JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 37 (2025) 484–491
Received: 5 July 2025
Accepted: 7 September 2025

Comparative Performance Analysis of Spring Boot and Quarkus Frameworks in Java Applications

Analiza porównawcza szkieletów Spring Boot i Quarkus pod kątem wydajności aplikacji Java

Grzegorz Szymanek*, Jakub Smołka

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
The paper presents comparative performance characterization of two Java application development frameworks, Spring Boot and Quarkus. A representative reference application was implemented using both frameworks to enable such a comparison. The study entailed measurements in terms of multiple metrics, including compilation time, initialization time for an application, final file sizes, CPU and RAM consumption, latency in HTTP response, throughput, and efficiency of database queries. Tests were run on both JAR and native image versions with varying loads. Results convincingly showed the benefits of native Quarkus in startup performance and resource utilization. Spring Boot is still a proven option with a broader tooling universe supporting it, though. This research gives interesting input to decide on the best technology for modern Java applications.

Keywords: Java; Spring Boot; Quarkus

Streszczenie
Artykuł przedstawia porównawczą charakterystykę wydajności dwóch szkieletów programistycznych do tworzenia aplikacji w języku Java: Spring Boot i Quarkus. W celu umożliwienia takiego porównania zaimplementowano reprezentatywną aplikację referencyjną w obu technologiach. Badanie obejmowało pomiary w różnych aspektach, takich jak czas kompilacji, czas inicjalizacji aplikacji, rozmiar pliku wynikowego, zużycie CPU i pamięci RAM, opóźnienie w odpowiedzi HTTP, przepustowość oraz efektywność zapytań do bazy danych. Testy przeprowadzono zarówno dla wersji JAR, jak i obrazu natywnego, przy różnych poziomach obciążenia. Wyniki jednoznacznie wskazały na zalety natywnego Quarkusa pod względem szybkości uruchamiania i efektywności wykorzystania zasobów. Mimo to Spring Boot pozostaje sprawdzonym rozwiązaniem, wspieranym przez szerszy ekosystem narzędziowy. Niniejsze badanie dostarcza cennych wskazówek przy wyborze odpowiedniej technologii dla nowoczesnych aplikacji Java.

Słowa kluczowe: Java; Spring Boot; Quarkus

*Corresponding author
Email address: s95587@pollub.edu.pl (G. Szymanek)

1. Introduction

The development of information technologies, along with the increasingly high demands of users for web applications, has placed software performance as an increasingly critical concern. In business settings and high-demand programs, the responsiveness of applications, effective use of resources, and flexibility in deployment are of utmost importance. In such a situation, the proper development tools and frameworks for creating web applications are crucial.

Java, one of the main drivers in the evolution of information systems over the years, relies on a number of frameworks that support the development of microservices architecture based applications. Among the leading and most popular solutions is Spring Boot, a stable and feature rich framework with a rich ecosystem of libraries and tools. In the context of evolving requirements such as startup time reduction and memory performance enhancement in containerized applications, a new framework has been introduced: Quarkus, which is "Supersonic Subatomic Java" tailored for the cloud age. Low application startup times and the capacity to generate native images through GraalVM, significantly enhancing resource consumption and execution speed, are Quarkus' main features.

1.1. The aim and object of the research

The primary objective of this study is to conduct an analysis and comparison of the performance metrics of Spring Boot and Quarkus frameworks in the aspect of developing contemporary Java based web applications. The analysis encompasses measuring the values of diverse technical parameters, including startup time, memory and CPU utilization, latency of HTTP responses, throughput, and data manipulation operation efficiency in JAR based and GraalVM native environments. From a literature review of recent research, the following hypotheses have been formulated:

- Quarkus supports faster application startup time and reduced resource consumption by way of native compilation through GraalVM than Spring Boot,
- though Spring Boot shows greater memory consumption and longer setup, it exhibits greater stability

during operation, which makes it a well known choice for big enterprise systems with long-term support,
- Quarkus has shorter HTTP response times than Spring Boot.

1.2. Literature Review

Numerous benchmarks have compared modern Java frameworks for their performance, startup time, and suitability to cloud native architectures. Of these, Spring Boot and Quarkus are two of the most widely benchmarked technologies.

A number of benchmarks have demonstrated that Quarkus has better runtime performance in the context of initialization time, memory consumption, and processor overhead—especially when compiled to native images via GraalVM [1-3]. All these make it extremely well suited to be utilized in serverless and containerized environments. Spring Boot, on the other hand, continues to exhibit better robustness under high loads as well as in sophisticated application configurations [4-6].

The porting of legacy microservices to the Quarkus framework has exhibited considerable reduction in deployment time and CPU utilization but can be associated with higher memory consumption [5]. Spring Boot, on the other hand, remains the first choice in enterprise domains, i.e., finance and government, because of its matured ecosystem and enormous capabilities for integration and configuration [7-9].

They are both heavily used for IoT and microservice designs. Research attests to the prowess of Quarkus in light weight deployments due to its minimal resource footprint, making it ideal for sensor networks and edge computing [2], [10]. Spring Boot remains a suitable contender due to its rich support for distributed systems and reliability for long running services [11].

Spring Boot is defined by its large testing framework, including tools such as JUnit, Mockito, and SpringRunner that enable comprehensive unit, integration, and system testing [12]. The use of these tools results in better software quality and reduced development time. An application based on Spring Boot required less code, indicating the advantages of its rich ecosystem and built in dependencies [13]. On the other hand, while Quarkus shows high performance, it may require more manual configuration in testing scenarios.

A study that included execution efficiency showed that Spring based applications consumed more energy and executed slower than applications that were built without using the framework. This was mainly due to reflection based runtime processing [14]. Although Spring increases developer productivity, it may not be optimally suited for energy limited environments.

Quarkus has also experimented with virtual threads as a third option to both blocking and reactive paradigms. Findings indicate that virtual threads can enhance transparency and simplicity of development but may be affected by higher garbage collection pressures under heavy loads [15]. Reactive paradigms, especially as realized in Spring Boot and Quarkus, always perform better than conventional threading models in scenarios with intensive I/O operations [16].

While Play Framework would be more performant in light, lowload applications, Spring Framework outperforms it when system complexity and user concurrency are greater [17]. In modeling business process fullstack applications, Spring Boot was compared with emerging stacks and is still competitive in medium scale scenarios [18].

Spring Boot's utilization along with other abstraction layers for the database, such as MyBatis and Hibernate, contributes to its performance overall. It has been discovered that certain combinations offer higher reliability and better speed, especially those involving caching mechanisms [19].

Several assessments highlighted the strong market position of Spring Boot. It has been heavily used in critical sectors like finance, healthcare, where the need for security, scalability, and multithreading support is top priority [7], [8]. In contrast, Quarkus, being newer, is increasingly popular because of its cloud native foundation and ability to scale effectively with minimal resources [1], [3].

In general, Quarkus performs better, more efficiently, and is more compatible with current cloud environments, particularly when it's natively compiled. Spring Boot, however, is still the most widely used option of large scale enterprise applications due to its reliability, mature ecosystem, and established tooling. It all hinges on the particular needs of the application, such as deployment environment, performance limitations, and developers' skillset, to prefer one over the other [20].

2. Research Methodology

This chapter presents the research methodology followed to assess and compare the performance of the Spring Boot and Quarkus frameworks. The chapter defines the aim and scope of the research, test environment setup, experimental application design, performance test scenarios, and data collection and analysis techniques.

2.1. Research Objective and Scope

The main objective of this study is to conduct a comparative analysis of the performance of two most widely utilized Java based web application frameworks: Spring Boot and Quarkus. The comparison is based on an assessment of the performance of the two frameworks in the development and execution of the same e-commerce application, such as the management of orders, categories, and products.

To enable a fair comparison, both frameworks were tested in two run modes: the standard Java Archive (JAR) run mode and a native executable created with GraalVM. Having the comparison done in two modes enables the examination of not just the intrinsic performance traits but also the impact of native compilation on system resource utilization and application responsiveness.

2.2. Testing Environment

To try to maintain consistency and minimize the impact of external variables that may impact the outcomes, all of

the experiments were run on a single laptop in a controlled lab. The laptop was running Windows 11 Education (64-bit) and served as the only environment for development, execution, and monitoring of both application versions during the research. This environment was chosen to simulate a common developer workstation while having a reproducible environment for Spring Boot and Quarkus application performance measurement.

Table 1: Computer specification

Component	Specification
Processor	AMD Ryzen 7 8845HS 5.1 GHz, 8 cores, 16 threads
RAM	32 GB DDR5
Graphics Card	Nvidia RTX 4070 8GB
Storage	1 TB SSD
Operating System	Windows 11

Table 1 shows the key specifications of the test laptop, including the processor, memory, operating system, and storage configuration. All software tools and frameworks used by the experiments were executed natively on the host platform without virtualization, except for specifically containerized tests executed with Docker.

2.3. Application Design and Technology Stack

The application under test is a sample online shopping system with mocked product management, categories, and orders from customers. It was purposely designed to be a typical CRUD based business application and includes both read intensive and write intensive operations. This allowed performance testing across a broad spectrum of backend workloads. Figure 1 presents the relational database schema used by the application.

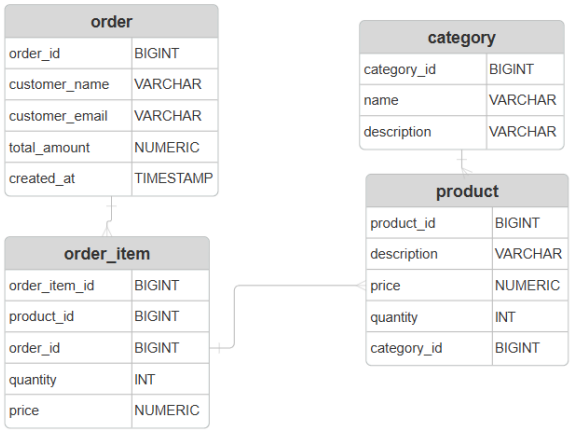

Figure 1: Relational database schema used in the tested application.

In order to ensure objective comparability, application logic, database schema, and functionality were standardized in both implementations, one Spring Boot based and the other Quarkus based. Both versions have the same RESTful endpoints and work against the same PostgreSQL relational database. Table 2 presents complete technology stack used in both implementations.

Table 2: Technology stack used in both Spring Boot and Quarkus implementations

Component	Version
Java	21
Maven	3.9.9
Gradle	8.13
Spring Boot	3.4.4
Quarkus	3.19.1
GraalVM	21
PostgreSQL	17.0
Docker	27.4.0

2.4. Test Scenarios and Performance Evaluation Approach

In order to comprehensively compare the performance of the applications built, several test scenarios were constructed to simulate real world workloads characteristic of CRUD based e-commerce web applications. The testing process involved the measurement of efficiency of both frameworks (Quarkus and Spring Boot) in various operational scenarios with standard metrics and monitoring tools (Grafana K6 and Prometheus). The assessment covered the following key areas:

1. Build performance: A comparison of the average compilation time using Maven and Gradle in both frameworks.
2. Startup performance: Native executable and JAR-based versions' application startup time measurement.
3. Artifact size comparison: Comparison of sizes of JAR files generated, native binaries, and Docker image sizes.
4. Database access performance: Measurement of the average time to read from and write to PostgreSQL database for specified load parameters.
5. Request performance testing:
    - GET requests – included three levels of complexity: the light load phase loads 15 categories and 1,000 products. The medium load handles 20 categories and 2,000 products. The heavy load runs an aggregation query to fetch the top 10 products based on 15,000 orders and 80,000 order items.
    - POST requests - creation of new orders containing multiple items and related stock updates,
    - DELETE requests - removal of products by ID, utilized for data cleanup simulation or admin activities,
    - PUT requestes - update records of existing products to match data changes,

- MIX requests - are a composite scenario that utilizes a series of POST, GET, PUT and DELETE requests, thereby covering the end to end lifecycle of a resource.

6. Resource usage: Monitoring CPU usage and RAM usage when executing tests.

All load tests based on HTTP were executed with k6, an open source performance testing tool. All scenarios were tested with three loads of users: 50, 500, and 1000 virtual users (VUs), with a fixed test duration of 2 minutes per iteration. These settings allowed response stability and throughput to be assessed against increasing levels of concurrency.

Further, resource consumption tests, specifically CPU and RAM usage, were also carried out as stress tests, which ran in 5 minute intervals, with two provided loads: 100 virtual users (VUs) and 1000 virtual users (VUs). These tests gave valuable insight into the long-term efficiency and scalability of each framework under continuous stress.

Behavior and resource metrics of applications were gathered by Prometheus and subsequently monitored by Grafana, enabling real time system performance monitoring with precision.

3. Results and Analysis

This chapter presents the findings derived from all experimental tests in this research. The measurements address various performance dimensions, such as construction time, initialization latency, resource utilization, request processing effectiveness, and database operations. The findings are organized by test type with a short analysis following every group of findings to point out important patterns of performance.

3.1. Build and Startup Performance

Figures 2 and 3 show the result of measuring the time to compile and start applications built with Spring Boot and Quarkus using different build configurations. Figure 2 presents the compilation times for applications built using Maven and Gradle, targeting both JAR files and native images. The fastest compilation was witnessed for Spring Boot JAR builds using Gradle, with an average time of 1.2 seconds, followed by Maven, at 4.2 seconds.

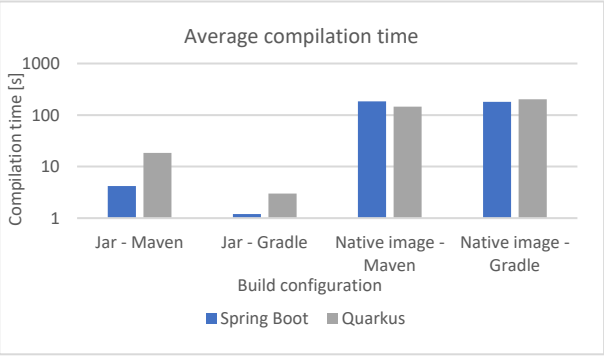

Figure 2: Average compilation time for Spring Boot and Quarkus applications depending on build configuration.

Quarkus JAR builds took 18.4 seconds with Maven and 3 seconds with Gradle. For native image, Quarkus build with Maven was the fastest at 146.8 seconds. Spring Boot build took 183.8 seconds with Maven and 180 seconds with Gradle, and Quarkus build with Gradle took 202.6 seconds.

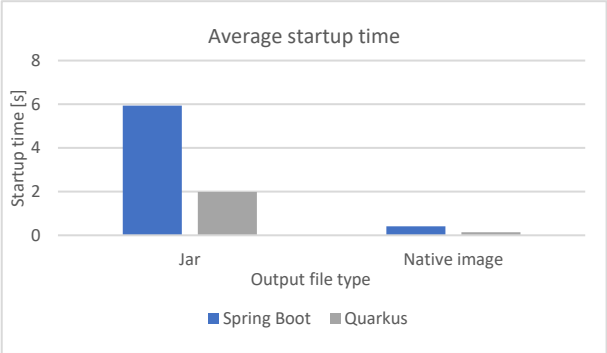

Figure 3: Average startup time for Spring Boot and Quarkus applications depending on output file type.

Figure 3 plots the average startup time to start an application versus the type of output file. Spring Boot application startup took 5.94 seconds with JAR files, while Quarkus took 1.99 seconds. Native images reduced the startup times significantly: Spring Boot native took 0.41 seconds, while Quarkus native took 0.14 seconds, demonstrating that Quarkus is better when it comes to startup times.

3.2. Artifact and Image Size

Figure 4 compares the sizes of the artifacts produced by Spring Boot and Quarkus when run for different build and deployment configurations, like normal JAR files, native binaries, and Docker images based on each type of output. For the JAR based deployments, the applications were bundled uber JARs containing all dependencies.

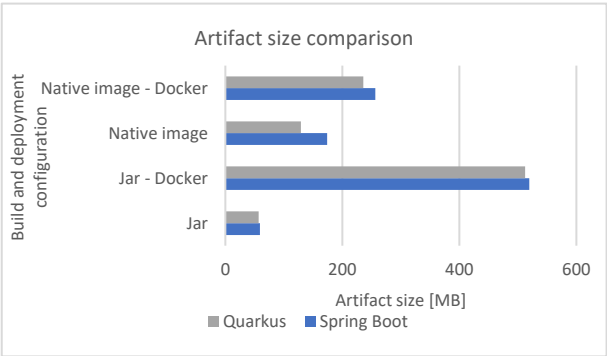

Figure 4: Comparison of artifact sizes for Spring Boot and Quarkus across different build and deployment configurations.

Regarding the size of the JAR file, Spring Boot produced somewhat larger artifacts (59.20 MB) than Quarkus (56.80 MB). The difference increased further in Dockerized JAR builds, where Spring Boot images weighed 519.51 MB, versus 512.44 MB for Quarkus.

For the compiled native binaries, the binaries were bigger for Spring Boot (174 MB) than for Quarkus (129 MB). This carried over to the native builds being

Dockerized, with Spring Boot's image weighing in at 256.35 MB, while Quarkus's image stayed close to 235.76 MB.

Results indicate that Quarkus will lead to marginally smaller artifacts and Docker images, particularly in native modes, something that can be advantageous when dealing with cloud environments and environments with strict bandwidth or storage limitations.

3.3. Database Access

Figure 5 shows a comparison of mean duration for database operations in relation to data retrieval and insertion using both frameworks on different build types (JAR and native image). During the test, 2000 products were written and read.

Quarkus demonstrated significantly better read performance compared to Spring Boot in both JAR and native image modes. The average data fetching time for Quarkus in native mode was only 7.6 ms, while Spring Boot took 11 ms. Similarly, in JAR mode, Quarkus outperformed Spring Boot with an average time of 12 ms compared to 14.6 ms. This indicates that Quarkus offers a noticeable performance advantage in simple data retrieval operations, regardless of the build type used.

While write operations typically involve more intensive use of database resources, the performance results show some variation between Spring Boot and Quarkus. Quarkus consistently achieved slightly better write times across both JAR and native builds. On average, Quarkus in native mode completed write operations in 1416 ms, compared to 1438.6 ms for Spring Boot. In JAR mode, Quarkus also outperformed Spring Boot with an average of 1383 ms versus 1403.4 ms. Although the differences are not dramatic, they do indicate that Quarkus has a slight edge in write performance under these test conditions.

The findings show that framework selection and build configuration have little impact on basic database interaction performance. It seems that for simple database usage, both Spring Boot and Quarkus offer comparable and consistent throughput.

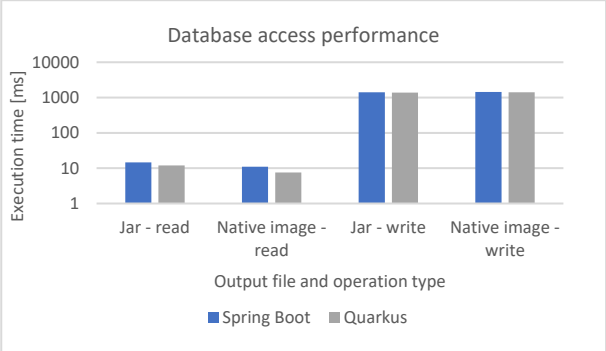

Figure 5: Database access performance (read and write) for Spring Boot and Quarkus using different output file types.

3.4. Request Handling Performance

During the conducted test, response times were analyzed for individual types of HTTP requests (GET, POST, PUT, DELETE and MIX) at three levels of load: 50, 500 and 1000 virtual users (VU). Figures 6 to 12 below present average response times for each type of request and configuration (Spring Boot / Quarkus, in JAR version and native image). The GET request workload was composed of three scenarios: the light scenario involved retrieving 15 categories along with 1000 associated products; the medium scenario consisted of retrieving 20 categories and 2000 products; and the heavy scenario included the retrieval of 15,000 orders and 80,000 order items.

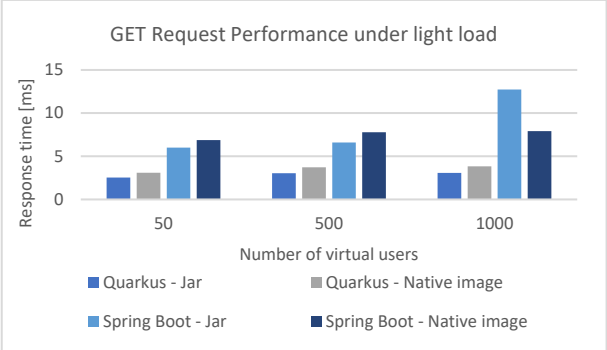

Figure 6: GET request response time under light load for Spring Boot and Quarkus (Jar and Native Image) across varying numbers of virtual users.

For the lighter GET queries (Figure 6), Quarkus in all the variants had a lower response time than Spring Boot, especially under the highest load. Differences were slight, however, for 50 and 500 users.

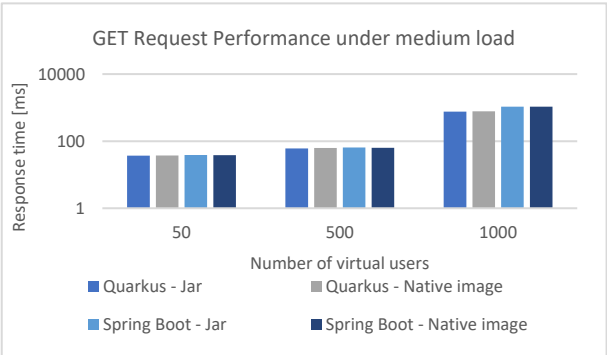

Figure 7: GET request response time under medium load for Spring Boot and Quarkus (Jar and Native Image) across different numbers of virtual users.

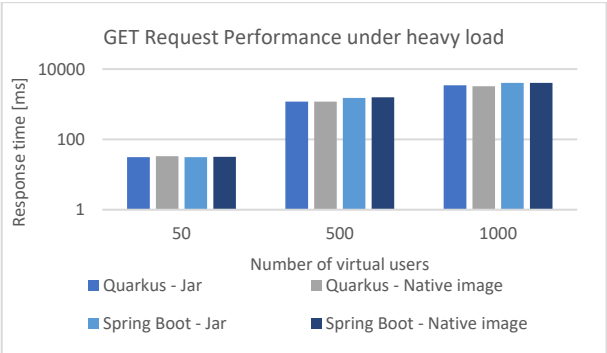

Figure 8: GET request response time under heavy load for Spring Boot and Quarkus (Jar and Native Image) across different numbers of virtual users.

For moderately complicated GET requests (Figure 7), frameworks' differences started to reveal themselves clearly. Native Quarkus gained even by several hundred milliseconds shorter response time compared to Spring Boot.

In the case of high load GET requests (Figure 8), namely those being aggregation queries, the variations were more noticeable. Spring Boot showed much higher response times, particularly under the load of 1000 users, where the average response time was 4040 ms (JAR) and 3250 ms (Quarkus Native). Quarkus Native version was the only one not reaching 100% success rate, with 97%.

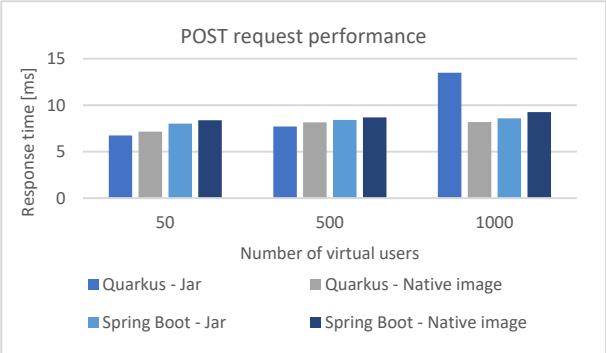

Figure 9: POST request response time for Spring Boot and Quarkus (Jar and Native Image) under different user loads.

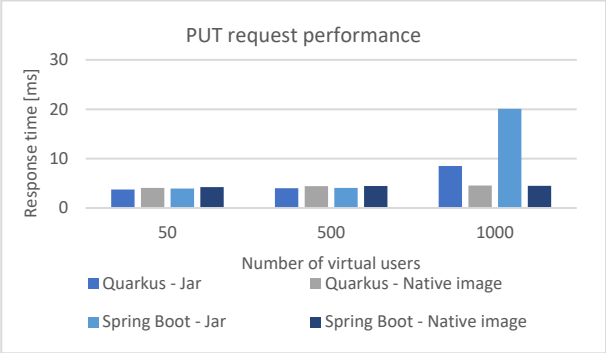

Figure 10: PUT request response time for Spring Boot and Quarkus (Jar and Native Image) across varying numbers of virtual users.

For the POST (Figure 9) and PUT (Figure 10) requests, Quarkus was faster in the majority of cases, with the exception of a single spike in time for Quarkus JAR (PUT, 1000 VU).

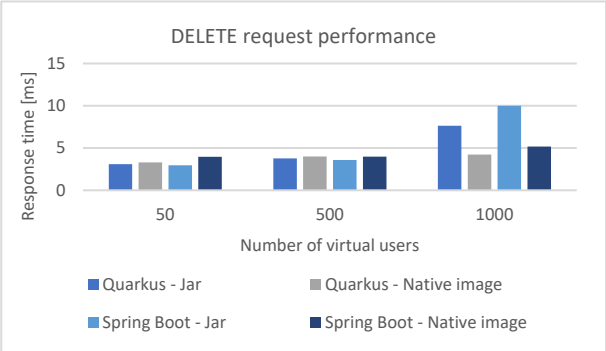

Figure 11: DELETE request response time for Spring Boot and Quarkus (Jar and Native Image) across different numbers of virtual users.

DELETE (Figure 11) presented the same tendencies – Quarkus offered lower latency at every level of load. Mixed CRUD scenario (POST → GET → PUT → DELETE), shown in Figure 12, exhibited the highest discrepancies. Spring Boot, at 1000 users, was achieving an average response time of over 190 ms (native) and 100 ms (JAR), while Quarkus in native version was achieving around 6 ms. That points to a clear advantage of Quarkus for high traffic conditions and complex operations.

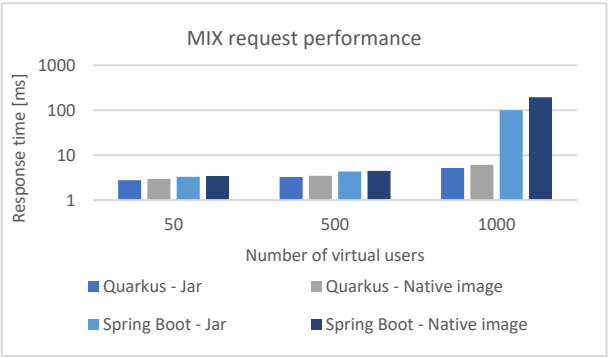

Figure 12: Mixed request response time for Spring Boot and Quarkus (Jar and Native Image) across different numbers of virtual users.

3.5. Resource usage

Resource consumption metrics, which are illustrated in Figures 13 and 14, present the average CPU and memory consumption of each framework variant under high load over a duration of 5 minutes conducted with 100 and 1000 virtual users. The load test involved retrieving 100 categories along with 20,000 products, as well as sorting the categories.

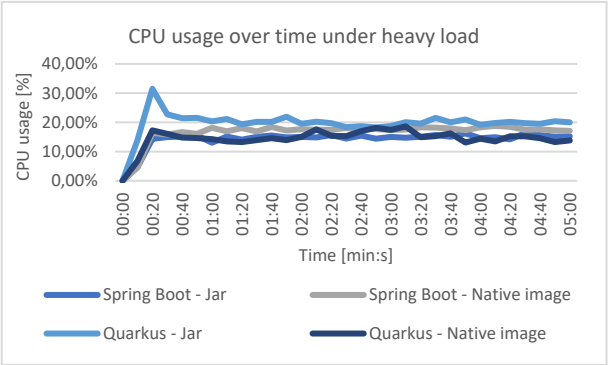

Figure 13: CPU usage over time under heavy load for Spring Boot and Quarkus (Jar and Native Image).

CPU usage under 1000 VUs was the highest for Quarkus JAR (20.3%), with Spring Boot Native following closely (17.09%). Quarkus Native had a little lower usage (14.94%), and the lowest was Spring Boot JAR (14.64%). As expected, the Quarkus JAR consumed more CPU than its native counterpart, which is consistent with the fact that native code generally executes more efficiently than code interpreted by the JVM, especially under high load.

Figure 14 illustrates memory usage under the same testing conditions.

Memory consumption analysis reveals a stark difference between native and JAR variants. Spring Boot JAR

consumed the most memory (628.18 MB), while Quarkus Native consumed the lowest memory (355.76 MB). The difference was even more significant at low load (100 VUs), where native images consumed up to 70% less memory compared to their respective JAR variants.

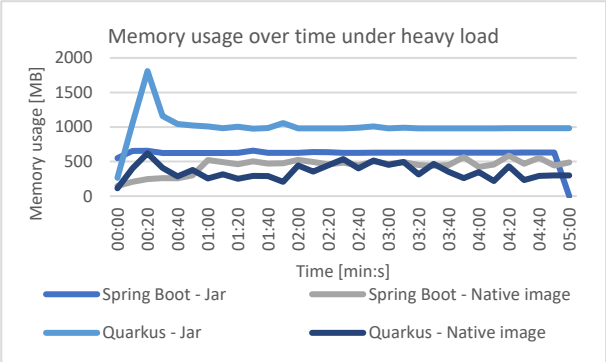

Figure 14: Memory usage over time under heavy load for Spring Boot and Quarkus (Jar and Native Image).

4. Discussion

The performance difference between Spring Boot and Quarkus frameworks revealed a set of trade offs according to the literature.

In startup and build performance analysis, Quarkus, when natively compiled, consistently showed the lowest startup times—confirming literature research that GraalVM greatly improves startup latency and resource consumption [1], [3]. Yet, as illustrated in our findings, these advantages were accompanied by increased compilation times and more complicated build pipelines.

On the memory and CPU side, our findings match those in studies such as [2] and [3], where Quarkus native images consumed less RAM during low to medium load. But during high load (1000 VUs), memory consumption for Quarkus JARs increased exponentially—surpassing Spring Boot in some cases. This corroborates previous observations of higher memory consumption for Quarkus under high volume tasks [5].

In HTTP request processing, Quarkus outperformed Spring Boot in most of the GET, POST, and DELETE operations in both latency and throughput. This aligns with previous research proving Quarkus' edge in response time and throughput [1], [4]. However, Spring Boot demonstrated higher stability under the combined CRUD and heavy GET loads, achieving a 100% request success rate compared to around 97% for Quarkus Native - proving its strength over heavy complexity and continuous load as stated in [6], [7].

Our test findings corroborated the hypothesis H1: that Quarkus native images offer better startup and runtime performance. They also lent partial support to H2: Spring Boot offered greater stability and predictable performance, especially under load. H3, for HTTP response time, also stood—Quarkus consistently yielded shorter response times in nearly all categories. Lastly, the findings mirror the wider industry trend: Quarkus is increasingly the best fit for new, lightweight, cloud native applications, with Spring Boot still dominant in enterprise use cases necessitating heavy tooling and long term support [8-10].

5. Conclusions

The aim of this study was to directly compare the performance potential of two popular Java frameworks, Spring Boot and Quarkus, by implementing the same web application using both technologies and executing an extensive set of benchmarks. The benchmarks covered the more traditional JAR based deployments along with native images compiled with GraalVM, providing a holistic picture of how each stacks up under different conditions.

The findings indicated that Quarkus, and more so its native version, recorded much quicker application startup times and is thus well adapted to cloud native environments where low latency and effective scalability are prioritized. Spring Boot, on its part, registered more consistent and stable performance in longer and intricate processes, an indicator of its higher appropriateness for enterprise type applications that demand constant long term reliability.

In comparing the handling of HTTP requests on different endpoints—starting from simple read access to compound write operations and CRUD application—Quarkus consistently demonstrated better response times and greater throughput levels. Spring Boot, nevertheless, exhibited good stability and consistent response rates, even at high load levels.

Resource usage patterns analysis showed that Quarkus native runs typically have less memory usage under lighter load levels, whereas Spring Boot shows less variable CPU and RAM consumption under extended periods of traffic. This balance reflects Quarkus' strength in ephemeral and containerized deployments, as Spring Boot remains a solid choice for traditional infrastructure under high service duration conditions.

In short, the noted differences in performance between the two frameworks give weight to the idea that Quarkus may be more appropriate for modern, cloud centric applications where fast startup times and low resource overhead are paramount. Spring Boot, on the other hand, is still a viable and flexible choice for complex applications that require high configurability, stability, and ecosystem support. The ultimate selection of such technologies should be based on the particular requirements of the target system and the application environment where it will be used.

References

[1] Ł. Wyciślik, Ł. Latusik, A. M. Kamińska, A comparative assessment of JVM frameworks to develop microservices, Appl. Sci. 13 (2023) 1343, https://doi.org/10.3390/app13031343.

[2] P. Plecinski, N. Bokla, T. Klymkovych, M. Melnyk, W. Zabierowski, Comparison of representative microservices technologies in terms of performance for use for projects based on sensor networks, Sensors 22 (2022) 7759, https://doi.org/10.3390/s22207759.

[3] M. Šipek, D. Muharemagić, B. Mihaljević, A. Radovan, Enhancing performance of cloud-based software

applications with GraalVM and Quarkus, In 2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO) IEEE (2020) 1746– 1751, https://doi.org/10.23919/MIPRO48935.2020.9245290.

[4] M. Jeleń, M. Dzieńkowski, The comparative analysis of Java frameworks: Spring Boot, Micronaut and Quarkus, J. Comput. Sci. Inst. 21 (2021) 287–294, https://doi.org/10.35784/jcsi.2724.

[5] B. Silva, P. Carvalho, Migration of a microservice from Payara Micro to Quarkus and performance analysis, In 2022 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME) IEEE (2022) 1- 5, https://doi.org/10.1109/ICECCME55909.2022.9988587.

[6] O. C. Novac, D. Ghiurău, M. C. Novac, C. E. Gordan, M. Oproescu, G. Bujdoso, Comparison of Node.Js and Spring Boot in web development, In 2023 15th International Conference on Electronics, Computers and Artificial Intelligence (ECAI) IEEE (2023) 1–7, https://doi.org/10.1109/ECAI58194.2023.10194025.

[7] D. Choma, K. Chwaleba, M. Dzieńkowski, The efficiency and reliability of backend technologies: Express, Django, and Spring Boot, Informatyka Automatyka Pomiary Gosp. Ochr. Środowiska 13(4) (2023) 73–78.

[8] M. Mythily, A. S. Arun Raj, I. T. Joseph, An analysis of the significance of Spring Boot in the market, In 2022 International Conference on Inventive Computation Technologies (ICICT) IEEE (2022) 1277–1281, https://doi.org/10.1109/ICICT54344.2022.9850910.

[9] M. Klymash, I. Tchaikovskyi, O. Hordiichuk-Bublivska, Y. Pyrih, Research of microservices features in information systems using Spring Boot, In 2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T) (2020) 507–510, https://doi.org/10.1109/PICST51311.2020.9467911.

[10] S. Pallewatta, V. Kostakos, R. Buyya, MicroFog: A framework for scalable placement of microservices-based IoT applications in federated fog environments, J. Syst. Softw. 209 (2023) 111910, https://doi.org/10.1016/j.jss.2023.111910.

[11] H. Suryotrisongko, D. P. Jayanto, A. Tjahyanto, Design and development of backend application for public complaint systems using microservice Spring Boot, Procedia Comput. Sci. 124 (2017) 736–743.

[12] M. Mythily, A. D. D. C. Durai, V. R. Kanakala, I. T. Joseph, R. Nambiar, An extensive review of Spring Boot testing based on business requirements of the software, In 2023 4th International Conference on Smart Electronics and Communication (ICOSEC) IEEE (2023) 1547–1553, https://doi.org/10.1109/ICOSEC58147.2023.10276283.

[13] S. Mohan, K. Goswami, Performance Analysis and Comparison of Node.Js and Java Spring Boot in Implementation of Restful Applications, Softw. Pract. Exper. (2025), https://doi.org/10.1002/spe.3418.

[14] C. Calero, M. Polo, M. Moraga, Investigating the impact on execution time and energy consumption of developing with Spring, Sustain. Comput. Inform. Syst. 32 (2021) 100603, https://doi.org/10.1016/j.suscom.2021.100603.

[15] A. Navarro, J. Ponge, F. Le Mouël, C. Escoffier, Considerations for integrating virtual threads in a Java framework: A Quarkus example in a resource-constrained environment, In 17th ACM International Conference on Distributed and Event-Based Systems (DEBS'2023) (2023), https://doi.org/10.1145/3583678.3596895.

[16] A. Sharma, K. Tahiliani, G. P. Dubey, Reactive-optimized sentence detection in Kubernetes using OpenNLP and native GraalVM image with framework metric comparison, In 2023 4th International Conference for Emerging Technology (INCET) IEEE (2023) 1–9, https://doi.org/10.1109/INCET57972.2023.10170347.

[17] M. Gajewski, W. Zabierowski, Analysis and comparison of the Spring framework and Play framework performance, used to create web applications in Java, In IEEE XVth International Conference on the Perspective Technologies and Methods in MEMS Design (2019) 170–173, https://doi.org/10.1109/MEMSTECH.2019.8817390.

[18] Y. Jayawardana, R. Fernando, G. Jayawardena, D. Weerasooriya, I. Perera, A full stack microservices framework with business modelling, In 2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer) IEEE (2018) 78–85, https://doi.org/10.1109/ICTER.2018.8615473.

[19] A. Ginanjar, M. Hendayun, Spring framework reliability investigation against database bridging layer using Java platform, Procedia Comput. Sci. 161 (2019) 1036–1045, https://doi.org/10.1016/j.procs.2019.11.214.

[20] A. Poniszewska-Marańda, K. Stępień, M. Głowiński, Function analysis of web services based on REST protocol with selected frameworks, In 2021 International Conference on Software, Telecommunications and Computer Networks (SoftCOM) IEEE (2021) 1–6, https://doi.org/10.23919/SoftCOM52868.2021.9559090.