UDC 336
DOI: 10.15587/1729-4061.2025.315493

EVALUATING THE IMPACT OF GRAALVM AND JVM ON MOBILE BANKING MICROSERVICES PERFORMANCE

Edwin Yosef Setiawan Sihombing
Corresponding author
Master of Computer Science*
E-mail: edwin.sihombing@binus.ac.id
Muhammad Zarlis
Professor of Computer Science*
*Department of Information System Management
BINUS University
K.H. Syahdan str., 9, Kemanggisan,
Palmerah Jakarta, Indonesia, 11480

This study investigates the implementation of Graal Virtual Machine (GraalVM) in Java Spring Boot microservices for mobile banking applications. Increasing digital banking demand and user growth necessitate systems that can handle high transaction volumes efficiently. Java Virtual Machine (JVM) environments face challenges, including slower application startup times, higher CPU usage, and increased memory consumption, limiting their suitability for such high-demand scenarios. To address these issues, this research uses a quasi-experimental design to compare microservices' performance on GraalVM and JVM by analyzing key metrics: application startup time, CPU usage, and memory consumption under various load scenarios. Results show that GraalVM significantly improves startup times, reducing delays by 25–30 seconds across services, thus enhancing responsiveness. CPU usage showed varied outcomes: mobile-service demonstrated reductions (e.g., 0.8678 to 0.7798 for 100 users), whereas profile-service and casa-service recorded slight increases under certain workloads (e.g., 0.7829 to 0.8569 for profile-service at 100 users). Memory consumption increased notably for GraalVM, particularly in high-load scenarios such as casa-service at 600 users (198.47 MB to 591.38 MB). These findings highlight the trade-offs of adopting GraalVM, with faster startup times offset by higher memory usage in specific services. The results underscore the importance of workload-specific evaluations when optimizing microservices. Practical applications of this research include guiding system architects in selecting appropriate runtime environments to enhance the performance and scalability of mobile banking systems, ensuring efficient operation under varying demands

Keywords: GraalVM, Java Spring Boot, Java Virtual Machine, application performance, resource optimization

Received 21.10.2024
Received in revised form 19.12.2024
Accepted 07.01.2025
Published 28.02.2025

How to Cite: *Sihombing, E. Y. S., Zarlis, M. (2025). Evaluating the impact of GraalVM and JVM on mobile banking microservices performance. Eastern-European Journal of Enterprise Technologies, 1 (13 (133)), 46–58. https://doi.org/10.15587/1729-4061.2025.315493*

1. Introduction

In recent decades, technological advancements such as the Internet, cloud computing, and mobile applications have profoundly reshaped the banking sector. These innovations have transformed financial services, enhancing convenience, accessibility, and user experience. Studies exploring e-banking convergence across EU countries [1] and digital transformation in Romania [2] highlight the broad impact of these developments on the banking industry. Among these advancements, mobile banking has emerged as a cornerstone of digital transformation, providing a seamless and always-available interface to meet the growing demand for faster and more accessible financial services.

The rapid adoption of mobile banking has significantly altered the financial landscape, enabling customers to perform transactions, manage accounts, and access various services without relying on physical bank branches. In Indonesia, for instance, mobile banking usage increased by 48 % in 2022, reaching 23 million users by late 2023, as evidenced in one of Indonesia's most growing banking. This growth underscores the vital role of mobile banking in modern financial ecosystems. However, it also reveals pressing challenges in managing the backend systems that support these applications, particularly concerning resource consumption and system performance under heavy demand.

While research on mobile banking systems has often focused on scalability and security [3], there remains a critical gap in addressing the efficiency of backend systems, especially within microservices architectures. Studies on performance-efficient core banking systems based on microservices architecture [4] further underscore these challenges. Research on system performance in urban land planning databases [5] highlights similar concerns, emphasizing the need for robust solutions in handling large-scale, resource-intensive applications. As user volumes continue to grow, issues such as high CPU and memory usage can lead to performance bottlenecks, slower transaction processing, and diminished user satisfaction. These issues, observed across various technological implementations, demonstrate the need for further research into improving backend efficiency.

2. Literature review and problem statement

A study on JDK frameworks, including GraalVM Community and Enterprise Editions, used DaCapo benchmarks on Java 8 and Java 11 to evaluate performance and found GraalVM Enterprise Edition to excel in many scenarios [6]. However, this study did not consider GraalVM's applicability to Spring Boot or complex service-oriented systems like mobile banking, which are characterized by high transaction volumes and strict latency requirements.

Another study focused on open-source projects and tested GraalVM on Java 11, showcasing its superior performance across benchmarking environments [7]. Despite its contributions, the study lacked a targeted analysis of mobile banking applications, ignoring the interplay between software performance and hardware configurations critical to Indonesian banking systems.

Research on GraalVM Native Image explored Rapid Type Analysis and Points-To Analysis in microservice frameworks like Spring [8]. Although it provided valuable technical insights, it did not address practical use cases such as mobile banking, leaving a gap in understanding its relevance to real-world applications.

Studies on Open Banking Architecture highlighted the efficiency of microservices over monolithic systems through load testing but excluded GraalVM from their scope [4]. This omission limits the applicability of these findings to the performance optimization of banking applications in Indonesia.

An evaluation of frameworks such as Spring Boot, Micronaut, and Quarkus analyzed startup performance, resource consumption, and compilation time [9]. While informative, this study overlooked GraalVM's role in optimizing Spring Boot applications within the context of Indonesia's mobile banking sector.

Research on GraalVM as a modern virtual machine emphasized its high performance and cross-language interoperability [10]. However, the study failed to investigate its use in Spring Boot microservices or its ability to handle high transaction volumes typical of mobile banking systems.

Other analyses have focused on GraalVM's debugging capabilities and advanced performance optimizations, such as ahead-of-time compilation and heap pre-population [11]. Despite these contributions, their relevance to mobile banking and Spring Boot contexts remains unexplored.

A study on resource-analysis tools demonstrated the potential for identifying software performance bottlenecks caused by misconfigurations [12]. While impactful, it did not assess GraalVM's role in addressing such bottlenecks in banking applications.

Similarly, a study on runtime microservices placement mechanisms showed significant resource savings and performance improvements [13]. However, it did not incorporate GraalVM, leaving questions about its potential benefits for banking systems unanswered.

Previous studies demonstrate unresolved issues that may stem from the fundamental trade-offs between performance optimization and resource overhead, practical integration challenges, or limitations in benchmarking real-world applications:

1. Practical difficulties in integrating GraalVM with widely used frameworks like Spring Boot.

2. Insufficient evaluation of GraalVM's performance in high-demand systems like mobile banking.

3. Trade-offs between GraalVM's advantages in startup time and its higher memory consumption.

A way to overcome these challenges lies in conducting a targeted evaluation of GraalVM's applicability to specific use cases. This study addresses this need by comparing GraalVM and JVM in Spring Boot microservices under realistic conditions. This analysis contributes to optimizing performance for high-demand mobile banking applications, which require rapid initialization, efficient CPU usage, and scalable resource management.

The rapid growth in mobile banking users has posed significant challenges for server infrastructure, particularly in managing rising transaction volumes while maintaining fast, secure services. Banks face critical issues with long microservice startup times and high resource consumption specifically CPU and memory usage. Slow startups can delay the availability of essential services, raising the risk of downtime, while high resource usage can create performance bottlenecks, further straining system efficiency. This not only jeopardizes seamless transaction processing but also escalates operational costs and affects the bank's ability to meet real-time customer expectations.

All these factors underscore the necessity of conducting a study to address two key questions:

1. Identifying solutions to maintain system efficiency and performance is critical to managing the rapid growth of mobile banking application users, particularly in handling large transaction volumes.

2. Accelerating microservice startup times to align with zero-downtime service goals is essential for improving service quality and enhancing the user experience in mobile banking applications.

3. The aim and objectives of the study

The aim of this study is to evaluate the performance improvements of GraalVM compared to JVM within Java Spring Boot microservices, particularly in a mobile banking environment where transaction speed and resource efficiency are critical for ensuring a seamless user experience.

To achieve this aim, the following objectives are accomplished:

– to measure and compare the startup times of microservices running on GraalVM and JVM to evaluate potential improvements in application responsiveness;

– to analyze CPU usage under various load scenarios (low, medium, and high) to assess whether GraalVM provides better CPU efficiency compared to JVM;

– to examine memory consumption across different load conditions to evaluate GraalVM's memory management and resource allocation compared to JVM.

4. Materials and methods

The object of this study is the performance differences between the Java Virtual Machine (JVM) and Graal Virtual Machine (GraalVM) in Java Spring Boot microservices. By focusing on critical metrics such as application startup time, CPU usage, and memory consumption, the study aims to understand the impacts of these runtime environments under varying user load scenarios. This evaluation provides insights into the suitability of each runtime environment for optimizing microservice performance.

This study uses a quasi-experimental design to evaluate performance differences between the Java Virtual Machine (JVM) and Graal Virtual Machine (GraalVM) in Java Spring Boot microservices. Chosen for its suitability in controlled yet realistic environments, this approach ensures reliable comparisons between the two runtime environments. The methodology, illustrated in Fig. 1, includes problem identification, literature review, requirement analysis, GraalVM implementation, testing, data collection, hypothesis testing, and conclusion, providing a systematic framework for the analysis.

This study tests the following hypotheses for performance differences in Java Spring Boot microservices running on GraalVM versus JVM:

1. Application startup time hypothesis:

– H_0: there is no significant difference in application startup time between GraalVM and JVM;

– H_1: GraalVM significantly reduces application startup time compared to JVM.

2. Application CPU usage hypothesis:

– H_0: GraalVM implementation does not produce a significant difference in the CPU usage of Java Spring Boot applications compared to JVM;

– H_1: GraalVM implementation produces a significant difference in the CPU usage of Java Spring Boot applications compared to JVM.

3. Application memory consumption hypothesis:

– H_0: GraalVM implementation does not produce a significant difference in the memory usage of Java Spring Boot applications compared to JVM;

– H_1: GraalVM implementation produces a significant difference in the memory usage of Java Spring Boot applications compared to JVM.

This structured approach ensures a comprehensive evaluation of the impact of GraalVM on mobile banking microservices, enabling the derivation of insights into its performance and resource utilization.

The following assumptions were made to maintain the validity and reliability of the study:

1. This study uses a quasi-experimental design to evaluate performance differences between the Java Virtual Machine (JVM) and Graal Virtual Machine (GraalVM) in Java Spring Boot microservices. Chosen for its suitability in controlled yet realistic environments, this approach ensures consistent testing conditions and reliable comparisons. The methodology, illustrated in Fig. 1, includes problem identification, literature review, requirement analysis, GraalVM implementation, testing, data collection, hypothesis testing, and conclusion, providing a systematic framework for the analysis.

2. Controlled testing conditions closely replicate real-world deployment scenarios for microservices.

3. Docker containers are used to create isolated environments, minimizing external influences on performance results.

4. Performance metrics (application startup time, CPU usage, and memory consumption) accurately represent the runtime environments' behavior under the defined load scenarios.

To streamline the research process and focus on key aspects, the following simplifications were adopted:

1. The study focuses on three critical performance metrics: application startup time, CPU usage, and memory consumption.

2. Testing is conducted under three predefined user load scenarios:

a) low load: 100 users;

b) medium load: 300 users;

c) high load: 600 users.

3. Three microservices – mobile-service, casa-service, and profile-service – were selected for their relevance to mobile banking operations.

4. Standardized resource allocations were applied within Docker containers, assuming minimal variability from the host machine.

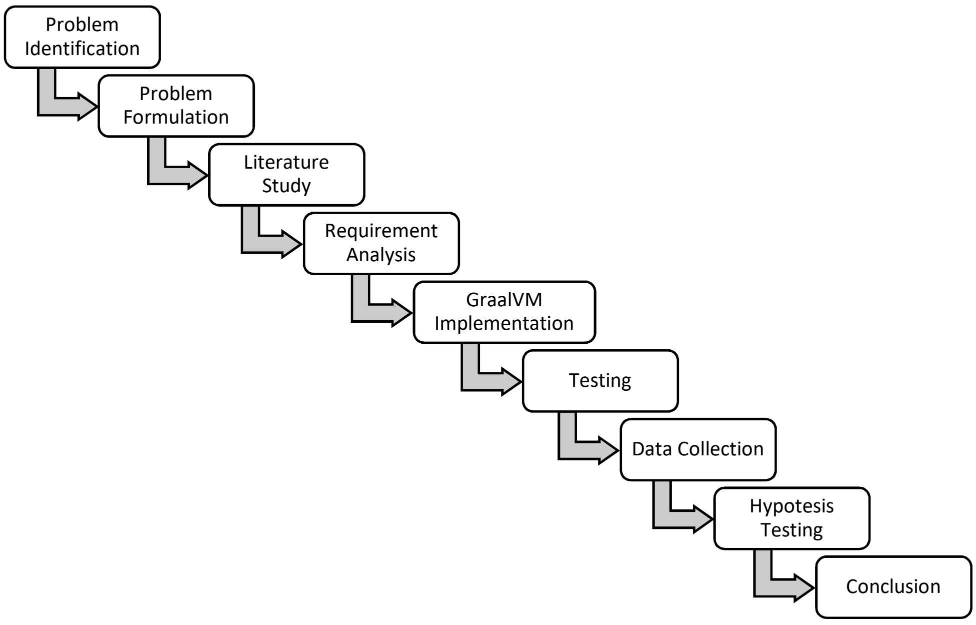

Fig. 1. Research methodology

Experimental setup.

The experimental environment was configured to ensure consistency and reproducibility across all tests. Docker containers were utilized to isolate the runtime environments for both JVM and GraalVM microservices. Each microservice was deployed in a container with specific resource constraints, simulating real-world deployment scenarios:

1. Resource constraints.

Controlling resource allocation is essential to isolate the effects of runtime environments. In this research, the resource constraints were carefully chosen to mimic the limitations typically encountered in a production environment, ensuring the results are relevant and practical for real-world applications. Each container was assigned specific CPU and memory constraints to ensure consistent conditions:

a) CPU allocation: each container was limited to 0.9 CPU cores;

b) memory allocation: maximum heap size 1,536 MB, and initial heap size 256 MB.

These constraints ensured that the runtime environment's performance could be isolated without interference from host machine variability.

2. Host machine specifications.

Although Docker provides isolated environments, the underlying hardware can influence container performance. The experiments were conducted on a host machine with the following specifications:

a) processor: Apple M1 Pro;

b) memory: 16 GB RAM;

c) operating system: Sonoma 14.6.1.

These specifications ensured sufficient resources for running multiple containers simultaneously while maintaining reliability.

3. Software and testing tools.

The following software and tools were employed to create and measure the experimental environment:

3. 1. Java Spring Boot Framework.

Java Spring Boot Framework version 3.2.0 was used to develop scalable and production-ready microservices. As highlighted in [5], Spring Boot simplifies development with features like embedded servers, automatic configuration, and dependency management, allowing developers to focus on business logic. Its compatibility with microservices and cloud architectures makes it ideal for scalable and secure applications.

3. 2. Docker.

Docker version 24.0 was used to containerize the runtime environments for JVM and GraalVM. As discussed in [4], Docker exemplifies container technology designed to implement the "Single Service Instance per Container" pattern. This pattern was chosen for its ability to dynamically manage resource usage, such as CPU and memory, while ensuring efficient resource utilization. Additionally, it provides faster application build times and startup compared to traditional virtual machines, making it ideal for creating isolated and reproducible testing conditions.

3. 3. Java Virtual Machine.

OpenJDK Version 17 served as the baseline runtime environment. The Java Virtual Machine (JVM), as highlighted in prior research [14], is an abstract computing engine that enables the execution of Java programs and other languages compiled into Java bytecode. It offers hardware and OS independence, ensuring portability across platforms, and provides features like small code size and robust security mechanisms to prevent malicious code from impacting users. These characteristics make JVM a reliable foundation for running scalable and secure applications.

3. 4. GraalVM.

GraalVM CE Version 21.0 was evaluated for its advanced runtime optimizations. GraalVM, developed by Oracle [10], is a polyglot virtual machine supporting multiple languages like Java, JavaScript, and Python. It provides Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilation, native image generation, and cross-language interoperability. Built on Java HotSpot VM, it enhances performance and flexibility, making it ideal for tasks requiring high responsiveness and efficient resource usage.

3. 5. Apache Jmeter.

Apache JMeter version 5.5 was used as a performance testing tool to simulate varying user loads and measure system responsiveness. Developed by the Apache Software Foundation [15], JMeter is an open-source tool that evaluates web application performance by generating high user loads and measuring server responsiveness. Its ability to simulate diverse testing scenarios helps identify bottlenecks and optimize application scalability and responsiveness.

3. 6. Monitoring tool.

Grafana was used as a monitoring tool to visualize and analyze real-time performance metrics such as CPU and memory consumption during the tests.

Performance metrics.

To evaluate the performance differences between the Java Virtual Machine (JVM) and Graal Virtual Machine (GraalVM), the study focused on three key metrics that are critical to assessing the efficiency and effectiveness of microservices in real-world scenarios. Each metric was carefully measured under identical testing conditions across varying user loads (100, 300, and 600 virtual users):

1. Application startup time.

Application startup time refers to the duration required for a microservice to initialize and become fully operational. This metric is crucial for understanding the responsiveness of runtime environments. Shorter startup times contribute to reduced downtime and improved user experience, making this metric particularly significant in systems adhering to a zero-downtime operational model.

The startup time was recorded for each microservice (mobile-service, casa-service, and profile-service) under both JVM and GraalVM environments using automated logs generated during container initialization.

2. Application CPU usage.

CPU usage measures the average amount of processing power consumed by the microservices under different load conditions. This metric provides insights into the efficiency of each runtime environment in managing computational tasks. Lower CPU usage indicates higher efficiency, enabling the system to handle more requests or maintain operations with reduced hardware requirements.

CPU usage was monitored in real time using Grafana, which aggregated data collected by system monitoring tools integrated with the Docker environment.

3. Application memory consumption.

Memory consumption refers to the amount of memory utilized by the microservices during operation. This metric is critical for understanding the resource demands of JVM and GraalVM under varying user loads. Efficient memory management is essential for optimizing infrastructure costs and ensuring system stability, particularly in environments with limited resources.

Memory usage was monitored using Grafana and Docker container statistics during the tests. Metrics included average memory consumption during steady-state operation under all load scenarios.

Testing scenarios.

The testing scenarios are designed to replicate real-world conditions and evaluate the performance of microservices under controlled and reproducible conditions. Fig. 2 illustrates the architecture and methodology used to test the microservices and collect performance data.

In this setup, Apache JMeter simulates concurrent user requests, representing typical user interactions with the application. These requests are processed by the microservices, which handle various banking operations such as user authentication, account management, and transaction processing. The microservices were deployed in isolated Docker containers with uniform resource constraints to ensure consistency in runtime comparisons. Performance data, including CPU usage, memory consumption, and application startup time, was collected in real-time using Grafana and microservice logs for subsequent analysis.

The core of this evaluation focused on three critical microservices, detailed below:

1. Mobile service: this microservice supports the initialization of the splash screen and the rendering of the home screen. It handles critical UI components and real-time updates, ensuring a seamless and responsive experience for end users as they access the application.

2. Casa service: manages account-related operations, including balance inquiries and transaction histories. This service is resource-intensive due to frequent database queries and high transaction volumes.

3. Profile service: handles user authentication and profile management, ensuring secure access to the application and protecting sensitive user data.

Each microservice was deployed in Docker containers configured with predefined resource constraints to standardize the runtime environment for JVM and GraalVM.

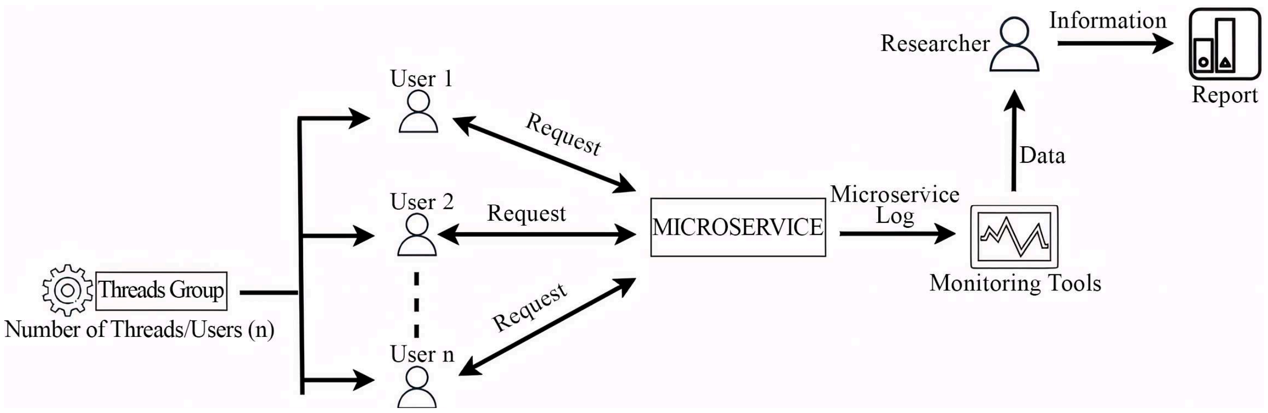

Fig. 2. Testing architecture illustrating the flow of user requests and performance monitoring

The performance of these microservices was then tested under varying conditions to evaluate their scalability and responsiveness, as detailed below.

To simulate real-world conditions, the microservices were subjected to three different load scenarios using Apache JMeter. These scenarios represented varying levels of user demand:

1. 100 users (low load).

This scenario established a baseline for application startup time, CPU usage, and memory consumption. It provided insights into GraalVM's efficiency under minimal traffic, focusing on its ability to handle basic operational requirements.

2. 300 users (medium load).

Simulating a more typical usage pattern, this scenario assessed system performance under moderate traffic. It allowed an evaluation of GraalVM's resource optimization capabilities in a steady operational state, reflecting conditions closer to real-world usage.

3. 600 users (high load).

Designed to test the system's limits, this high-load scenario simulated peak traffic conditions that could occur during periods of intense demand. It provided insights into how GraalVM manages extreme loads and resource allocation under stress.

Statistical analysis.

To validate the findings and assess the significance of performance differences between JVM and GraalVM, statistical analysis was conducted on the collected data. This ensured that the observed differences were not due to random chance and provided robust insights into the impact of the runtime environments on the tested microservices.

Before applying statistical tests, the normality of the data was evaluated to determine the appropriate statistical method. The Shapiro-Wilk test was used to assess whether the performance metrics (application startup time, CPU usage, and memory consumption) followed a normal distribution. This step was critical in choosing between parametric and nonparametric tests for hypothesis testing:

1. Paired Sample T-Test.

The Paired Sample T-Test [16], also called the dependent t-test, compares two means from related samples, such as the same subjects under different conditions. By assuming normally distributed differences, it controls variability and improves statistical sensitivity. A p-value below the significance level (e.g., 0.05) indicates a significant difference, leading to rejection of the null hypothesis.

2. Wilcoxon Signed Rank Test.

Nonparametric analysis is used when data lacks a normal distribution or its distribution is unknown, offering flexibility for various scenarios [17, 18]. The Wilcoxon Signed Rank Test, a nonparametric alternative to the paired t-test, compares paired samples by ranking differences and computing a test statistic. It is ideal for non-normal data, providing robust results without strict parametric assumptions.

To ensure the reliability and accuracy of the experimental results, several validation steps were undertaken. The testing environment was standardized by deploying all microservices in isolated Docker containers with identical resource constraints. Performance metrics were monitored using Grafana and corroborated with microservice logs to ensure consistency and completeness of the collected data. Statistical analysis methods, including both parametric and nonparametric tests, were carefully chosen based on the normality of the data, ensuring that the analysis aligned with the underlying data distribution. Additionally, all tests were conducted multiple times under each load condition to minimize the impact of anomalies and ensure reproducibility. These measures collectively validate the robustness of the experimental design and provide confidence in the conclusions drawn from the study.

5. Performance outcomes of GraalVM and JVM in mobile banking microservices

5. 1. Evaluating startup time enhancements of microservices on GraalVM and JVM

5. 1. 1. Startup time observations

This section evaluates the differences in application startup times between GraalVM and JVM for the mobile-service, profile-service, and casa-service.

Fig. 3 presents the startup times for the three services (mobile-service, casa-service, and profile-service) running on JVM. This data demonstrates the baseline performance before implementing GraalVM.

Fig. 4 shows the startup times for the same services after transitioning to GraalVM, highlighting the impact of the new runtime environment.

The startup time data was collected by measuring the time it took for each service to initialize in both JVM and GraalVM environments over ten trials. The results for each trial are presented in Fig. 1–3. On average, GraalVM showed a significant reduction in startup time for all services:

1. Mobile-service.

GraalVM reduced the startup time of the mobile-service by an average of approximately 25 seconds compared to JVM. This improvement, while small in absolute terms, represents a significant percentage reduction relative to the overall startup time of the service. The reduction is particularly impactful in high-frequency deployment scenarios, where cumulative time savings across multiple instances can enhance system efficiency and reduce operational downtime. This makes GraalVM a favorable choice for microservices that require rapid initialization, such as the mobile-service.

2. Casa-service.

For the casa-service, GraalVM demonstrated an average improvement of 27 seconds in startup time. Given the critical nature of casa-service operations in handling core transactional workloads, even small reductions in startup time contribute to overall system performance. Faster initialization allows quicker recovery during system updates or restarts, aligning well with the zero-downtime operational objectives of modern microservices architectures. This improvement highlights GraalVM's ability to optimize resource-intensive services efficiently.

3. Profile-service.

The profile-service achieved an average startup time improvement of 27 seconds with GraalVM compared to JVM. This faster startup time ensures that user-facing services are available more quickly, which is crucial for maintaining a seamless user experience in mobile banking applications. In scenarios involving frequent scaling or service redeployment, such performance improvements can significantly enhance the perceived responsiveness of the system, directly contributing to improved customer satisfaction.

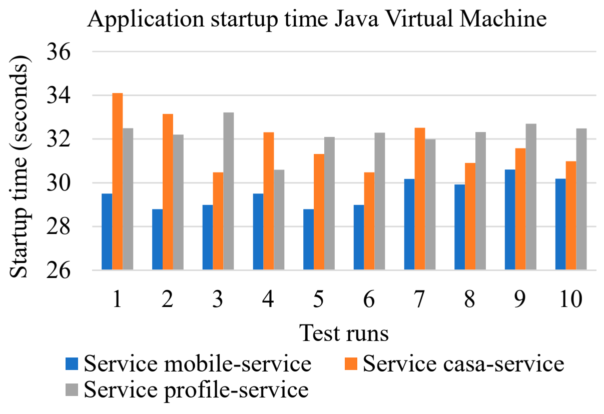

Fig. 3. Application startup times for services running on Java Virtual Machine

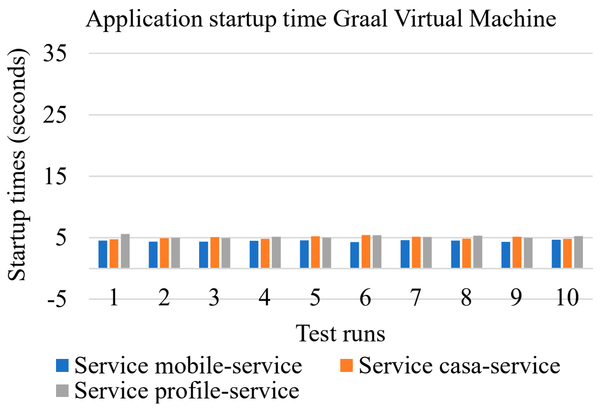

Fig. 4. Application startup times for services running on Graal Virtual Machine

5. 1. 2. Statistical validation of startup time improvements

To confirm the significance of the observed reductions in startup times, a paired sample t-test was conducted. This analysis compared the startup times for each service on JVM and GraalVM. The test results are summarized in Table 1.

Table 1

Paired sample t-test results for Spring Boot application startup time

Micro services	Mean difference	Standard deviation	Standard error mean	Confidence interval		t-value	df	p-value
				Lower bound	Upper bound			
Mobile-service	25.090	0.622	0.197	24.645	25.536	127.499	9	5.7057×10^{-16}
Profile-service	27.069	0.718	0.227	26.556	27.582	119.296	9	1.0378×10^{-15}
Casa-service	26.782	1.337	0.423	25.825	27.738	63.334	9	3.0773×10^{-13}

The analysis supports rejecting the null hypothesis (H_0) and accepting the alternative hypothesis (H_1) for all three services. GraalVM reduced startup times by an average of 25.09 milliseconds for the mobile-service, 26.78 milliseconds for the casa-service, and 27.07 milliseconds for the profile-service. These reductions are statistically significant, with p-values well below 0.001. The confidence intervals further validate the reliability of these improvements, demonstrating that GraalVM consistently enhances startup performance.

5. 2. Analyzing CPU efficiency of GraalVM and JVM across varied load conditions

5. 2. 1. CPU usage trends under different load scenarios

This section evaluates the CPU usage of GraalVM and JVM under three different user loads: 100, 300, and 600 users. The comparison aims to assess whether GraalVM offers efficiency improvements in resource utilization compared to JVM. The results are summarized in Table 2.

Table 2

Average CPU usage for all user loads

Users	Mobile-service		Profile-service		Casa-service	
	JVM	GraalVm	JVM	GraalVm	JVM	GraalVm
100	0.8678	0.7798	0.7829	0.8569	0.9864	0.9448
300	0.840	0.805	0.5815	0.6816	0.8335	0.9116
600	0.8669	0.8018	0.6918	0.7066	0.7951	0.9101

Table 2 compares average CPU usage across user loads (100, 300, and 600 users) for the mobile-service, profile-service, and casa-service running on JVM and GraalVM. The following observations are noted:

1. Low load scenario (100 users).

At 100 users, GraalVM demonstrated lower CPU usage compared to JVM for the mobile-service (0.8678 and 0.7798) and casa-service (0.9864 and 0.9448). However, the profile-service showed slightly higher CPU usage with GraalVM (0.8569) compared to JVM (0.7829).

2. Medium load scenario (300 users).

For 300 users, GraalVM consistently exhibited reduced CPU usage across the mobile-service (0.840 and 0.805) and profile-service (0.5815 and 0.6816). In the casa-service, GraalVM consumed slightly more CPU resources (0.8335 and 0.9116), indicating varying performance depending on service workload.

3. High load scenario (600 users).

Under heavy load, GraalVM showed no significant improvement in CPU usage for the mobile-service (0.8669 and 0.8018) and got slightly higher usage for the profile-service (0.6918 and 0.7066) and casa-service (0.7951 and 0.9101).

5. 2. 2. Statistical analysis of CPU usage

To validate these findings, the Wilcoxon signed-rank test was conducted to assess the significance of CPU usage differences between GraalVM and JVM. The test results are summarized for each load condition:

1. Low load scenario (100 users).

Table 3 shows the Wilcoxon signed-rank results for 100 users. For the mobile-service, GraalVM demonstrated lower CPU usage in most cases, with a rank sum of 45 for lower CPU usage compared to JVM. The profile-service and casa-service, however, showed relatively balanced rank sums, indicating no consistent differences between the two environments.

Table 4 provides the statistical summary for the 100-user scenario. While the mobile-service showed a statistically significant difference in CPU usage (z-value=–2.460, p-value=0.014), the direction of the difference indicates that GraalVM often used more CPU than JVM, contradicting expectations of improved efficiency. For the profile-service (p-value=0.471) and casa-service (p-value=0.235), no statistically significant differences were observed, indicating no clear CPU advantage for GraalVM.

For the mobile-service, the null hypothesis (H_0) is rejected as the p-value (0.014) indicates a statistically significant difference in CPU usage. However, the rank sums show that JVM often used less CPU than GraalVM, suggesting that JVM is more efficient for this service under low load.

For the profile-service and casa-service, the null hypothesis cannot be rejected (p-values=0.471 and 0.235), and the rank sums do not strongly favor either GraalVM or JVM, indicating comparable performance between the two.

2. Medium load scenario (300 users).

Table 5 presents the Wilcoxon signed-rank results for CPU usage at 300 users. For the mobile-service, the rank sum where GraalVM used more CPU than JVM was 26, whereas the rank sum where GraalVM used less CPU was 205, indicating that GraalVM generally consumed less CPU for this service under medium load. In contrast, for the profile-service and casa-service, the rank sums strongly favored JVM, with GraalVM consistently using more CPU. Both services had a rank sum of 231 where GraalVM used more CPU and a rank sum of 0 where GraalVM used less CPU, suggesting that JVM performed better for these services under medium load.

Table 6 provides the statistical summary for the 300-user scenario. The mobile-service showed a statistically significant difference in CPU usage (z-value=–3.123, p-value=0.002), indicating a measurable difference between GraalVM and JVM. For the profile-service and casa-service, the z-values were –4.052 (p-value=0.00005) for both, confirming statistically significant differences in CPU usage. However, the rank sums suggest that these differences favor JVM, as GraalVM consistently used more CPU than JVM for these two services.

Table 3

CPU usage Wilcoxon signed rank results for 100 virtual users

Microservices	GraalVm<JVM	Rank Sum (GraalVm<JVM)	GraalVM>JVM	Rank Sum (GraalVm>JVM)	Ties	Total
Mobile-service	9	45	12	186	0	21
Profile-service	16	136	5	95	0	21
Casa-service	8	124	11	66	0	21

Table 4

CPU usage statistical summary for 100 virtual users

Microservices	z-value	p-value
Mobile-service	–2.460	0.014
Profile-service	–0.721	0.471
Casa-service	–1.188	0.235

Table 5

CPU usage Wilcoxon signed rank results for 300 virtual users

Microservices	GraalVm<JVM	Rank Sum (GraalVm<JVM)	GraalVM>JVM	Rank Sum (GraalVm>JVM)	Ties	Total
Mobile-service	17	205	4	26	0	21
Profile-service	0	0	21	231	0	21
Casa-service	0	0	21	231	0	21

Table 6

CPU usage statistical summary for 300 virtual users

Microservices	z-value	p-value
Mobile-service	–3.123	0.002
Profile-service	–4.052	0.00005
Casa-service	–4.052	0.00005

For the mobile-service, the null hypothesis (H_0) is rejected (p-value=0.002), indicating a significant difference in CPU usage between GraalVM and JVM. The rank results suggest that GraalVM often used less CPU than JVM for this service under medium load.

For the profile-service and casa-service, the null hypothesis is also rejected (p-value=0.00005 for both). However, the rank results show that GraalVM consistently used more CPU than JVM for these services, indicating that JVM was more efficient under medium load for these particular microservices. These findings highlight service-specific differences in CPU usage between GraalVM and JVM at medium load.

3. High load scenario (600 users).

Table 7 presents the Wilcoxon signed-rank results for CPU usage at 600 users. For the mobile-service, the rank sum where GraalVM used less CPU than JVM was 78, while the rank sum where GraalVM used more CPU than JVM was 153.

This indicates that GraalVM often consumed more CPU than JVM under high load for this service. Similarly, for the profile-service, the rank sum where GraalVM used less CPU was 81, while the rank sum where GraalVM used more was 150, showing a similar trend. For the casa-service, the rank sum where GraalVM used more CPU was 231, with no cases where GraalVM used less CPU than JVM (rank sum=0), indicating that JVM consistently used less CPU for this service.

Table 8 provides the statistical summary for the 600-user scenario. For the mobile-service and profile-service, the p-values were 0.191 and 0.229, respectively, indicating no statistically significant differences in CPU usage between GraalVM and JVM. However, for the casa-service, the z-value was –4.050 (p-value=0.00005), indicating a statistically significant difference in CPU usage. The rank results show that this difference favored JVM, with GraalVM consistently using more CPU.

For the mobile-service and profile-service, the null hypothesis (H_0) cannot be rejected (p-values=0.191 and 0.229), as there were no statistically significant differences in CPU usage between GraalVM and JVM.

For the casa-service, the null hypothesis is rejected (p-value=0.00005), indicating a significant difference in CPU usage. The rank results suggest that JVM consistently outperformed GraalVM in terms of CPU efficiency for this service under high load. These findings suggest that under high load conditions, JVM generally performed better or similarly to GraalVM in terms of CPU usage.

Table 7

CPU usage Wilcoxon signed rank results for 600 virtual users

Microservices	GraalVm<JVM	Rank Sum (GraalVm<JVM)	GraalVM>JVM	Rank Sum (GraalVm>JVM)	Ties	Total
Mobile-service	4	78	17	153	0	21
Profile-service	6	81	15	150	0	21
Casa-service	0	0	21	231	0	21

Table 8

CPU usage statistical summary for 600 virtual users

Microservices	z-value	p-value
Mobile-service	–1.308	0.191
Profile-service	–1.204	0.229
Casa-service	–4.050	0.00005

5. 3. Comparative analysis of memory consumption in GraalVM and JVM under different loads

5. 3. 1. Memory usage observations

This section evaluates the memory consumption of GraalVM and JVM under three different user loads: 100, 300, and 600 users. The comparison aims to assess whether GraalVM offers improvements in memory efficiency compared to JVM. The results are summarized in Table 9.

Table 9

Average memory consumption for all user loads

Users	Mobile-service		Profile-service		Casa-service	
	JVM (MB)	GraalVm (MB)	JVM (MB)	GraalVm (MB)	JVM (MB)	GraalVm (MB)
100 users	154.0767	241.4048	145.6524	216.4286	150.3090	275.0505
300 users	172.2514	192.3571	164.2295	182.9048	167.5210	200.3333
600 users	165.4314	201.9048	193.6714	239.0476	198.4733	591.3810

Table 9 presents the average memory consumption for three services (mobile-service, profile-service, and casa-service) under varying user loads (100, 300, and 600 users) when running on JVM and GraalVM. The following observations summarize the results:

1. Low load scenario (100 users).

At 100 users, GraalVM exhibited consistently higher memory consumption compared to JVM across all services. For the mobile-service, GraalVM consumed an average of 241.4048 MB, significantly more than JVM's 154.0767 MB, suggesting that GraalVM incurs additional memory overhead under low traffic. Similarly, for the profile-service, GraalVM used 216.4286 MB, compared to 145.6524 MB for JVM, indicating a substantial difference. The casa-service showed the most pronounced disparity, with GraalVM consuming 275.0505 MB, nearly double JVM's memory usage of 150.3090 MB. These results suggest that under low load conditions, JVM is more memory-efficient across all services.

2. Medium load scenario (300 users).

For 300 users, GraalVM continued to consume more memory than JVM across all services, although the magnitude of the differences was less than in the low-load scenario. The mobile-service showed a moderate gap, with GraalVM consuming 192.3571 MB, compared to JVM's 172.2514 MB. For the profile-service, GraalVM's memory usage was 182.9048 MB, which was higher than JVM's 164.2295 MB, but the difference was less pronounced. In the casa-service, GraalVM consumed 200.3333 MB, while JVM used 167.5210 MB, showing a consistent but smaller gap. These findings indicate that while GraalVM's memory usage is still higher under medium load, the differences are less substantial than at lower traffic levels.

3. High load scenario (600 users).

Under high load conditions with 600 users, the memory consumption differences between GraalVM and JVM became more pronounced, particularly for the casa-service. For the mobile-service, GraalVM consumed 201.9048 MB, compared to JVM's 165.4314 MB, reflecting a noticeable increase. The profile-service followed a similar pattern, with GraalVM consuming 239.0476 MB versus JVM's 193.6714 MB. However, the most striking difference was observed in the casa-service, where GraalVM's memory consumption spiked to 591.3810 MB, almost three times JVM's 198.4733 MB. These results suggest that as traffic increases, GraalVM's memory usage escalates significantly, particularly for resource-intensive services like the casa-service, whereas JVM demonstrates greater stability in memory consumption.

5. 3. 2. Statistical validation of memory consumption differences

The Wilcoxon signed-rank test was used to evaluate the differences in memory consumption between GraalVM and JVM across three load scenarios: low (100 users), medium (300 users), and high (600 users). The results are summarized below:

1. Low load scenario (100 users).

Table 10 shows the Wilcoxon signed-rank results for memory consumption under a low load of 100 users. For all three services – mobile-service, profile-service, and casa-service – GraalVM consistently consumed more memory than JVM. The mobile-service had a rank sum of 231 for GraalVM>JVM, with no cases where GraalVM consumed less memory (GraalVM<JVM=0) or tied with JVM. Similarly, the profile-service and casa-service followed the same pattern, with rank sums of 231 for GraalVM>JVM and no instances of GraalVM<JVM. These results confirm a consistent trend of higher memory usage by GraalVM across all services under low load conditions.

Table 11 provides the statistical summary for the 100-user scenario. The results reveal statistically significant differences in memory consumption across all services. For the mobile-service, the z-value was –4.029, with a p-value of 0.00005. The profile-service exhibited a z-value of –4.052, also with a p-value of 0.00005, while the casa-service showed a z-value of –4.043, with the same p-value. The rank results from Table 10 confirm this trend, as GraalVM>JVM had a rank sum of 231 for all services, and there were no instances of GraalVM consuming less memory than JVM.

The rank results in Table 10 directly align with the statistical findings in Table 11, showing that GraalVM consistently consumed more memory than JVM in all cases. The rejection of the null hypothesis (H_0) for all services confirms statistically significant differences in memory consumption. The rank sums further emphasize that JVM demonstrated superior memory efficiency under low load conditions.

2. Medium load scenario (300 users).

Table 12 shows the Wilcoxon signed-rank results for memory consumption under a medium load of 300 users. For the mobile-service, the rank sums indicate that GraalVM consumed more memory than JVM in most cases, with a rank sum of 205 for GraalVM>JVM compared to 26 for GraalVM<JVM. Similarly, in the profile-service, the rank sum for GraalVM>JVM was 195, while GraalVM<JVM had a rank sum of 36, suggesting a general trend of higher memory usage by GraalVM. For the casa-service, all instances showed GraalVM consuming more memory than JVM, with a rank sum of 231 for GraalVM>JVM and no cases where GraalVM used less memory.

Table 13 provides the statistical summary for the 300-user scenario. The mobile-service had a z-value of –3.123 and a p-value of 0.002, indicating a statistically significant difference in memory consumption between GraalVM and JVM. The profile-service also showed a statistically significant difference, with a z-value of –2.789 and a p-value of 0.005. Finally, the casa-service demonstrated the most significant difference, with a z-value of –4.043 and a p-value of 0.00005.

Table 10

Memory consumption Wilcoxon signed rank results for 100 virtual users

Microservices	GraalVm<JVM	Rank Sum (GraalVm<JVM)	GraalVM>JVM	Rank Sum (GraalVm>JVM)	Ties	Total
Mobile-service	0	0	21	231	0	21
Profile-service	0	0	21	231	0	21
Casa-service	0	0	21	231	0	21

Table 11

Memory consumption statistical summary for 100 virtual users

Microservices	z-value	p-value
Mobile-service	–4.029	0.00005
Profile-service	–4.052	0.00005
Casa-service	–4.043	0.00005

Table 12

Memory consumption Wilcoxon signed rank results for 300 virtual users

Microservices	GraalVm<JVM	Rank Sum (GraalVm<JVM)	GraalVM>JVM	Rank Sum (GraalVm>JVM)	Ties	Total
Mobile-service	4	26	17	205	0	21
Profile-service	8	36	14	195	0	21
Casa-service	0	0	21	231	0	21

Table 13

Memory consumption statistical summary for 300 virtual users

Microservices	z-value	p-value
Mobile-service	–3.123	0.002
Profile-service	–2.789	0.005
Casa-service	–4.043	0.00005

For all three services, the null hypothesis (H_0) is rejected, as the p-values indicate statistically significant differences in memory consumption. The rank sums in Table 12 highlight that GraalVM consistently consumed more memory than JVM in most cases under medium load conditions.

This trend was particularly pronounced for the casa-service, where all observations favored GraalVM using more memory than JVM. These results suggest that JVM is more memory-efficient than GraalVM when managing medium traffic levels.

3. High load scenario (600 users).

Table 14 shows the Wilcoxon signed-rank results for memory consumption under a high load of 600 users. For the mobile-service and casa-service, GraalVM consistently consumed more memory than JVM, with rank sums of 231 for GraalVM>JVM and no cases where GraalVM consumed less memory (GraalVM<JVM=0). The profile-service showed a slight variation, with GraalVM>JVM having a rank sum of 216, while GraalVM<JVM had a rank sum of 15, indicating some instances where JVM used more memory than GraalVM, though the majority of cases still favored GraalVM consuming more memory.

Table 14

Memory consumption Wilcoxon signed rank results for 600 virtual users

Microservices	GraalVm<JVM	Rank Sum (GraalVm<JVM)	GraalVM>JVM	Rank Sum (GraalVm>JVM)	Ties	Total
Mobile-service	0	0	21	231	0	21
Profile-service	5	15	16	216	0	21
Casa-service	0	0	21	231	0	21

Table 15 provides the statistical summary for the 600-user scenario. The mobile-service had a z-value of –4.030, with a p-value of 0.00005, indicating a statistically significant difference in memory consumption between GraalVM and JVM. The profile-service also showed a significant difference, with a z-value of –3.507 and a p-value of 0.0004. Similarly, the casa-service demonstrated a statistically significant difference, with a z-value of –4.043 and a p-value of 0.00005.

Table 15

Memory consumption statistical summary for 600 virtual users

Microservices	z-value	p-value
Mobile-service	–4.030	0.00005
Profile-service	–3.507	0.0004
Casa-service	–4.043	0.00005

For the mobile-service and casa-service, the rank results from Table 14 confirm that GraalVM consistently consumed more memory than JVM in all instances under high load conditions.

For the profile-service, while the null hypothesis (H_0) is rejected due to the significant p-value, the rank sums indicate that GraalVM consumed more memory than JVM in most cases, though there were a few instances where JVM consumed more memory. Overall, these findings suggest that JVM remains more memory-efficient than GraalVM, even under high traffic conditions.

6. Discussion of the performance differences between GraalVM and JVM in Java microservices

This study demonstrates notable differences between GraalVM and JVM in startup time, CPU usage, and memory consumption, revealing both advantages and trade-offs with practical implications for Java Spring Boot microservices. GraalVM consistently achieved faster application startup times, outperforming JVM by 25–27 milliseconds across all tested microservices as detailed in Table 1. This improvement can be explained by GraalVM's Ahead-of-Time (AOT) compilation, which reduces runtime overhead by precompiling applications into native executables. Fig. 3, 4 illustrate these results, with GraalVM showing a marked advantage in initialization speed. Faster startup times benefit microservices requiring minimal downtime. Compared to similar studies, the findings reinforce GraalVM's strengths in startup time improvements and its suitability for use cases where rapid initialization is critical.

While GraalVM demonstrated advantages in startup times, its impact on CPU usage revealed a nuanced relationship depending on services and load levels. At 100 users, Table 2 shows that GraalVM reduced CPU usage for mobile-service compared to JVM, with 0.7798 versus 0.8678. The statistical significance in Table 4, with a p-value of 0.014, confirms the impact, while the rank results in Table 3 indicate a predominantly positive outcome. There were 9 instances where GraalVM used less CPU than JVM with a rank sum of 45, and 12 instances where GraalVM used more CPU with a rank sum of 186, suggesting a marginal increase in CPU usage in some cases. For profile-service and casa-service, the p-values of 0.471 and 0.235 indicate no statistically significant differences at this load.

At 300 users, the impact of GraalVM becomes more evident across all services. Table 6 highlights statistically significant impacts with a p-value of 0.002 for mobile-service and less than 0.0001 for profile-service and casa-service. The rank results in Table 5 confirm that mobile-service benefits from GraalVM using less CPU than JVM in 17 instances with a rank sum of 205, although there are 4 instances where GraalVM used more CPU with a rank sum of 26. However, profile-service and casa-service show consistent increases in CPU usage by GraalVM, with all 21 instances indicating GraalVM used more CPU than JVM and a rank sum of 231 for each, indicating a clear negative impact for these services at this load.

At 600 users, the performance of GraalVM varies significantly. For mobile-service and profile-service, Table 8 shows no statistically significant differences with p-values of 0.191 and 0.229. The rank results in Table 7, however, reveal mixed outcomes for mobile-service, with 4 instances where GraalVM used less CPU than JVM with a rank sum of 78 and 17 instances where GraalVM used more CPU with a rank sum of 153. Profile-service exhibits a similar trend, with 6 instances where GraalVM used less CPU than JVM with a rank sum of 81 and 15 instances where GraalVM used more CPU with a rank sum of 150, indicating a moderate negative impact. For casa-service, the p-value of less than 0.0001 and rank results with 21 instances of GraalVM using more CPU than JVM and a rank sum of 231 confirm a consistently negative impact, with GraalVM using more CPU in all cases.

These findings underscore the variability in GraalVM's CPU usage impacts. While GraalVM offers efficiency improvements under specific conditions, particularly for mobile-service at lower loads, it exhibits higher CPU usage in resource-intensive services like casa-service, especially at higher loads. This variability highlights the importance of evaluating workload characteristics and service demands when deploying GraalVM to ensure optimal performance.

Memory consumption further adds to this complexity, as GraalVM consistently required more memory than JVM across all user loads and services. Table 9 highlights these differences, showing significant increases in memory usage for GraalVM compared to JVM. For example, in the casa-service under a 600-user load, memory usage increased from 198.47 MB on JVM to 591.38 MB on GraalVM, representing the most significant difference observed. Similarly, in the profile-service, memory usage rose from 193.67 MB on JVM to 239.05 MB on GraalVM under the same load.

The Wilcoxon Signed Rank results in Tables 10, 12, 14 confirm that GraalVM consistently used more memory than JVM, with all instances showing GraalVM>JVM across all services and load levels. This finding is further supported by the statistical analyses in Tables 11, 13, 15, which show significant differences in memory consumption for all microservices, with p-values well below 0.01 across all scenarios.

These results can be attributed to GraalVM's aggressive resource allocation strategy, which prioritizes computational performance and optimization over memory efficiency. While this strategy may enhance speed and throughput, it results in notably higher memory consumption, particularly under high-load scenarios. For instance, mobile-service, which exhibited a smaller difference at lower loads, still saw memory consumption rise from 165.43 MB on JVM to 201.90 MB on GraalVM under a 600-user load.

These findings suggest that GraalVM is better suited for scenarios that prioritize computational performance and efficiency over resource conservation. While the higher memory consumption may not be a concern for systems with sufficient resources, it presents a significant trade-off in resource-constrained environments. Additionally, these results contribute to a broader understanding of the trade-offs involved, as previous studies have often overlooked memory consumption in their evaluations of runtime environments like GraalVM.

By addressing the challenge of improving startup times, the study aligns with the objectives outlined. GraalVM's AOT compilation effectively reduces initialization delays, providing a solution to the problematic aspect of maintaining system efficiency during rapid user growth. However, the trade-offs in memory consumption suggest that this solution partially addresses the issue, leaving room for further optimization. Compared to similar studies, the results provide a more detailed understanding of GraalVM's balance of benefits and drawbacks, emphasizing the importance of aligning its adoption with specific microservice requirements.

By addressing the challenge of improving startup times, the study highlights GraalVM's AOT (Ahead-of-Time) compilation, which effectively reduces initialization delays, as corroborated by previous research [6], demonstrating significant startup time improvements in similar Java-based microservices. This provides a viable solution to the problematic aspect of maintaining system efficiency during rapid user growth. However, consistent with findings in other studies [7], our analysis shows that while startup performance improves, there are notable trade-offs in memory consumption, particularly under high-load conditions.

Compared to prior work emphasizing GraalVM's efficiency in CPU usage under medium loads [9], our study extends this understanding by providing a more granular analysis across low, medium, and high-load scenarios. The trade-offs in memory consumption highlighted in earlier research [4] align with our findings, suggesting that while GraalVM offers advantages in CPU efficiency, its adoption requires careful consideration of specific resource constraints and service demands. The results contribute to the broader discourse by presenting a detailed understanding of GraalVM's balance of benefits and drawbacks, emphasizing the importance of aligning its adoption with the unique requirements of Java microservices. Future research could further explore strategies to mitigate the observed memory trade-offs, building on the foundation provided by recent studies [8].

Despite its contributions, this study has limitations that must be considered. The findings are specific to three microservices and may not generalize to other application types or workloads. Environmental factors such as hardware and network configurations, although controlled here, could significantly affect reproducibility in real-world scenarios. The focus on user loads of 100, 300, and 600 excludes extreme or highly variable conditions, leaving questions about performance scalability unanswered. Moreover, the absence of compute- or I/O-intensive tasks limits the generalizability of these findings to broader application types. An additional limitation is GraalVM's higher memory consumption, which could lead to increased infrastructure costs, a factor not analyzed in detail. The statistical results in Tables 10, 12, 14 reinforce this observation, showing consistently higher memory usage with GraalVM under all tested scenarios. Furthermore, the manual adjustments required for configuring GraalVM, such as modifying logging frameworks and metadata for reflection, add complexity to its adoption in production environments.

These limitations suggest several avenues for future research. To provide a more holistic evaluation, future studies could explore additional performance metrics, such as latency, throughput, and energy consumption. Testing broader workloads, including data-intensive and real-time streaming applications, would help assess GraalVM's versatility across diverse use cases. Optimization techniques, such as fine-tuning garbage collection and memory allocation, could mitigate its high memory consumption, enhancing its applicability in resource-constrained deployments. Scaling the study to include larger datasets and multi-region deployments would also offer valuable insights into GraalVM's potential in distributed systems. Finally, automation tools to streamline GraalVM's configuration could reduce its complexity, making it a more accessible choice for developers.

By addressing these aspects, this study contributes to a nuanced understanding of GraalVM's practical benefits and trade-offs. Future research building on these findings can offer actionable recommendations for optimizing Java Spring Boot microservices in dynamic and demanding environments, paving the way for improved performance and resource management in modern microservice architectures.

7. Conclusions

1. GraalVM significantly improved application startup times across all microservices, with average reductions of 25–27 seconds. The paired sample T-test validated these results, showing statistically significant differences with p-values below 0.001. The observed mean differences were 25.090 seconds for mobile-service, 27.069 seconds for profile-service, and 26.782 seconds for casa-service. These findings indicate that GraalVM is particularly effective in scenarios requiring rapid initialization, such as frequent redeployments or dynamic scaling, providing a clear advantage over JVM in improving application responsiveness.

These findings highlight GraalVM's advantage in scenarios requiring rapid initialization, such as frequent redeployments or dynamic scaling.

2. CPU usage analysis using the Wilcoxon Signed Rank Test revealed varying performance patterns between GraalVM and JVM across different load conditions:

a) low load (100 users): GraalVM exhibited marginally lower CPU usage in mobile-service, with a significant difference observed (z-value=–2.460, p-value=0.014). However, no significant differences were found for profile-service (p-value=0.471) and casa-service (p-value=0.235);

b) medium load (300 users): JVM outperformed GraalVM across all services, with significant differences detected in profile-service and casa-service (z-value=–4.052, p-value=0.00005). Mobile-service also showed a significant advantage for JVM (z-value=–3.123, p-value=0.002);

c) high load (600 users): JVM maintained superior CPU efficiency, particularly in casa-service (z-value=–4.050, p-value=0.00005). Differences in mobile-service and profile-service under high load were not statistically significant.

These findings indicate that while GraalVM offers competitive CPU performance under low and medium loads in specific contexts, JVM consistently demonstrates superior efficiency under high-load scenarios, especially in CPU-intensive services such as casa-service. The detailed rank values and statistical summaries provide further insight into the comparative performance of the two runtimes.

3. Memory usage analysis across various load scenarios for mobile-service, profile-service, and casa-service revealed that GraalVM consistently utilized significantly more memory compared to JVM. The findings, validated by the Wilcoxon Signed Rank Test, are summarized as follows:

a) low load (100 users): GraalVM exhibited substantially higher memory consumption across all services, with mobile-service consuming 241.4048 MB compared to JVM's 154.0767 MB, profile-service using 216.4286 MB compared to JVM's 145.6524 MB, and casa-service utilizing 275.0505 MB compared to JVM's 150.3090 MB. The statistical analysis confirmed significant differences in all cases (z-values≤–4.029, p-values=0.00005);

b) medium load (300 users): GraalVM's memory consumption remained significantly higher, with mobile-service at 192.3571 MB compared to JVM's 172.2514 MB, profile-service at 182.9048 MB versus JVM's 164.2295 MB, and casa-service at 200.3333 MB compared to JVM's 167.5210 MB. Statistical significance was observed across all services (z-values≤–3.123, p-values≤0.002);

c) high load (600 users): the difference in memory usage was most pronounced under high-load conditions. GraalVM used 201.9048 MB versus JVM's 165.4314 MB for mobile-service, 239.0476 MB versus JVM's 193.6714 MB for profile-service, and a dramatic 591.3810 MB versus JVM's 198.4733 MB for casa-service. All differences were statistically significant (z-values≤–4.030, p-values≤0.00005).

These results underscore GraalVM's strategy of upfront memory allocation, which enhances application responsiveness at the cost of increased memory consumption. While this characteristic can be beneficial in scenarios prioritizing speed, the higher memory overhead may pose challenges in resource-constrained environments.

Conflict of interest

The authors declare that they have no conflict of interest in relation to this research, whether financial, personal, authorship or otherwise, that could affect the research and its results presented in this paper.

Financing

The study was financially supported by BINUS University, Jakarta, Indonesia.

Data availability

The manuscript has data included as electronic supplementary material.

Use of artificial intelligence

The authors confirm that they did not use artificial intelligence technologies when creating the current work.

References

1. Grigorescu, A., Oprisan, O., Lincaru, C., Pirciog, C. S. (2023). E-Banking Convergence and the Adopter's Behavior Changing Across EU Countries. Sage Open, 13 (4). https://doi.org/10.1177/21582440231220455
2. Ionașcu, A. E., Gheorghiu, G., Spătariu, E. C., Munteanu, I., Grigorescu, A., Dănilă, A. (2023). Unraveling Digital Transformation in Banking: Evidence from Romania. Systems, 11 (11), 534. https://doi.org/10.3390/systems11110534
3. Kim, L., Jindabot, T. (2022). Evolution of customer satisfaction in the e-banking service industry. Innovative Marketing, 18 (1), 131–141. https://doi.org/10.21511/im.18(1).2022.11
4. Aydemir, F., Başçiftçi, F. (2022). Building a Performance Efficient Core Banking System Based on the Microservices Architecture. Journal of Grid Computing, 20 (4). https://doi.org/10.1007/s10723-022-09624-z
5. Yin, P., Cheng, J. (2023). A MySQL-Based Software System of Urban Land Planning Database of Shanghai in China. Computer Modeling in Engineering & Sciences, 135 (3), 2387–2405. https://doi.org/10.32604/cmes.2023.023666

6. Larsson, R. (2020). Evaluation of GraalVM Performance for Java Programs. Available at: http://www.diva-portal.org/smash/get/diva2:1457592/FULLTEXT01.pdf
7. Fong, F., Raed, M. (2021). Performance comparison of GraalVM, Oracle JDK and OpenJDK for optimization of test suite execution time. Available at: https://www.diva-portal.org/smash/get/diva2:1597213/FULLTEXT01.pdf
8. Kozak, D., Jovanovic, V., Stancu, C., Vojnar, T., Wimmer, C. (2023). Comparing Rapid Type Analysis with Points-To Analysis in GraalVM Native Image. Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, 129–142. https://doi.org/10.1145/3617651.3622980
9. Wyciślik, Ł., Latusik, Ł., Kamińska, A. M. (2023). A Comparative Assessment of JVM Frameworks to Develop Microservices. Applied Sciences, 13 (3), 1343. https://doi.org/10.3390/app13031343
10. Sipek, M., Mihaljevic, B., Radovan, A. (2019). Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM. 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 1671–1676. https://doi.org/10.23919/mipro.2019.8756917
11. Kreindl, J., Rigger, M., Mössenböck, H. (2018). Debugging native extensions of dynamic languages. Proceedings of the 15th International Conference on Managed Languages & Runtimes - ManLang '18, 1–7. https://doi.org/10.1145/3237009.3237017
12. Li, S., Jia, Z., Li, Y., Liao, X., Xu, E., Liu, X. et al. (2019). Detecting Performance Bottlenecks Guided by Resource Usage. IEEE Access, 7, 117839–117849. https://doi.org/10.1109/access.2019.2936599
13. Sampaio, A. R., Rubin, J., Beschastnikh, I., Rosa, N. S. (2019). Improving microservice-based applications with runtime placement adaptation. Journal of Internet Services and Applications, 10 (1). https://doi.org/10.1186/s13174-019-0104-0
14. Cosmina, I. (2021). Java 17 for Absolute Beginners. Apress, 801. https://doi.org/10.1007/978-1-4842-7080-6
15. Indrianto, I. (2023). Performance testing on web information system using apache jmeter and blazemeter. Jurnal Ilmiah Ilmu Terapan Universitas Jambi, 7 (2), 138–149. https://doi.org/10.22437/jiituj.v7i2.28440
16. Gravetter, F. J., Wallnau, L. B. (2017). Statistics for the Behavioral Sciences. Cengage Learning. Available at: http://ndl.ethernet.edu.et/bitstream/123456789/29095/1/Frederick%20J%20Gravetter_2017.pdf
17. Deshpande, J. V., Naik-Nimbalkar, U., Dewan, I. (2018). Nonparametric Statistics Theory and Methods. World Scientific. Available at: https://sadbhavnapublications.org/research-enrichment-material/2-Statistical-Books/Nonparametric-Statistics-Theory-and-Methods.pdf
18. Corder, G. W., Foreman, D. I. (2014). Nonparametric Statistics. A Step by Step Approach. Wiley. Available at: https://faculty.ksu.edu.sa/sites/default/files/nonparametric_statistics_a_step-by-step_approach.pdf