Evaluating Programming Language Efficiency for Serverless TLQ Data Pipelines on AWS Lambda

Performance and Cost Analysis of Multi-Language Implementations on AWS Lambda

Kobe Benavente
School of Engineering & Technology
University of Washington
Tacoma, Washington, USA
kobeb@uw.edu

William Hay
School of Engineering & Technology
University of Washington
Tacoma, Washington, USA
whay@uw.edu

William Chhiv
School of Engineering & Technology
University of Washington
Tacoma, Washington, USA
wchhiv@uw.edu

ABSTRACT

This paper compares Python, Node.js, and Java for implementing Transform–Load–Query (TLQ) data processing pipelines on AWS Lambda. Using a dataset of approximately 551,000 Spotify tracks, we implement identical pipeline stages in each language and evaluate execution performance, cold start behavior, memory utilization, and estimated hosting cost. Results show that no single language consistently outperforms the others across all pipeline stages. Python demonstrates efficient memory usage and strong performance in data transformation and loading tasks, Node.js achieves the highest overall throughput due to fast query execution, and Java provides competitive query performance but incurs higher runtime and memory overhead during data loading. Overall, execution duration and memory allocation were stronger deciders of cost than cold start latency. These findings highlight that language selection for serverless TLQ pipelines should be guided by workload characteristics and stage-specific performance requirements rather than a single performance metric.

CCS CONCEPTS

- Computer systems organization → Cloud computing
- Software and its engineering → Software performance

KEYWORDS

Serverless, AWS Lambda, FaaS, Python, Node.js, Java, performance comparison, TLQ, efficiency, transform, load, query, Spotify

1 Introduction

Serverless computing has grown increasingly common for deploying cloud applications as it offers automatic scaling, pay-per-use pricing, and the removal of server management overhead. As part of this shift, AWS Lambda, the leading Function-as-a-Service (FaaS) platform, provides support for multiple programming languages such as Python, Node.js, and Java [1,2]. However, language choice can have varying impacts on application performance, and it can be challenging for developers to decide which language to use when building cloud-based systems. Factors such as cold-start latency, memory consumption, and overall hosting cost all influence this decision. Therefore, understanding these trade-offs is essential for developers designing serverless data processing pipelines.

This paper presents a comparative study of programming language performance for a Transform-Load-Query (TLQ) data processing pipeline implemented on AWS Lambda. TLQ pipelines are commonly used in scenarios where large datasets need to be cleaned, organized, stored, and later analyzed. For example, preparing data for reporting dashboards, sorting and categorizing information for recommendation systems, or structuring datasets so applications can quickly search and filter them. Our implementation processes a sizable Spotify music dataset, with each record including detailed artist, song, and album information. The pipeline performs data transformation, database loading, and analytical queries across three independent Lambda functions implemented in Python, Node.js, and Java.

Our experimental evaluation measures runtime performance, cold start overhead, memory utilization, and estimated hosting costs for each language implementation. By holding the application logic, dataset, memory

configuration, and cloud infrastructure constant, we isolate the programming languages as the independent variable to allow for direct comparison.

1.1 Research Questions

RQ-1: How does programming language choice (Python, Node.js, Java) impact the execution performance and throughput of serverless functions in a TLQ data processing pipeline?

RQ-2: How do cold start initialization times and memory utilization vary across programming languages on AWS Lambda, and what are the resulting implications for hosting costs?

2 Case Study

Our study's TLQ pipeline processes a publicly available Spotify dataset from Kaggle.com containing approximately 551,000 music tracks. The raw dataset includes 39 columns per record, covering artist names, song titles, album information, genre classifications, and audio features such as tempo, loudness, danceability, energy, and popularity scores.

The pipeline consists of three Lambda functions that execute sequentially. The Transform function reads raw CSV data from S3, normalizes the schema, and performs data enrichment operations such as parsing durations, categorizing numeric values into labeled tiers, and cleaning text fields. The Load function retrieves the transformed data and batch-inserts records into a relational database. The Query function provides an interface for analytical queries, supporting operations such as aggregations, filtering, and ranking over the loaded data.

Serverless execution works well for this type of workload because the pipeline processes a large dataset in separate stages rather than running all the time. Each stage can be triggered on its own, tested independently, and scaled automatically by AWS Lambda based on how much data needs to be processed. This structure also makes it easier to compare programming languages fairly, since the logic, dataset, and cloud setup stay the same across all implementations, and only the language itself changes.

2.1 Design Tradeoffs

Designing a serverless TLQ pipeline on AWS Lambda involves several tradeoffs related to language choice, pipeline structure, and execution flow. While many architectural options are available, this project prioritizes comparing programming languages in a realistic serverless data processing setting. As a result, most system components were kept consistent across implementations, and differences were limited to choices that directly affect how each language behaves in a serverless environment.

A key tradeoff is the choice of programming language. Python, Node.js, and Java were selected because they are commonly used with AWS Lambda but differ in runtime characteristics. Python and Node.js are generally lighter-weight and associated with faster startup times, while Java relies on the Java Virtual Machine (JVM), which introduces additional initialization overhead but may offer stronger performance during longer executions [4,5]. These differences represent a tradeoff between startup latency and potential compute efficiency.

Another design tradeoff involves how the pipeline is structured. Instead of putting all TLQ stages into one Lambda function, the pipeline was split into separate Transform, Load, and Query functions. This makes the pipeline easier to understand, test, and change if needed. The downside is that having more functions means more Lambda invocations and a higher chance of cold starts, but this approach was chosen because it is common in real serverless applications and makes it easier to see how each language performs at different stages.

Execution flow also comes with tradeoffs. The pipeline runs in a sequential order, meaning each stage finishes before the next one starts. While running stages in parallel or using orchestration tools could improve performance, those options would add more complexity and make it harder to isolate the effects of programming language choice. Using a simple sequential flow helps keep behavior predictable and makes performance differences easier to compare.

Finally, using a relational database introduces tradeoffs in a serverless setup. Relational databases are useful for storing structured data and supporting analytical queries, but connecting to a database can add extra delay, especially during cold starts. This matters when comparing languages, since some runtimes may take longer to start up and connect to external services than others.

2.2 Serverless Application Implementation

Turning to the implementation details of the serverless application, our systems use Amazon S3 for intermediate file storage and AWS RDS with MySQL for persistent data storage. Our applications are organized as three stateless functions corresponding to different stages of the workflow, with persistent state maintained entirely through external

storage services rather than in-memory or session-based mechanisms.

The Transform function reads a raw CSV file from Amazon S3 and converts them into a normalized format suitable for database insertion. During transformation, text fields are cleaned and standardized, numeric values are safely parsed, durations are converted into minutes, and additional categorical labels are derived from numeric attributes. The transformed output is written to a new CSV file using the Lambda temporary storage directory and then uploaded back to S3. The Node.js implementation processes data incrementally using streams to minimize memory usage, while the Python implementation reads the file into memory before processing. This difference reflects common patterns in each language rather than a deliberate design choice. Python's standard CSV library is designed around in-memory iteration, while Node.js's ecosystem favors stream-based processing for handling large files. While the underlying approach differs, this does not compromise the validity of the resulting comparison. Both implementations read the same input, apply the same transformation logic, and produce identical output schemas. The difference in how each language handles file I/O is itself part of what makes a language comparison meaningful, developers choosing a language for a real-world pipeline would encounter these same patterns and tradeoffs.

The Load function retrieves the transformed CSV file from S3 and inserts the records into a MySQL relational database hosted on AWS RDS. Each implementation connects to a single database instance designed for single-user operation and does not track users or maintain session state. Before inserting records, the Load function ensures that the required database table exists and truncates any existing data to ensure a clean state for each pipeline execution. Data is inserted using batched SQL operations with a batch size of 1000 rows to reduce database overhead and improve insertion efficiency. Both implementations use the same batch size to ensure consistency across the comparison. The Python implementation uses the pymysql library with explicit transaction control, while the Node.js implementation uses the mysql2/promise client with async/await for database interactions. After insertion completes, the database connection is closed, and all persistent state is stored in the database.

The Query function provides read-only access to the stored data through a set of predefined SQL queries. These queries support operations such as grouping records by artist, computing average metrics by category, filtering records based on threshold values, and comparing statistics between explicit and clean tracks. Each query invocation establishes a new database connection, executes the requested SQL statement, and returns results as structured JSON. While Python and Node.js differ in their database client libraries and execution models, both implementations follow the same query logic and return equivalent results.

Data is passed between services using persistent storage rather than direct in-memory communication. The Transform function outputs its results to Amazon S3, which serves as the handoff point to the Load function. The Load function writes transformed records into the MySQL database, which is then accessed by the Query function. The application is not designed to support multiple users or independent user sessions, and each deployment operates on a single dataset using one database instance. This design keeps the implementation simple while clearly defining how state and data flow are managed throughout the application.

Each team member maintained their own isolated infrastructure, including separate S3 buckets and RDS MySQL instances, to develop and test their language implementation independently. However, all deployments used consistent configurations, including the same AWS region, memory allocation, and database schema, to ensure that performance differences could be attributed to language choice rather than infrastructure variation.

To support development consistency and reduce implementation drift across languages, large language models (LLMs) were used as a development support tool. More specifically, AI assistance was used to help draft SQL queries, verify that transformation and query logic matched as closely as possible between Python, Node.js, and Java implementations, and assist in diagnosing debugging issues during development. The use of AI was limited to code generation support, clarification of library usage, and error interpretation; all final implementation decisions, testing, and validation were performed by the authors. This approach helped ensure functional equivalence across implementations while allowing the study to focus on language-level performance characteristics rather than differences introduced by developer error or incorrect logic.

2.3 Experimental Approach

For our experimental approach, each TLQ stage was tested using manual invocations through the AWS Lambda console. Because of time constraints, each configuration was run once rather than multiple times, and results were not averaged. Both cold start and warm start executions were recorded. Cold starts were measured by invoking a function after it had been idle, while warm starts were

measured by invoking the function again immediately afterward using the same execution environment.

The Transform and Load functions were configured with 1024 MB of memory, while the Query function was configured with 128 MB of memory. Lambda timeout values differed between functions based on expected workload length, but all executions completed well within their configured limits. As a result, timeout settings did not affect the observed performance. All functions were deployed without a VPC and executed in the us-east-1 AWS region within a single availability zone. The default x86_64 architecture was used for all Lambda functions to keep the runtime environment consistent.

Although our Query function code supports multiple predefined analytical queries, only one was used for evaluation. Specifically, the tested query returned the top 10 artists by popularity, grouping records by artist and ranking them based on an aggregated popularity metric. This query was chosen because it involves database access, aggregation, sorting, and result formatting, making it suitable for comparing query performance across languages. The remaining query options were not included in the performance measurements.

Performance data was collected from AWS CloudWatch Logs using the REPORT entries generated by AWS Lambda [3]. These logs provided execution duration, billed duration, initialization time for cold starts, and maximum memory usage. Hosting costs were calculated using AWS Lambda's published pricing for the selected region, calculated as *(Memory in MB / 1024)* × *Billed Duration* × *Price per GB-second*.

This approach allows the experiments to be repeated using the same dataset and configuration, while acknowledging that results may vary due to the use of single-run measurements.

3 Experimental Results

RQ-1: How does programming language choice (Python, Node.js, Java) impact the execution performance and throughput of serverless functions in a TLQ data processing pipeline?

Evaluating our results, we can see that there is no single best language for all stages of the TLQ pipeline. Different languages excel at different stages depending on the workload characteristics. In the Transform stages seen in Figure 1, Python and Java performed comparably in warm conditions (84,525 ms and 85,585 ms respectively), while Node.js lagged behind at 99,071 ms, approximately 14,500 ms slower. This suggests that Node.js's stream-based processing approach, while memory-efficient, introduces additional overhead for CPU-intensive data transformation operations.

In the Load stage, the performance gap shifts dramatically as seen in Figure 2. Python and Node.js completed database insertions in similar timeframes (34,226 ms and 33,477 ms warm), while Java required 54,215 ms, nearly 60% longer than the other implementations. This indicates that Java's database client library introduces substantial overhead when handling batch insertions of 551,000 records.

The Query stage produced the most dramatic differences. Node.js completed the analytical query in 3,186 ms warm, with Java close behind at 3,526 ms. Python, however, required 22,948 ms for the same operation, roughly 7x slower than the other two languages. Given that each configuration was tested with a single run rather than multiple trials, this disparity warrants further investigation. The difference could be attributed to Python's pymysql library overhead, variability in AWS RDS connection latency, or other environmental factors that a single measurement cannot isolate. Additional testing with multiple runs and averaged results would be necessary to determine if this performance gap is consistent.

When aggregating warm execution times across all three stages, Node.js achieved the fastest total pipeline throughput at 135,734 ms, followed by Python at 141,699 ms and Java at 143,326 ms. Node.js overcame its slower Transform performance through exceptional Query efficiency, while Python and Java showed inverted strengths. Python excelled at Transform and Load but struggled with Query, whereas Java demonstrated strong Query performance but significantly slower Load operations. These findings suggest that language selection for serverless TLQ pipelines should be guided by which stage dominates the workload rather than assuming any single language will outperform across all operations.

RQ-2: How do cold start initialization times and memory utilization vary across programming languages on AWS Lambda, and what are the resulting implications for hosting costs?

Cold start behavior and memory utilization varied across Python, Node.js, and Java, with measurable implications for execution latency and estimated hosting cost. Contrary to our prior assumptions about Java on AWS Lambda, our results do not show Java consistently incurring the highest cold start initialization overhead. Instead, initialization behavior varied by pipeline stage.

In the Transform and Load stages, Java exhibited the lowest cold start initialization times among the three languages, with initialization durations of 515 ms and

323.57 ms, respectively. Python and Node.js both showed higher initialization costs in these stages, particularly Python in the Transform stage (809 ms) and Node.js in the Load stage (768 ms). In the Query stage, however, Java recorded the highest initialization time at 515 ms, while Python and Node.js initialized more quickly. These results indicate that cold start initialization time alone does not follow a consistent language-based pattern and can vary depending on workload and runtime behavior.

While Java often demonstrated favorable initialization times, this did not always translate into lower total cold execution duration. In the Load stage, Java's overall execution time was significantly longer than Python and Node.js despite its lower initialization cost, suggesting that runtime execution and database client overhead had a greater impact on performance than cold start initialization alone. This highlights that initialization time represents only one component of cold start cost and should not be interpreted on its own.

Memory utilization showed clearer and more consistent differences across languages. In the Transform stage, Node.js consumed substantially more memory (530–551 MB) than Python (188 MB) and Java (224–225 MB). In the Load stage, memory usage increased across all languages due to batch database insertion, with Java using the most memory (964–965 MB), followed by Node.js (761 MB) and Python (602–605 MB). In the Query stage, overall memory consumption was lower due to the reduced workload and smaller memory allocation (128 MB), though Java still used more memory than Python and Node.js.

These differences in execution time and memory usage directly affected estimated hosting costs. Because AWS Lambda pricing is determined by allocated memory and billed execution duration, longer runtimes generally had a larger impact on cost than cold start initialization time. In the Transform stage, Node.js incurred the highest estimated cost per run due to longer execution time, despite its efficient streaming model. In the Load stage, Java was the most expensive option, with nearly double the cost of Python and Node.js, driven primarily by its longer execution duration. In the Query stage, Python's significantly slower execution resulted in the highest cost per run, while Node.js achieved the lowest cost due to fast query completion and modest memory usage.

Overall, the results show that cold start initialization time alone is not a reliable predictor of serverless performance or cost. Java demonstrated competitive and sometimes lower initialization overhead, but higher runtime and memory costs in certain stages reduced its cost efficiency. Python benefited from lower memory usage but became costly when execution time increased. Node.js traded higher memory consumption for faster execution in I/O and query-heavy workloads, often resulting in lower overall cost. These findings emphasize that language selection for AWS Lambda should consider the full interaction between initialization time, runtime performance, and memory utilization rather than relying on generalized assumptions about cold start behavior.

Metric	Python (Cold)	Python (Warm)	Node.js (Cold)	Node.js (Warm)	Java (Cold)	Java (Warm)
Duration (ms)	92,498	84,525	101,577	99,071	90137	85585
Billed Duration (ms)	93,307	84,526	102,325	99,072	90653	85586
Init Duration (ms)	809	N/A	747	N/A	515	N/A
Max Memory Used (MB)	188	188	530	551	224	225
Max Memory Allocated (MB)	1024	1024	1024	1024	1024	1024
Estimated Hosting Cost Per Run	$0.00156	$0.00141	$0.00171	$0.00165	$0.00151	$0.00143

Figure 1: Comparison of Transform function performance across programming languages.

Metric	Python (Cold)	Python (Warm)	Node.js (Cold)	Node.js (Warm)	Java (Cold)	Java (Warm)
Duration (ms)	30,923	34,226	31,629	33,477	60,768	54,215
Billed Duration (ms)	31,294	34,226	32,398	33,478	61,092	54,215
Init Duration (ms)	370.68	N/A	768	N/A	323.57	N/A
Max Memory Used (MB)	602	605	761	761	964	965
Max Memory Allocated (MB)	1024	1024	1024	1024	1024	1024
Rows Inserted	551,449	551,449	551,443	551,443	551,443	551,443
Estimated Hosting Cost Per Run	$0.0005216	$0.0005704	$0.00054	$0.00056	$0.00102	$0.0009

Figure 2: Comparison of Load function performance across programming languages.

Metric	Python (Cold)	Python (Warm)	Node.js (Cold)	Node.js (Warm)	Java (Cold)	Java (Warm)
Duration (ms)	26813	25509	4542	3186	17827	3526
Billed Duration (ms)	27017	25510	4796	3187	18507	3257
Init Duration (ms)	203	N/A	253	N/A	515	N/A
Max Memory Used (MB)	50	50	99	99	224	225
Max Memory Allocated (MB)	128	128	128	128	128	128

Figure 3: Comparison of Query function performance across programming languages.

4 Conclusions

This study evaluated Python, Node.js, and Java for implementing a Transform–Load–Query (TLQ) pipeline on AWS Lambda and found that language choice meaningfully affects performance and cost in a stage-dependent manner. Rather than a single language coming out as superior, each runtime proved more suitable for different parts of the pipeline. Python aligned well with transformation and loading workloads due to its balanced execution time and lower memory usage, Node.js performed best in query-heavy scenarios with low latency, and Java showed competitive query performance but higher execution and memory costs during data loading. These results suggest that effective serverless pipeline design depends more on matching language characteristics to workload demands than on selecting a single “best” runtime.

Our results provide practical insight into how language-level tradeoffs manifest in a realistic serverless TLQ pipeline. Future work could address some of our shortcomings by repeating experiments across multiple runs, varying memory allocations, and testing additional workloads or storage backends to better understand how language choice interacts with scale, concurrency, and cost in production serverless systems. However, we still feel that our research provides an insightful starting point for the topic at hand.

REFERENCES

[1] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., ... & Suter, P. (2017). Serverless computing: Current trends and open problems. In Research advances in cloud computing (pp. 1-20). Springer, Singapore.

[2] Castro, P., Ishakian, V., Muthusamy, V., & Slominski, A. (2019). The rise of serverless computing. Communications of the ACM, 62(12), 44-54.

[3] Wang, L., Li, M., Zhang, Y., Ristenpart, T., & Swift, M. (2018). Peeking behind the curtains of serverless platforms. In Proceedings of the 2018 USENIX Annual Technical Conference (pp. 133-146).

[4] Manner, J., Endreß, M., Heckel, T., & Wirtz, G. (2018). Cold start influencing factors in function as a service. In 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (pp. 181-188).

[5] López, P. G., Arjona, A., Sampaio, A., & Mendonça, N. C. (2017). Serverless computing: A performance perspective. In Proceedings of the 10th International Conference on Utility and Cloud Computing (pp. 161-168).