

# A Comparative Study of Response Time and Throughput in Spring Boot and Quarkus REST APIs

Arpan Chakraborty<sup>1</sup>

<sup>1</sup> Student, Department of Computer Science and Engineering

\*\*\*

**Abstract**—This paper presents a comprehensive performance comparison between Spring Boot and Quarkus, two widely used Java backend frameworks. Identical REST APIs were built in both frameworks, including a database-connected `/users` POST endpoint using H2 in-memory storage. Load testing was conducted using Apache JMeter across multiple concurrency levels (10, 50, 100, and 200 concurrent users), while heap memory consumption was measured using VisualVM. Both frameworks were also containerized using Docker to compare startup time under standard JVM and Docker deployment modes. For the simple `/hello` endpoint, both frameworks achieved 1 ms average response time, with Quarkus showing slightly higher throughput. Both frameworks maintained 0% error rate across all load levels. In production mode, Quarkus consumed only 17 MB of heap memory compared to Spring Boot's 20 MB. In Docker mode, Quarkus started in 4.275 seconds compared to Spring Boot's 11.234 seconds, demonstrating a significant startup time advantage. Results indicate that framework selection should be driven by workload characteristics and deployment requirements.

**Index Terms**—Spring Boot, Quarkus, REST API, Performance, Java, Microservices, Throughput, JMeter, VisualVM, Docker, Concurrency

## I. INTRODUCTION

Java backend development has grown significantly over the past decade, with Spring Boot becoming the dominant framework for building REST APIs. Recently, Quarkus emerged as a modern alternative designed for cloud-native and microservices environments. Quarkus claims to offer faster startup times and lower memory consumption compared to Spring Boot, which raises an important question: does Quarkus actually perform better under real load conditions and containerized deployments?

To answer this question, identical REST APIs were designed and implemented in both frameworks, and their performance was measured using industry-standard tools. This study evaluates both frameworks under multiple concurrency levels, database-connected endpoints, and Docker containerized deployments, which reflect real-world microservices behavior more accurately. Apache JMeter was used for load simulation, VisualVM for heap memory monitoring, and Docker for containerized startup time benchmarking. The goal is to provide practical, data-driven guidance for developers choosing between these two frameworks.

## II. LITERATURE REVIEW

Several researchers have examined Java framework performance from different angles. Pereira et al. [1] analyzed energy efficiency across Java collection frameworks and found that framework-level design choices significantly impact resource consumption. Seo et al. [2] studied energy behavior in pervasive Java systems, noting that runtime overhead varies considerably depending on execution context. Taibi et al. [5] conducted a survey on microservices adoption in practice and identified performance and resource efficiency as primary selection criteria for backend frameworks. Dragoni et al. [6] provided a comprehensive overview of microservices architecture evolution, highlighting the growing importance of lightweight and fast-starting frameworks in cloud-native deployments.

While these works establish that framework internals affect performance, they do not directly compare Spring Boot and Quarkus under concurrent HTTP load or containerized deployments. Other online benchmarks suggest Quarkus has advantages in startup time and native compilation, but few studies examine standard JVM mode database performance under varying concurrency levels with Docker deployment comparison. This gap in the literature motivated the present study, which focuses on measurable runtime behavior under controlled, scalable, and containerized load conditions.

### III. METHODOLOGY

Two REST APIs were developed from scratch, one using Spring Boot version 3.2.0 and another using Quarkus version 3.9.1. Both APIs were implemented in Java 17 and exposed two endpoints: a simple /hello GET endpoint returning a text message, and a /users endpoint supporting both GET and POST operations connected to an H2 in-memory database. Spring Boot used Spring Data JPA for database access, while Quarkus used Hibernate ORM with Panache [4], [3].

Both applications were deployed on the same Windows machine to eliminate network-related variables. Spring Boot ran on port 8080 and Quarkus ran on port 8081. Quarkus was packaged in production mode using mvn package and executed with java -jar. Both frameworks were also containerized using Docker with an eclipse-temurin:17-jdk-alpine base image to measure startup time under container deployment.

For the /hello endpoint, Apache JMeter 5.6.3 was configured with 100 concurrent users, 10 second ramp-up, and 10 iterations. For the multi-level concurrency analysis on the /users POST endpoint, four separate test runs were conducted:

- 10 threads, ramp-up 5s, loop 10
- 50 threads, ramp-up 10s, loop 10
- 100 threads, ramp-up 20s, loop 10
- 200 threads, ramp-up 30s, loop 10

An HTTP Header Manager was configured with Content-Type: application/json for all POST requests. Heap memory was measured using VisualVM during idle state after startup for both frameworks in production mode. Docker startup time was measured from container launch to the first ready log message.

### IV. RESULTS

Table I presents the performance results for the /hello endpoint. Table II presents the multi-level concurrency results for the /users POST endpoint. Table III presents the startup time comparison. Figures 1, 2, and 3 visualize the results.

**TABLE I - PERFORMANCE COMPARISON: /HELLO ENDPOINT**

Metric	Spring Boot	Quarkus
Avg Response Time (ms)	1	1
Avg Throughput (req/sec)	8.7	9.6
Heap Memory Usage (MB)	20	17
Min Response Time (ms)	0	0
Max Response Time (ms)	213	213

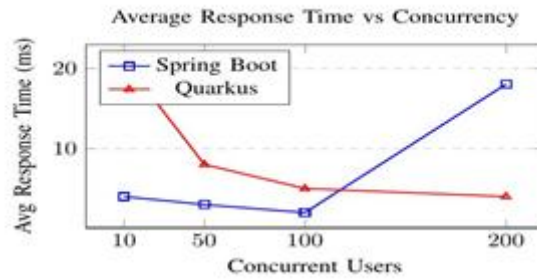


Fig. 1. Response Time under increasing concurrency

TABLE II  
MULTI-LEVEL CONCURRENCY: /USERS POST ENDPOINT

Threads	SB Avg (ms)	Q Avg (ms)	SB TP	Q TP
10	4	21	22.0	23.0
50	3	8	14.1	9.4
100	2	5	17.3	14.0
200	18	4	66.7	19.4

SB = Spring Boot, Q = Quarkus, TP = Throughput (req/sec), Error rate = 0.00% for all tests

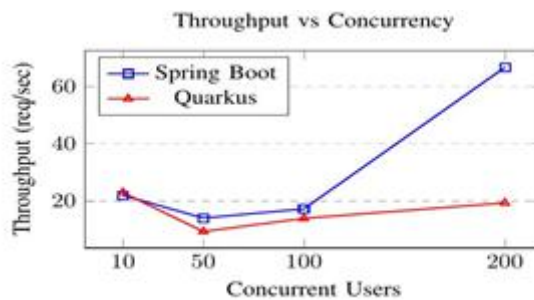


Fig. 2. Throughput under increasing concurrency

For the /hello endpoint, both frameworks achieved identical average response times of 1 ms. Quarkus showed slightly higher throughput at 9.6 req/sec compared to Spring Boot at 8.7 req/sec. In production mode, Quarkus consumed only 17 MB of heap memory compared to Spring Boot's 20 MB.

For the multi-level concurrency test on the /users POST endpoint, both frameworks maintained 0% error rate across all load levels. At lower concurrency (10 threads), Spring Boot achieved faster response time at 4 ms compared to Quarkus at 21 ms. As concurrency increased to 200 threads, Quarkus demonstrated better response time at 4 ms versus Spring Boot's 18 ms, indicating that Quarkus handles high concurrency more efficiently in terms of latency. Spring Boot achieved significantly higher throughput at 200 threads (66.7 req/sec) compared to Quarkus (19.4 req/sec).

For startup time, both frameworks performed similarly in JVM mode at approximately 3 seconds. However, under Docker containerized deployment, Quarkus started in 4.275 seconds compared to Spring Boot's 11.234 seconds,

demonstrating that Quarkus is approximately 2.6 times faster to start in a containerized environment. Figure 3 clearly illustrates this startup time difference.

TABLE III STARTUP TIME COMPARISON: JVM VS DOCKER

Mode	Spring Boot	Quarkus
JVM Mode (seconds)	3.4	3.162
Docker Mode (seconds)	11.234	4.275

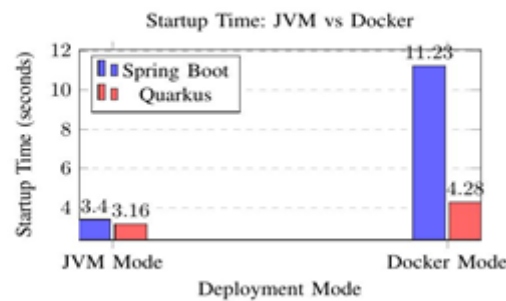


Fig. 3. Startup Time comparison: JVM vs Docker mode

## V. THREATS TO VALIDITY

Several potential threats to the validity of this study must be acknowledged. First, all experiments were conducted on a single Windows machine, which may not reflect performance characteristics observed in production server environments or cloud infrastructure. Hardware-specific factors such as CPU cache behavior, memory bandwidth, and operating system scheduling may influence results.

Second, both APIs used an H2 in-memory database rather than an external production database such as PostgreSQL or MySQL. Real-world applications typically interact with network-attached databases, which introduce additional latency and connection overhead not captured in this study.

Third, JVM-based applications are subject to warmup effects, where the Just-In-Time compiler optimizes frequently executed code paths over time. Early test iterations may exhibit higher latency than steady-state performance. Although multiple test runs were conducted, JVM warmup behavior may still influence results.

Fourth, Docker startup time measurements represent a single observation per framework. Variability due to system load, Docker layer caching, and container initialization overhead may affect reproducibility. Future studies should repeat startup measurements multiple times and report mean and standard deviation.

## VI. CONCLUSION

This study compared Spring Boot and Quarkus REST APIs across multiple concurrency levels, endpoint types, and deployment modes. For lightweight endpoints, both frameworks performed equally in response time, with Quarkus showing a slight throughput advantage. In production mode, Quarkus demonstrated lower heap memory consumption at 17 MB compared to Spring Boot's 20 MB.

Under increasing concurrency, Quarkus maintained lower response times at higher load levels, while Spring Boot achieved significantly higher throughput at 200 concurrent users. In Docker containerized deployment, Quarkus

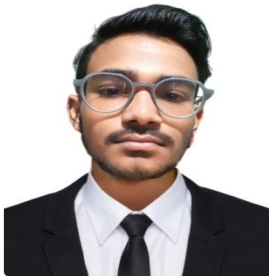
demonstrated a significant startup time advantage, starting approximately 2.6 times faster than Spring Boot (4.275s vs 11.234s). Both frameworks maintained 0% error rate across all test configurations.

These findings confirm that Quarkus is well-suited for cloud-native and microservices deployments where fast container startup, low memory footprint, and latency efficiency under high concurrency are priorities. Spring Boot remains a strong choice for applications requiring high raw throughput. Future work should include testing Quarkus in native compilation mode using GraalVM, comparing CPU utilization under load, and evaluating performance with external production databases such as PostgreSQL.

## REFERENCES

- [1] R. Pereira et al., "The Influence of the Java Collection Framework on Overall Energy Consumption," IEEE/ACM 5th International Workshop on Green and Sustainable Software, 2016.
- [2] C. Seo et al., "Estimating the Energy Consumption in Pervasive Java-Based Systems," IEEE PerCom, 2008.
- [3] E. Deandrea and M. Mandrone, "Quarkus Cookbook: Kubernetes-Optimized Java Solutions," O'Reilly Media, 2020.
- [4] C. Walls, "Spring Boot in Action," Manning Publications, 2022.
- [5] D. Taibi et al., "Microservices in Practice: A Survey Study," IEEE Access, vol. 7, 2019.
- [6] N. Dragoni et al., "Microservices: Yesterday, Today, and Tomorrow," Present and Ulterior Software Engineering, Springer, 2017.

## BIOGRAPHIES



### **Arpan Chakraborty**

is a 2nd year B.Tech student in the Department of Computer Science and Engineering. His research interests include Java backend development, microservices architecture, and cloud-native technologies. He is passionate about performance benchmarking of modern Java frameworks.