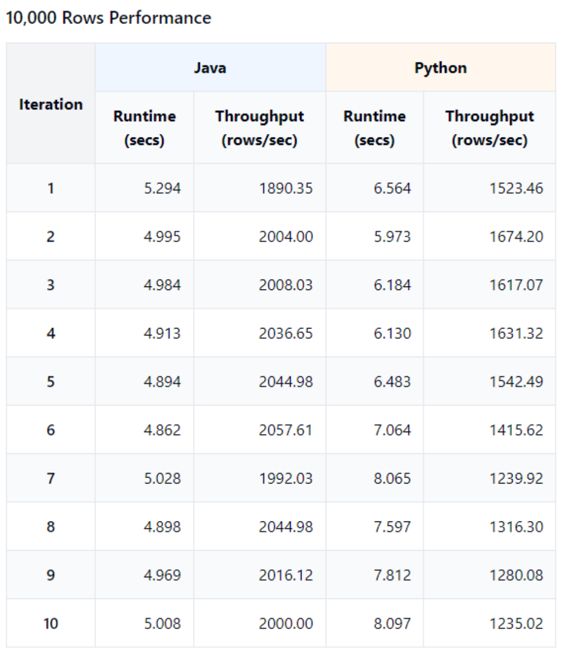

5,000 Rows Performance

Iteration	Java		Python	
	Runtime (secs)	Throughput (rows/sec)	Runtime (secs)	Throughput (rows/sec)
1	3.318	1510.57	4.301	1162.52
2	3.032	1650.16	3.484	1435.13
3	2.980	1677.85	3.475	1438.84
4	3.025	1655.62	3.624	1379.69
5	2.916	1718.21	3.628	1378.16
6	2.921	1712.32	3.810	1312.33
7	2.880	1736.11	3.947	1266.78
8	2.933	1706.48	4.027	1241.61
9	3.117	1607.71	4.209	1187.93
10	2.936	1706.48	4.502	1110.61

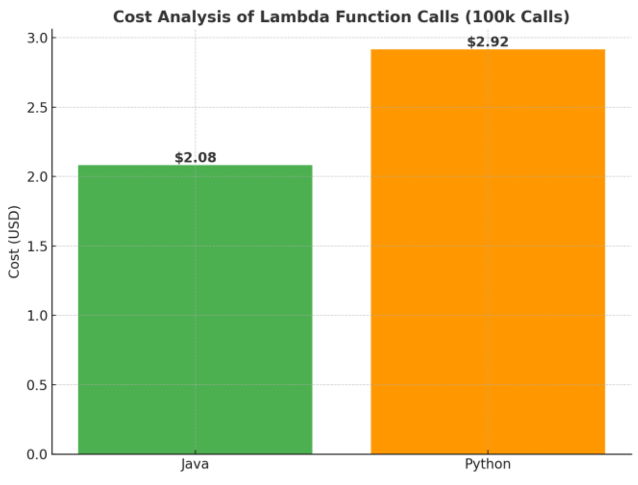
Comparison: 10,000 Rows






10,000 Rows Performance

Iteration	Java		Python	
	Runtime (secs)	Throughput (rows/sec)	Runtime (secs)	Throughput (rows/sec)
1	5.294	1890.35	6.564	1523.46
2	4.995	2004.00	5.973	1674.20
3	4.984	2008.03	6.184	1617.07
4	4.913	2036.65	6.130	1631.32
5	4.894	2044.98	6.483	1542.49
6	4.862	2057.61	7.064	1415.62
7	5.028	1992.03	8.065	1239.92
8	4.898	2044.98	7.597	1316.30
9	4.969	2016.12	7.812	1280.08
10	5.008	2000.00	8.097	1235.02



Cost = Runtime (s) x Memory (GB) x Number of Calls x Cost/GB-Second

4 Conclusions

The comparative analysis of the Transform-Load-Query (TLQ) pipeline implementations in Python and Java provides valuable insights into the trade-offs between these programming languages in the context of serverless computing. Through our experimentation and data visualization, we have observed clear differences in performance metrics, scalability, and potential cost implications. These observations will guide the design and optimization of serverless applications leveraging AWS Lambda.

Performance Differences

Our results reveal that Java consistently outperforms Python in terms of execution time and data throughput. For instance, in the 5,000-row dataset tests, Java's average runtime remained stable at around 3 seconds, while Python's runtime increased beyond 4 seconds for larger input sizes. Similarly, Java demonstrated higher throughput rates, exceeding 1,700 rows/second in most iterations, while Python's throughput consistently lagged, declining significantly as dataset sizes increased. These results suggest that Java's optimizations and lower overhead provide a significant advantage for time-sensitive workloads.

Scalability and Consistency

The scalability tests further highlighted Java's superior performance consistency. For the 10,000-row datasets, Java's throughput remained close to 2,000 rows/second, even under varying input loads, while Python experienced a notable decline in throughput as dataset sizes increased. Java's steady performance across multiple iterations demonstrates its reliability for handling larger-scale workloads.

Cost Implications

A detailed cost analysis of our 10,000 row results further underscores the efficiency of Java over Python. Using the AWS

Lambda pricing model, we calculated the cost of processing 100,000 function calls with a 256 MB memory allocation. Java incurred an approximate cost of $2.08, while Python's cost was $2.92, representing a 28.5% cost savings for Java. This reduction is attributed to Java's shorter runtimes and higher throughput, which reduce the execution time billed by AWS Lambda. These findings are visually represented in our cost analysis bar graph for 100,000 function calls, accompanied by the formula:

Cost = Runtime (s) x Memory (GB) x Number of Calls x Cost/GB-Second.

Developmental Tradeoffs

While Java excels in runtime performance, Python remains advantageous in terms of ease of development and tool compatibility. Our exploration of tools like ChatGPT, Claude, and Copilot highlighted Python's suitability for rapid prototyping and iterative development, especially for teams prioritizing productivity over raw performance. These tools also demonstrated utility in translating serverless functions across languages, streamlining the implementation process for multi-language pipelines.

Recommendations and Future Work

Based on our findings, we recommend considering Java for applications requiring high throughput and low latency, particularly when cost-efficiency is critical for large-scale deployments. Python, on the other hand, is better suited for rapid development cycles and scenarios where ease of use outweighs performance considerations.

Future work will include:

1. Conducting detailed cost analysis to quantify potential savings.

2. Evaluating the impact of cold starts on performance for both languages.

3. Testing additional programming languages to broaden the comparative analysis.

4. Exploring hybrid approaches that leverage the strengths of both Java and Python within a single pipeline.

By implementing these recommendations and continuing our analysis, we aim to provide a comprehensive guide for optimizing serverless applications using multi-language pipelines. These insights will be invaluable for organizations seeking to maximize the performance, scalability, and cost-effectiveness of their AWS Lambda deployments.