PERFORMANCE OPTIMIZATION OF CLOUD-NATIVE JAVA SPRING BOOT MICROSERVICES USING GRAALVM NATIVE IMAGE TECHNOLOGY

Marko MARKOVIĆ

Ana VELJIĆ

Dušan RAJČEVIĆ

Abstract: *This research presents a comprehensive empirical analysis of performance optimization for Java Spring Boot microservices through the integration of GraalVM Native Image technology. The study focuses on the paradigm shift from traditional Just-In-Time (JIT) compilation to Ahead-of-Time (AOT) compilation to address the inherent challenges of "Cold Start" latency and high memory overhead in cloud-native environments. To evaluate these performance deltas, a specialized microservice, 'pioneer-service', was developed and benchmarked. The experimental results reveal a transformative improvement in system efficiency: startup latency was mitigated from 5.033 seconds to a mere 0.425 seconds (an 11.8x acceleration), while runtime memory consumption was reduced from 480 MB to 31 MB (a 93.5% reduction in RSS footprint). These findings validate that AOT compilation is a critical enabler for high-density deployment in resource-constrained, serverless, and auto-scaling architectures, effectively positioning Java as a top-tier contender for modern high-performance cloud infrastructure.*

Keywords: *GraalVM, Spring Boot, Native Image, AOT Compilation, Cloud-native, Performance Optimization.*

INTRODUCTION

Currently, the trends in the development of software applications, especially in the field of cloud-native and microservices architectures, dictate the necessity of meeting certain requirements in terms of resource optimization, startup time, and scalability of the developed applications. In this regard, it is necessary to highlight the fact that in cloud-native applications, where services are dynamically started and stopped in accordance with the workload, the latency of the initial start of the application may become a critical parameter of the system's performance. Java applications, which are based on the Java Virtual Machine (JVM), have demonstrated outstanding performance in terms of execution, but at the same time, they have a certain memory footprint and relatively slow startup times due to various mechanisms of dynamic class loading, JIT compilation, and runtime optimization. These limitations of Java applications in the context of cloud-native applications have become a certain challenge. As a result, the

development of GraalVM Native Image and Ahead-of-Time Compilation allows users to compile Java applications into native executable files, thereby achieving faster startup times and lower memory footprints (Preuveneers & Joosen, 2019). The main goal of this paper is to analyze the application of these technologies in Spring Boot applications in relation to the traditional JVM approach.

METHODOLOGY

The primary goal of this research is to conduct a comparative empirical analysis between the standard execution model of the Java Virtual Machine and the GraalVM Native Image model based on ahead-of-time compilation. To ensure the reliability and repeatability of the obtained results, a structured methodology was adopted that includes the standardization of the working environment, precise performance measurement, and rigorous diagnostic instrumentation. Such a controlled environment eliminated the variability in performance that could arise from differences in hardware resources or versions of software libraries. The application named pioneer-service is designed as a representative microservice that faithfully simulates scenarios from a real business environment.

Instead of a simple abstraction, the service includes a complete REST API layer, Jakarta Persistence specification for data management, and an internal mechanism for initial database population. Such complexity is necessary because it forces the GraalVM static analyzer to handle reflection, dynamic proxies, and database driver initialization during the build process, which are critical points in cloud-native applications (Hřivnáč, 2024). By using the H2 in-memory database version 2.2, performance measurement that includes Hibernate initialization has been enabled, ensuring that the results reflect a functional system ready for production use.

The evaluation process focuses on two primary performance indicators: latency during system startup and memory footprint measured thru Resident Set Size. The startup time is measured from the moment the binary file is executed until the point when the Spring context reports readiness for operation, which in the standard model includes class loading and JIT compilation, while in the native image it measures the current execution of machine code. The memory footprint was precisely recorded two seconds after the system stabilized using an internal diagnostic algorithm within the main class of the application (Khordadi, 2020). This algorithm uses the Runtime API to calculate actual memory consumption, and the data has been further verified with tools at the operating system level to confirm the empirical accuracy of the measurements.

The process of transforming the application into a native binary file followed a strict workflow based on the assumption of a closed world. The

source code was first compiled into standard Java bytecode, after which the Spring AOT mechanism scanned the application context to generate metadata about code accessibility. In the final phase, the GraalVM native image generation tool performed aggressive static analysis, removing all unused parts of the standard library and translating the remaining code into a standalone executable file specific to the operating system (Long et al., 2023). Each test was repeated ten times to eliminate background noise from the operating system, and the results were presented as the arithmetic mean of these measurements, thereby achieving a high scientific validity of the study.

Technological framework and comparison of approaches

Spring Boot 3.4 represents a modern platform for the development of microservices applications, with enhanced support for GraalVM and native images, which allows for more efficient deployment in cloud-native environments while maintaining the familiar programming model (Ye et al., 2024). GraalVM enables both JIT and AOT compilation, with the AOT approach generating native binary files with a pre-initialized heap, significantly reducing startup time (Šipek et al., 2019). The key difference between the JIT and AOT approaches lies in the method of optimization. JIT compilation performs optimizations during execution and allows for high performance in long-running processes, but with increased initialization time and greater resource consumption. On the other hand, AOT compilation allows for almost instant startup and a smaller memory footprint, but it requires explicit configuration for dynamic functionalities such as reflection (Wimmer et al., 2019; Yuhala et al., 2021). Research shows that the choice between these two approaches depends on the nature of the workload: JIT is more suitable for long-running systems, while AOT is more efficient in latency-sensitive and serverless scenarios (Ugwumba & Jaja, 2025). In the development and testing process, in-memory databases such as H2 are often used, which allow for simple and fast testing without external dependencies. This approach is particularly suitable in the context of native images, as it allows for reproducible performance measurements, including startup time and memory consumption (Belafia et al., 2021).

DESIGN AND ARCHITECTURE OF THE SYSTEM

The system architecture follows a set of principles regarding modularity and separation of concerns, facilitating easier maintenance, testing, and optimization of the application. Special attention has been paid to adapting to a cloud-native environment, where the main concerns include a low memory footprint and a stateless communication paradigm. The application has been created based on a custom Model View Controller (MVC) template,

optimized for RESTful web services (Wiewiórka et al., 2022). The architecture consists of three layers:

- **Model (Data Layer)** – defines the data structure and the entities that are persisted.
- **Data access layer (Repository Layer)** – abstracts communication with the database.
- **Web layer (Controller Layer)** – processes HTTP requests and generates JSON responses.

This division allows GraalVM to precisely identify all dependencies and execution paths during AOT analysis, thereby reducing the need for runtime mechanisms such as reflection and dynamic class scanning. The Jakarta Persistence (JPA) specification was used for data management, which enables object-relational mapping without the need for manually writing SQL queries. Within the system, the User entity has been implemented, with the basic attributes id and name. As a database, an H2 in-memory database was used, which allows for quick startup and execution of tests without external dependencies. This choice is particularly significant in the context of GraalVM Native Image technology, as it allows for the analysis of Hibernate's initialization time, which is one of the most demanding segments during AOT compilation. Communication with the system was implemented thru the REST architecture. An endpoint /user has been implemented, which supports the HTTP GET method for retrieving all users from the database.

The architecture is designed to enable GraalVM to statically discover all HTTP routes during the build process. This way, the need for runtime scanning of the annotations is removed, which is normally the standard practice for JVM applications, and latency for the first request is greatly reduced. The data flow of the application can be explained through the following steps (Armbruster et al., 2007):

1. The client sends an HTTP GET request to the application.
2. The Controller receives the request and forwards it to the Repository layer.
3. The Repository performs a query on the H2 database.
4. The results are mapped to User objects.
5. The runtime environment serializes objects into JSON and returns a response to the client.

This thread is optimized for AOT execution, ensuring minimal overhead during runtime.

Technical implementation and performance evaluation

The implementation of the proposed system is based on the modern Spring Boot 3.4 framework and a specific GraalVM distribution of Java 25. The primary engineering goal of this solution was the complete elimination of classical runtime bytecode interpretation and the transition to Ahead-of-Time (AOT) compilation. This approach allows for the translation of the application directly into native machine code specific to the operating system, thereby achieving maximum performance and minimal latency during execution.

The application is designed using the MVC (Model-View-Controller) architecture, which provides a clear segregation of responsibilities thru the following key components (Mohan & Goswami, 2025):

- Domain Model (User): This component represents the core of the data system. Using Jakarta Persistence (JPA) annotations, the model defines the data structure that is automatically mapped to a relational database. In the context of Native Image technology, this layer is critical because all reflections on entities must be defined during the compilation phase, thereby avoiding slow dynamic scanning at runtime.
- Repository layer: It enables the abstraction of data access thru the Spring Data JPA interface. This layer allows for the execution of complex CRUD operations without the need for manually writing SQL code. During the native build process, GraalVM performs static analysis of these interfaces to pre-generate the necessary proxy objects.
- Controller layer: Represents the external interface of the system and the primary entry point for HTTP requests. Through REST endpoints, this layer orchestrates communication between the user and the backend logic. Thanks to AOT optimization, the route mapping is fixedly defined in the binary file, resulting in an immediate response to the first received request.
- Main application (Entry Point): The central component that manages the startup and initialization of the entire Spring context. In addition to the standard startup logic, a Command Line Runner has been implemented that performs initial database population, thereby confirming the system's functionality immediately after a successful launch.

Special attention in this work is devoted to runtime diagnostics. Within the application itself, an algorithm for measuring actual memory consumption (Resident Set Size - RSS) has been implemented immediately after startup. This allows for direct, empirical comparison with the standard JVM environment. While the JVM requires a significant amount of RAM to operate

the virtualization layer itself, the native executable file eliminates this "overhead," allowing the application to run with minimal resources, which is critical for modern microservices architectures.

Implementation and Technical Analysis

The implementation of the pioneer-service is built upon the Spring Boot 3.4 framework and the GraalVM distribution of Java 25. The primary engineering objective of this system was the total elimination of runtime bytecode interpretation in favor of Ahead-of-Time (AOT) compilation (Oracle, 2026). By shifting the compilation process, the application is translated directly into OS-specific machine code, which results in peak execution performance and minimal latency.

The system architecture follows a strict separation of concerns, beginning with the Domain Model. The User.java class represents the core data structure, utilizing Jakarta Persistence (JPA) annotations to map Java objects to relational database tables. In a standard execution environment, the persistence provider would use reflection at runtime to scan for these entities, a process that significantly delays application readiness. However, within the GraalVM native image pipeline, this scanning occurs during the build phase. The following implementation demonstrates this entity structure.

```
package dev.quaestus.pioneer_service;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;

    public User() {}

    public User(String name) {
        this.name = name;
    }

    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

The Data Access Layer is managed via the UserRepository interface, which extends JpaRepository. This abstraction allows for complex CRUD operations without manual SQL implementation. For Native Image compatibility, GraalVM performs a deep static analysis of this interface during compilation to pre-generate the necessary dynamic proxies that would otherwise be unavailable in a restricted native environment.

```
        package dev.quaestus.pioneer_service;

        import
org.springframework.data.jpa.repository.JpaRepository;
        import org.springframework.stereotype.Repository;

        @Repository
        public interface UserRepository extends JpaRepository<User,
Long> {

        }
```

The REST API Layer, governed by the UserController, serves as the primary ingress point for HTTP traffic. By employing Constructor Injection, the system ensures that dependencies are resolved statically. Because the routing table is finalized during the AOT process, the service achieves near-instantaneous response times upon the first request, as the traditional overhead of runtime annotation scanning is removed.

```
package dev.quaestus.pioneer_service;


import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

import java.util.List;


@RestController

public class UserController {

    private final UserRepository repository;


    public UserController(UserRepository repository) {
```

```
        this.repository = repository;
    }

    @GetMapping("/users")
    public List<User> getAllUsers() {
        return repository.findAll();
    }
}
```

The Application Entry Point facilitates the initialization of the Spring context and includes a diagnostic module for real-time performance monitoring. A Command Line Runner bean is used to seed the H2 in-memory database, ensuring the system is fully operational before diagnostics are captured. The algorithm calculates the Resident Set Size (RSS) by measuring the difference between total and free memory, providing an empirical basis for comparison against traditional JVM footprints.

```
package dev.quaestus.pioneer_service;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class PioneerServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(PioneerServiceApplication.class,
args);
        long memoryUsed = (Runtime.getRuntime().totalMemory() -
```

```
                                Runtime.getRuntime().freeMemory())    /
1024 / 1024;
    }


    @Bean
    CommandLineRunner initDatabase(UserRepository repo) {
        return args -> {
            repo.save(new User("GraalVM Performance Test"));
            repo.save(new User("AOT Optimization Demo"));
        };
    }
}
```

PERFORMANCE EVALUATION

The empirical testing was conducted on a workstation environment running Windows 11, equipped with 16GB of RAM and a 14-core high-performance processor. The objective was to measure the delta between a dynamic Java Virtual Machine (JVM) execution and a standalone Ahead-of-Time (AOT) compiled binary.

Table 1. Comparative Performance Metrics: Standard JVM vs. GraalVM Native Image Execution

Metric	Standard Execution (JVM)	Native Image (.exe file)	Improvement Factor
Startup Time	5.033 s	0.425 s	11.8x Faster
RAM	~480 MB	31 MB	93.5% Reduction
Build Duration	Instant (~5 s)	Extensive (~10 min)	One-time overhead

This data shows a fundamental change in the way in which the pioneer service interacts with system resources. The decrease in startup time from more than five seconds to less than half a second is due to the absence of class loading, bytecode interpretation, and JIT compilation. The entry point of the application in the Native Image is already compiled into machine code and therefore allows for near-instant availability of the service. Another important point is the optimization of memory footprint. The traditional JVM process

consumes a large amount of memory due to memory allocation to the permanent generation, heap management, and virtual machine overhead. The Native Image process, on the other hand, consumes a minimum of 31 MB of memory, which is the Resident Set Size (RSS), as it only contains code paths that have been deemed reachable in the static analysis phase and therefore represents a 93.5% memory savings.

CHALLENGES AND LIMITATIONS

Moreover, the experimental results obtained from the pioneer-service demonstrate that the integration of GraalVM Native Image fundamentally redefines the execution lifecycle of Spring Boot microservices. This is because, unlike the traditional Just-In-Time (JIT) compilation, which heavily depends on a high-performance Virtual Machine to execute dynamic class loading and reflective metadata scanning at startup time, the Ahead-of-Time (AOT) compilation strategy moves these expensive operations to the build time. By adhering to the Closed World Assumption, the GraalVM static analyzer eliminates all unreachable code paths and determines the heap state in advance, resulting in a binary image that does not require interpretation. This architectural change is the major reason for the 11.8x acceleration in the startup time. Moreover, the removal of the JVM's internal management structures, such as the Just-In-Time compiler threads and the permanent generation memory, results in a 93.5% smaller RSS footprint for the service. This makes the microservice highly responsive and efficient, making it an optimal candidate for high-density auto-scaling environments in the cloud, where "Cold Start" latency is a critical performance bottleneck.

CONCLUSION

The research conducted through the development and analysis of the pioneer-service has conclusively demonstrated that the GraalVM Native Image technology has marked a major shift in the deployment of Java-based microservices. By successfully migrating from the dynamic execution model of the standard Java Virtual Machine to a static, Ahead-of-Time (AOT) compiled binary, the study has marked a transformative shift in the reduction of startup latency and memory footprint. By achieving an 11.8x acceleration in startup speed and a 93.5% reduction in the Resident Set Size (RSS), the study has conclusively demonstrated that the traditional "Cold Start" and high-overhead issues associated with Java are no longer relevant. This study has conclusively demonstrated that the Spring Boot 3.4 framework, in conjunction with the GraalVM Native Image technology, has marked a major shift in the deployment of Java as a top-tier contender for modern, resource-constrained architectures, which were historically dominated by lower-level languages.

Apart from the immediate technical achievements, the results of this paper have profound consequences for the economic and environmental

viability of cloud-native applications. The dramatic memory savings will also have a direct impact on deployment density, enabling organizations to deploy many more microservice applications on the same underlying infrastructure, thus lowering the operational costs of public and private cloud platforms. Moreover, the ability to scale from zero to ready in an instant also makes this technology the perfect choice for serverless and event-driven applications. By removing the energy-intensive JIT compilation process at runtime, AOT compilation also helps to further global Green IT initiatives by lowering the ecological footprint of large-scale datacenter operations.

In conclusion, with regard to the final evaluation, it is noted that, while the process of using the Native Image technology requires a more complex build process, departing from the flexibility offered by the "Open World" characteristic of Java, the trade-off is worthwhile, especially with regard to cloud-native applications. The pioneer-service is an example of a successful proof of concept, showing that with the correct configuration, while applying the AOT principles, Java can compete with the efficiency of native applications without compromising the productivity and robustness offered by the Spring ecosystem. Future research directions should aim at further automating the generation of reachability metadata for third-party libraries, which is the last step toward the universal adoption of native compilation for the Java ecosystem. This research proves that GraalVM Native Image is not only an optimization technique, but an evolution of Java, especially with regard to microservices and serverless computing.

While writing the present article, the authors (Marko MARKOVIĆ, Ana VELJIĆ, Dušan RAJČEVIĆ) used ChatGPT in order to translate from Serbian into English. After using this tool/service, the authors reviewed and edited the content as needed and takes full responsibility for the content of the published article.

REFERENCES

Armbruster, A., Baker, J., Cunei, A., Flack, C., Holmes, D., Pizlo, F., Pla, E., Prochazka, M., & Vitek, J. (2007). A real-time Java virtual machine with applications in avionics. *ACM Transactions on Embedded Computing Systems, 7*(1), Article 5. https://doi.org/10.1145/1324969.1324974

Belafia, R., Jeanjean, P., Barais, O., Le Guernic, G., & Combemale, B. (2021). From monolithic to microservice architecture: The case of extensible and domain-specific IDEs. *Proceedings of the 24th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 454–463. https://doi.org/10.1109/MODELS-C53483.2021.00070

Hřivnáč, J. (2024). Multilanguage Frameworks Towards the Ideal Multilanguage Environment. *Epj Web of Conferences*, 295, 05016. https://doi.org/10.1051/epjconf/202429505016

Khordadi, A. (2020). Execution Migration of Managed Languages Programs Among Heterogeneous-ISA Processing Units. *Middleware'20 Doctoral Symposium*, 38-41. https://doi.org/10.1145/3429351.3431749

Long, H. D., Vergilio, T., & Kor, A. (2023). Comparative Performance and Energy Efficiency Analysis of Jvm Variants and Graalvm in Java Applications. *SSRN*, https://doi.org/10.2139/ssrn.4373169

Mohan, J. S. S. and Goswami, K. (2025). Performance Analysis and Comparison of Node.Js and Java Spring Boot in Implementation of Restful Applications. *Software Practice and Experience*, 55(7), 1209-1233. https://doi.org/10.1002/spe.3418

Oracle. (2026). GraalVM Native Image Reference Guide: Understanding the Closed-World Assumption and Static Analysis. GraalVM Documentation. Available: https://www.graalvm.org/native-image/

Preuveneers, D. and Joosen, W. (2019). Towards Multi-party Policy-based Access Control in Federations of Cloud and Edge Microservices. *IEEE*, 29-38. https://doi.org/10.1109/eurospw.2019.00010

Šipek, M., Muharemagic, D., Mihaljević, B., & Radovan, A. (2020). Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus. *IEEE*, 1746-1751. https://doi.org/10.23919/mipro48935.2020.9245290

Ugwumba, N. and Jaja, P. (2025). Ahead-of-Time vs. Just-in-Time Compilation Trade-offs: Empirical performance studies. Preprint. *Research Square.* https://doi.org/10.21203/rs.3.rs-7915532/v1

Wiewiórka, M., Szmurło, A., Stankiewicz, P., & Gambin, T. (2022). Cloud-native distributed genomic pileup operations. *Bioinformatics*, https://doi.org/10.1101/2022.08.27.475646

Wimmer, C., Stancu, C., Hofer, P., Jovanovic, V., Wögerer, P., Kessler, P., Pliss, O., & Wuerthinger, T. (2019). Initialize once, start fast: Application initialization at build time. *Proceedings of the ACM on Programming Languages, 3*(OOPSLA), 1–29. https://doi.org/10.1145/3360610

Ye, C., Shen, Z., Wu, Y., & Loskot, P. (2024). Reconsidering Python Syntax to Enhance Programming Productivity. *International Journal for Research in Applied Science and Engineering Technology,* 12(3), 776-785. https://doi.org/10.22214/ijraset.2024.58903

Yuhala, P., Ménétrey, J., Felber, P., Schiavoni, V., Tchana, A., Thomas, G., Guiroux, H., & Lozi, J.-P. (2021). Montsalvat: Intel SGX shielding for GraalVM native images. *In Proceedings of the 22nd International Middleware Conference* (pp. 352–364). https://doi.org/10.1145/3464298.3493406

Notes on the authors:

Marko MARKOVIĆ, M.Sc., is a Ph.D. candidate at the Faculty of Economics and Engineering Management, University Business Academy in Novi Sad and a Teaching Assistant at the Faculty of Applied Management, Economics and Finance, University Business Academy in Novi Sad. Email: marko.markovic@mef.edu.rs

Ana VELJIĆ, M.Sc., is a Ph.D. candidate at the Faculty of Technical Sciences Čačak, University of Kragujevac and Teaching Assistant at the Faculty of Applied Management, Economics and Finance, University Business Academy in Novi Sad. E-mail: ana.veljic@mef.edu.rs

Dušan RAJČEVIĆ, M.Sc., is a Ph.D. candidate at the Faculty of Economics and Engineering Management, University Business Academy in Novi Sad and a Teaching Assistant at the Faculty of Applied Management, Economics and Finance, University Business Academy in Novi Sad. Email: dusan@mef.edu.rs