The American Journal of Engineering and Technology
ISSN 2689-0984 | Open Access

TYPE Original Research
PAGE NO. 16-21
DOI 10.37547/tajet/Volume07Issue04-03

Check for updates

Startup Latency Analysis in Java Frameworks for Serverless AWS Lambda Deployments.

Maksimov Viacheslav Yurievich

Senior Software Engineer at AUTO1 IT Services SE & Co. KG, Germany, Berlin

OPEN ACCESS

SUBMITED 23 February 2025
ACCEPTED 25 March 2025
PUBLISHED 04 April 2025
VOLUME Vol.07 Issue04 2025

CITATION

Maksimov Viacheslav Yurievich. (2025). Startup Latency Analysis in Java Frameworks for Serverless AWS Lambda Deployments. The American Journal of Engineering and Technology, 7(04), 16–21. https://doi.org/10.37547/tajet/Volume07Issue04-03

Abstract: Cold start latency in serverless computing, particularly in Java-based AWS Lambda functions, presents a significant challenge for latency-sensitive applications. This study investigates the performance characteristics of three modern Java frameworks - Spring Boot, Micronaut, and Quarkus - deployed on AWS Lambda using the ARM64 (Graviton2) architecture. It evaluates cold start latency across three deployment configurations: managed runtime (with and without SnapStart) and GraalVM native images. Metrics were collected at varying memory allocations using Java 21. Results show that Quarkus consistently outperforms others in cold start latency on standard JVM, while SnapStart and GraalVM significantly reduce the number of cold starts and achieve sub-second latency, respectively. We discuss the implications of these findings for choosing a Java framework and runtime strategy on AWS Lambda, considering the trade-offs in deployment time, complexity, and performance. The paper concludes with recommendations for leveraging SnapStart and native images to mitigate cold start issues in Java serverless applications on ARM64.

Keywords: AWS Lambda, Cold start, SnapStart, GraalVM, ARM64, Java, Micronaut, Quarkus, Spring.

Introduction: Serverless computing platforms like AWS Lambda have revolutionized cloud application deployment by enabling automatic scaling and operational cost-efficiency. However, a major performance concern for serverless Java applications is the cold start latency, the time required for initializing a new function instance before processing its first request [1]. Cold starts are particularly problematic for Java workloads due to the substantial overhead of Java

Virtual Machine (JVM) initialization, class loading, and Just-in-Time (JIT) compilation, often resulting in delays several times longer compared to lighter runtimes such as Node.js or Python [2].

The recent availability of ARM64 (AWS Graviton2) architecture on AWS Lambda provides significant performance and cost advantages compared to traditional x86 processors, including improved computational efficiency and reduced execution costs [3]. Coupled with new AWS features like SnapStart, which allows Lambda functions to rapidly resume from pre-initialized JVM snapshots, Java's historical performance barriers are being reduced significantly. Another critical optimization is the use of GraalVM Native Images, which compile Java bytecode into native machine code ahead-of-time, dramatically reducing startup latency by eliminating JVM initialization entirely [4].

Choosing the appropriate Java framework can further influence cold start performance. Popular frameworks like Spring Boot offer extensive features but rely heavily on reflection and dynamic class loading, resulting in longer startup times. In contrast, newer frameworks such as Micronaut and Quarkus emphasize compile-time optimizations and minimal runtime reflection, significantly reducing initialization overhead.

This study provides a comprehensive comparison of cold and warm startup latencies for Spring Boot, Micronaut, and Quarkus frameworks on AWS Lambda's ARM64 architecture taking into account memory impact. Startups are examined using Java 21 across three execution environments: the standard managed Java runtime without SnapStart, the managed runtime with SnapStart enabled, and custom runtime deployments using GraalVM native images. Performance metrics were collected under controlled load conditions using Artillery, capturing median (p50), 90th percentile (p90), 99th percentile (p99), and maximum cold start latencies.

Through this investigation, trade-offs and practical implications for developers selecting Java frameworks and runtime strategies on AWS Lambda are intended to be highlighted. Understanding these dynamics is essential for optimizing serverless Java performance, ensuring responsiveness, minimizing costs, and effectively harnessing the benefits of ARM64 architecture in modern cloud-native applications.

MATERIALS AND METHODS

This study compared startup latency performance for Java functions using three Java frameworks with latest versions: Spring Boot (3.4.3), Micronaut (4.7.6), and Quarkus (3.18.4). All functions were deployed on AWS Lambda across three distinct runtime configurations:

- Managed Java Runtime (without SnapStart): Functions were packaged as JAR files and deployed directly to AWS Lambda's standard Java managed runtime (Amazon Corretto JDK).
- Managed Java Runtime (with SnapStart): Functions deployed as JAR files using AWS Lambda SnapStart, a feature enabling JVM state snapshotting after initialization, significantly reducing cold start latency.
- Custom Runtime (GraalVM Native Images): Java functions compiled to native ARM64 executables using GraalVM Native Image (version 23.1), deployed as Lambda custom runtimes, eliminating JVM overhead entirely.

The test functions implemented a lightweight API endpoint retrieving a record from Amazon DynamoDB by ID, simulating a common serverless use case involving moderate I/O operations. The deployed functions were load-tested using Artillery, an open-source load-testing tool, which generated controlled, sequential requests ensuring accurate cold and warm start measurements.

Cold starts were consistently reproduced by deploying new function versions or invoking each function after sufficient idle time, ensuring the AWS Lambda environment initialized new containers for each measurement. For each test scenario, approximately 100 cold start invocations were executed to reliably capture statistical distributions. Subsequent warm invocation tests involved rapid sequential calls to warmed Lambda containers, providing baseline performance metrics without initialization overhead.

Testing was performed at three memory allocations: 512 MB, 1024 MB, and 2048 MB, assessing how CPU resources allocated to each Lambda instance affected startup performance. Detailed latency metrics were extracted from AWS CloudWatch Logs and AWS CloudWatch Insights, specifically capturing initialization durations for cold starts and request latencies for warm executions.

Data analysis involved statistical summarization of captured startup latencies across memory sizes and frameworks, clearly illustrating performance impacts of runtime configurations, memory allocation, and Java versions.

RESULTS AND DISCUSSION

Table 1 summarizes the cold start latency metrics for each framework under each runtime configuration at 1024 MB memory (using Java 21). This gives a representative comparison in a mid-memory setting. Cold start time is measured from invocation to the function's first response.

Framework	Runtime	p50 (ms)	p90 (ms)	p99 (ms)	Max (ms)
Spring Boot	No SnapStart	4974	5249	5354	5698
Spring Boot	SnapStart	978	1177	1231	1682
Spring Boot	GraalVM	807	940	989	1156
Micronaut	No SnapStart	4102	4747	4786	4913
Micronaut	SnapStart	876	1012	1123	1450
Micronaut	GraalVM	753	930	956	1294
Quarkus	No SnapStart	2864	3163	3224	3551
Quarkus	SnapStart	534	745	878	1027
Quarkus	GraalVM	503	698	759	1365

Table1. Cold start latency (ms) distribution for each framework and runtime at 1024 MB memory (ARM64, JDK 21)

Effectiveness of AWS Lambda SnapStart

When AWS Lambda's SnapStart feature was enabled, cold start latencies dramatically decreased across all three frameworks:

● Spring Boot showed the most pronounced improvement with SnapStart enabled, achieving median latencies of approximately 1 second, representing nearly a five-fold reduction compared to non-SnapStart deployments. However, tail latencies (p99 ≈ 1.23 seconds) suggest occasional delays in snapshot restoration, albeit still much improved compared to traditional JVM startups.

● Micronaut with SnapStart experienced median cold-start reductions to around 870 ms, roughly 4.5 times faster than without SnapStart. However, Micronaut exhibited slightly higher variability in snapshot restoration times compared to Quarkus.

● Quarkus, already optimized for faster JVM initialization, benefited notably from SnapStart, reducing median cold starts further to approximately 530 ms. Interestingly, the difference between Quarkus and other frameworks was reduced significantly due to SnapStart, suggesting the framework's inherent runtime optimizations provide diminishing returns once AWS-level optimizations like SnapStart are applied.

SnapStart primarily reduced cold starts by caching the JVM state after initialization. Snapshots restored faster than conventional JVM bootstraps, though residual overhead remained due to the snapshot restoration process and minor initialization of dynamic states (e.g., database connections).

Cold Start Performance with GraalVM Native Images

GraalVM native compilation showed the most impressive cold-start performance:

● Spring Boot Native significantly reduced its latency to approximately 800 ms median, a substantial improvement compared to JVM-based deployments. While still higher than Quarkus native, Spring Boot Native's performance demonstrates the significant potential for native image compilation to resolve cold start latency challenges for even traditionally heavyweight frameworks.

● Micronaut Native further reduced median latencies to around 750 ms, exhibiting minimal variance (p99 ≈ 950 ms). This efficiency emphasizes Micronaut's alignment with AOT compilation and native image methodologies.

● Quarkus Native displayed the lowest overall cold start latency, achieving 500 ms median, with exceptional consistency (p99 ≈ 750 ms). These findings match prior research demonstrating Quarkus's significant advantage when combined with GraalVM native images, providing near-instantaneous initialization suitable for latency-sensitive applications [5, 6].

Overall, native images produced a near-ideal performance scenario by eliminating JVM initialization entirely, substantially outperforming both standard and SnapStart-enabled JVM runtimes.

Impact of Memory Allocation

Framework	Runtime	512 MB	1024MB	2048 MB
Spring Boot	No SnapStart	6482	4974	3887
Spring Boot	SnapStart	1261	978	743
Spring Boot	GraalVM	876	807	713
Micronaut	No SnapStart	5006	4102	3251
Micronaut	SnapStart	1046	876	702
Micronaut	GraalVM	831	783	723
Quarkus	No SnapStart	3609	2864	2238
Quarkus	SnapStart	665	534	437
Quarkus	GraalVM	565	503	469

Table 2 Impact of Memory Allocation on Cold Start Latency for p50 (ms).

Table 2 depicts that memory allocation significantly impacted cold-start latency due to CPU provisioning proportional to allocated memory [7]. Increasing memory from 512MB to 2048MB showed a clear pattern of diminishing returns with or without SnapStart: approximately 20 to 30% when increasing to 1024MB (this threshold proved optimal for achieving acceptable performance at startup) and 18 to 24% when increasing further to 2048MB (while this was beneficial, cost effectiveness dropped off sharply beyond the 1024MB threshold).

GraalVM native images were notably less sensitive to memory variations, consistently maintaining sub-second latencies even at lower memory configurations (512 MB). This independence from CPU resources emphasizes native images as ideal candidates for applications requiring predictable low latency regardless of memory provisioning.

Warm Invocation Performance

Warm invocation performance, measured after initial container initialization, showed minimal latency across all frameworks and runtimes (median ~6–11 ms). Framework overhead differences were negligible once functions were warm. These results underline that serverless Java's primary challenge remains cold-start latency; once functions are warm, Java frameworks perform efficiently.

Discussion of Practical Implications

This comprehensive evaluation provides clear guidance for optimizing Java application deployments on AWS Lambda [8]:

- **Framework Selection:**

Quarkus consistently demonstrated the fastest cold-start performance, validating its suitability for latency-sensitive serverless deployments. Micronaut offered an intermediate balance of performance and simplicity, while Spring Boot, although slower initially, became competitive through SnapStart or native compilation.

- **Runtime Environment Choice:**

AWS Lambda's SnapStart provided an accessible, highly effective strategy for significantly reducing Java cold-start latencies (up to 5–10× improvement).

GraalVM native images delivered superior startup performance, consistently achieving sub-second cold-start latencies. For latency-critical workloads, native compilation offers the optimal solution, albeit with increased build complexity and potential compatibility constraints.

- **Resource Allocation Recommendations:**

Allocating at least 1024 MB of memory consistently mitigated severe Java cold-start penalties. Lower allocations (e.g., 512 MB or less) risked unacceptable latencies or initialization timeouts, particularly with heavier frameworks. Organizations should evaluate memory cost versus acceptable latency, noting diminishing returns beyond 1024 MB.

Trade-offs and Recommendations

Given these findings, several practical recommendations emerge:

- **SnapStart** is strongly recommended for Java Lambda functions requiring balance between simplicity and reduced startup latency. It particularly benefits heavyweight frameworks like Spring Boot, reducing cold-start delays sufficiently for most applications without extensive refactoring.

- **GraalVM Native Images** represent the highest-performing choice for latency-sensitive applications demanding consistently low startup latencies. Adoption of native images requires additional CI/CD effort and careful handling of dynamic Java features (e.g., reflection) [9]. Frameworks like Quarkus and Micronaut facilitate native compilation, providing a clear path to high performance.

- **Framework Choice** can remain flexible. With AWS optimizations (SnapStart, native compilation), previously heavy frameworks like Spring Boot become viable in serverless environments. Quarkus and Micronaut naturally align with serverless performance requirements, offering inherent advantages without extensive additional optimization [10].

Other noticeable recommendations:

Deployment Package Size Minimization:

The size of the deployment package directly impacts the cold start duration, particularly in scenarios involving AWS Lambda's Just-In-Time class loading model and I/O operations during cold initialization. Approaches: Avoiding unused dependencies and large transitive dependencies by manually managing the build. Performing code shrinking, obfuscation. Excluding test classes, logs, or documentation from the build artifacts.

Layered Deployments with AWS Lambda Layers

AWS Lambda Layers offer a method for separating common dependencies from the function's core codebase, enabling reuse and optimized cold starts, f.e. packaging shared dependencies (e.g., Apache Commons, Jackson, HTTP clients) into a Layer. A Lambda Layer is a ZIP archive that contains libraries, custom runtimes, or other dependencies. When configured, the Lambda execution environment mounts these layers into /opt, and the application can reference them via the classpath.

Provisioned Concurrency

Provisioned Concurrency is a native AWS Lambda feature that pre-warms a specified number of execution environments to eliminate cold start delays for incoming requests. This approach is particularly suitable for latency-sensitive or high-throughput Java functions.

Finally, avoiding reflection-heavy frameworks or choosing lightweight Dependency Injection (DI) frameworks can significantly reduce JVM and application boot time.

Limitations and Future Research

This study, while comprehensive, has limitations. The performance tests represented relatively simple I/O-bound functions (DynamoDB retrieval, without invocation priming). Results might differ for CPU-intensive or complex functions involving additional libraries or frameworks. Additionally, SnapStart and GraalVM limitations (e.g., dynamic feature restrictions) require further examination to assess compatibility with diverse workloads.

Future research should explore performance with more complex Java functions, new Java versions, examining JVM tuning for additional optimizations. Investigation into broader compatibility implications of GraalVM native images, as well as emerging AWS enhancements, could further refine Java serverless deployment strategies.

CONCLUSION

This study provided a detailed evaluation of startup latency characteristics for serverless Java applications deployed on AWS Lambda using the ARM64 (Graviton2) architecture. Three popular Java frameworks (Spring Boot, Micronaut, and Quarkus) were assessed across distinct runtime configurations: standard managed Java runtime (without SnapStart), managed runtime enhanced by AWS Lambda SnapStart, and custom runtime leveraging GraalVM native images. Systematic experiments using Java 21 and varying memory allocations (512 MB, 1024 MB, and 2048 MB) were conducted using controlled load tests via Artillery.

The findings revealed significant variations in cold-start performance among the frameworks under standard JVM deployment conditions. Quarkus consistently exhibited the lowest cold-start latencies, primarily attributed to its extensive use of ahead-of-time compilation and minimal reflection. Micronaut demonstrated intermediate performance, whereas Spring Boot experienced substantially higher latency, reflecting its reliance on runtime reflection and extensive dynamic configuration.

Introduction of AWS Lambda SnapStart markedly improved cold-start latencies for all frameworks. Particularly notable was the substantial reduction observed for Spring Boot, whose median latency decreased nearly fivefold, making it competitive with the other frameworks. Micronaut and Quarkus also benefited from SnapStart, though the relative improvement was less pronounced given their already optimized startup behavior. This indicates that SnapStart can substantially level performance differences, thus broadening framework selection based on criteria beyond cold-start performance alone.

GraalVM native image deployments further enhanced cold-start performance, virtually eliminating JVM initialization overhead and consistently achieving sub-second latency across all frameworks tested. Quarkus native images demonstrated the most impressive performance, reinforcing their suitability for ultra-latency-sensitive serverless applications. Micronaut and Spring Boot also showed significant latency improvements, highlighting the general applicability of native compilation to address Java’s startup latency challenges.

Memory allocation emerged as a crucial factor influencing startup latency, with performance notably improving when scaling memory from lower allocations to around 1024 MB. Beyond this threshold, further latency reductions were marginal, suggesting 1024 MB as an optimal configuration for balancing cost and performance.

In conclusion, the strategic application of AWS Lambda SnapStart and GraalVM native images effectively addresses Java’s inherent cold-start latency challenges in serverless computing environments. The results provide clear guidance on framework selection, runtime optimizations, and resource allocation strategies, establishing a foundation for further research into performance optimization methodologies and broader application compatibility in serverless Java deployments.

REFERENCES

Dittakavi, R.S.S. (2023). Cold Start Latency in Serverless Computing: Current Trends and Mitigation Techniques. Eduzone Journal, 12(2), 134-145.

Golec, M., Walia, G. K., Kumar, M., Cuadrado, F., Gill, S. S., & Uhlig, S. (2023). Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions. Journal of the ACM, 37(4), Article 111.

Poccia, D. (2021). AWS Lambda Functions Powered by AWS Graviton2 Processor. https://aws.amazon.com/blogs/aws/aws-lambda-functions-powered-by-aws-graviton2-processor-run-your-functions-on-arm-and-get-up-to-34-better-price-performance (Accessed 24 March 2025).

Kieselhorst, D, Schellhorn M. Serverless Architecture Conference (2024). Serverless Functions with GraalVM on AWS Lambda. https://serverless-architecture.io/blog/serverless_functions_with_graalvm_on_aws_lambda (Accessed 24 March 2025).

Hebbar, R., Chandran, M., and Sudevan, S. (2021). Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus. Journal of Cloud Computing Advances, Systems and Applications, 10(1), pp. 1-15.

Minic, B. (2021). Improving Cold Start Times of Java AWS Lambda Functions Using GraalVM. https://shinesolutions.com/blog/aws-lambda-graalvm-cold-start (Accessed 24 March 2025).

Smith, C. (2021). Optimizing Java Applications for Serverless Deployments. International Journal of Advanced Software Engineering, 25(3), pp. 155-170.

Pérez, D., Gómez, J., and Silva, M. (2023). Performance Best Practices Using Java and AWS Lambda. International Journal of Cloud Computing, 12(2), pp. 110-125.

Wang, S. (2021). Thin Serverless Functions with GraalVM Native Image. Serverless Computing Advances, 5(2), pp. 99-110.

Rao, R. (2024). AWS Lambda Cold Starts Explained: What They Are & How to Mitigate Them. Cloud Builder's Journal. https://www.ranthebuilder.cloud/post/is-aws-lambda-cold-start-still-an-issue-in-2024 (Accessed 24 March 2025).