African Journal of Advanced Pure and Applied Sciences (AJAPAS)
Online ISSN: 2957-644X
Volume 4, Issue 3, 2025
Page No: 226-233
Website: https://aaasjournals.com/index.php/ajapas/index
ISI 2025: 1.126 SJIFactor 2024: 6.752 معامل التأثير العربي: 1.62

Performance Evaluation of Spring Boot Framework targeting pertinence for high demanding systems

Adnan Elsherif [1 *], Ibrahim Ramadan [2], Ali Aburas [3]
[1,2,3] Computer Science Department, Faculty of Science, University of Tripoli, Tripoli, Libya

تقييم أداء *Spring Boot Framework* الذي يستهدف الملاءمة للأنظمة عالية الطلب

عدنان محمود الشريف[1*]، ابراهيم محمد بن رمضان[2]، علي ابوراس[3]
[3،2،1] قسم الحاسب الآلي، كلية العلوم، جامعة طرابلس، طرابلس، ليبيا

*Corresponding author: adnan.sherif@uot.edu.ly

Received: June 15, 2025	Accepted: August 07, 2025	Published: August 17, 2025

Abstract:
The proliferation of internet connected devices, such as smartphones, tablets, and laptops, has driven demand for distributed applications, necessitating scalable and cost-effective solutions. Cloud computing has emerged as a critical enabler for organizations to deliver services remotely, prompting the development of specialized frameworks to support distributed architectures. However, such frameworks may introduce performance overhead, particularly for high-demand services. This study evaluates the Spring Boot framework, a Java-based API development tool, by implementing a foreign exchange (Forex) server capable of processing currency buy/sell transactions online. The system leverages key framework features, including user authentication and synchronous/asynchronous messaging. To assess performance, the server is profiled using JProfiler, with latency measured by instrumenting framework-related components with time triggers. Stress testing via JMeter includes two experimental scenarios: simulating scalability under increasing user load (1,000 to 10,000 users with 500,000 total requests), and varying request intensity under fixed user load (1,000 users with 100 to 900 requests per user). Findings quantify Spring Boot's performance overhead and its impact on throughput and latency. We discuss implications for adopting the framework in high-demand distributed systems, offering actionable conclusions for developers and architects.

Keywords: Spring Boot, Performance Testing, High-demanding Systems, Java , Distributed Systems.

الملخص
أدى الانتشار الواسع للأجهزة المتصلة بالإنترنت، مثل الهواتف الذكية والأجهزة اللوحية وأجهزة الحاسوب المحمولة، إلى زيادة الطلب على التطبيقات الموزعة، مما استلزم حلولاً قابلة للتوسع وفعالة من حيث التكلفة، وقد برزت الحوسبة السحابية كممكن أساسي للمنظمات لتقديم الخدمات عن بُعد، مما حفز تطوير إطارات عمل متخصصة لدعم البُنى الموزعة، ومع ذلك، قد تُدخل إطارات العمل عبئًا على الأداء، لا سيما في الخدمات ذات معدل عالي من الطلبات، تهدف هذه الدراسة إلى تقييم إطار عمل Spring Boot، وهو أداة لتطوير واجهات برمجة التطبيقات تعتمد على لغة Java، من خلال تنفيذ خادم لتداول العملات الأجنبية (Forex) قادر على معالجة معاملات الشراء/البيع للعملات عبر الإنترنت. يستفيد النظام من ميزات الإطار الأساسية، بما في ذلك المصادقة على المستخدم والتراسل المتزامن وغير المتزامن. لتقييم الأداء، تم استخدام أداة JProfiler لتحليل الخادم، حيث تم تحديد مكونات الإطار ذات الصلة وتم تجهيزها بمحفزات زمنية لقياس التأخر الزمني. تم إجراء اختبارات التحمل باستخدام JMeter من خلال سيناريوين تجريبيين: الأول لاختبار قابلية التوسع تحت ضغط عدد المستخدمين (من 1,000 إلى 10,000 مستخدم مع عدد طلبات ثابت قدره 500,000)، والثاني لاختبار كثافة الطلبات عند عدد مستخدمين ثابت (1,000 مستخدم مع زيادة عدد الطلبات لكل مستخدم من 100 إلى 900). توضح النتائج مقدار العبء الذي يفرضه إطار العمل وتأثيره على الإنتاجية وزمن التأخير. وتناقش الدراسة انعكاسات اعتماد Spring Boot في الأنظمة الموزعة ذات الطلب العالي، مقدمةً استنتاجات عملية للمطورين والمعماريين.

الكلمات المفتاحية: Spring Boot، اختبار الاداء، أنظمة ذات الطلب العالي ، جافا ، الأنظمة الموزعة .

Introduction

Modern software systems are increasingly distributed, driven by declining communication costs, ubiquitous internet access, and the proliferation of cloud computing. Cloud-based architectures enable service delivery through lightweight clients (e.g., web browsers), while the integration of computing hardware into everyday devices such as smartphones, IoT sensors, and smart home systems, has further amplified demand for remote data access and control.

Application Programming Interfaces (APIs) serve as the backbone of these distributed ecosystems, standardizing interactions between heterogeneous systems without exposing internal implementations [1]. Among tools for API development, **Spring Boot** has emerged as a prominent open-source Java framework, extending the Spring ecosystem with features like **auto-configuration, embedded servers, JPA integration, and prebuilt security modules** [2]. These capabilities accelerate development but enforce architectural constraints that may not align with all use cases. Crucially, the abstraction layers and built-in functionalities (e.g., authentication, session management) can introduce **latency overhead**, raising concerns about suitability for high-throughput applications.

Performance testing evaluates how a system performs under various conditions, ensuring it meets speed, stability, and scalability requirements. Within performance testing, **load testing** simulates expected user traffic to measure response times and resource utilization, helping teams optimize system efficiency. **Stress testing**, on the other hand, pushes the system beyond its normal operational limits to identify breaking points and assess recovery mechanisms. These tests are essential for ensuring applications remain reliable under real-world conditions, preventing failures during peak usage, and optimizing infrastructure for scalability. [3]

This work investigates the **performance impact of Spring Boot** in high-demand internet services, addressing the core question: *Does the framework's convenience justify its computational cost under scalable loads?* We evaluate overhead through empirical testing of a Forex transaction server, measuring latency and throughput under stress conditions.

The remainder of this paper is organized as follows: The next section reviews related work on Spring Boot, while subsequent sections detail our methodology, results, and conclusions.

Related Work

Several studies have explored the use of Spring Boot in distributed systems, though few rigorously evaluate its performance overhead. Below, we summarize relevant works and identify gaps this study addresses.

Yi *et al.* [4]designed a music streaming API using Spring Boot (backend) and Vue.js (frontend), enabling users to play and share music. While their architecture demonstrated portability and efficiency through a decoupled frontend-backend design, no performance analysis was conducted, leaving scalability unverified. Similarly, In [5], Zhao et al. developed an office automation system with Spring Boot, highlighting its stability but omitting empirical performance metrics.

Huang et al. [6] introduced RESPECTOR, a tool for generating OpenAPI specifications for Java-based REST APIs (particularly Spring Boot and Jersey). Their results showed that RESPECTOR outperformed existing tools in detecting missing or conflicting specifications in developer-written APIs. However, this work focused on API documentation accuracy rather than runtime performance.

Studies [7] and [8] leveraged Spring Boot's modularity for enterprise systems: He et al. [7] implemented a personnel management system with real-time monitoring, logging user operations and errors for debugging. Li et al. [8] designed a grassroots appraisal system with failure-recovery mechanisms (e.g., automated backups). Both works demonstrated Spring Boot's scalability under feature expansion and user growth but provided no quantitative performance data.

Dhalla [9] conducted a benchmarking study of Spring Boot versus MS.NET Core, measuring response times and error rates under load using JMeter. Their results favored MS.NET Core in throughput and resource efficiency, though both frameworks degraded under extreme loads. However, this comparison did not isolate Spring Boot's intrinsic overhead a gap our work addresses.

Research Gap & Our Contribution

Prior studies primarily focus on Spring Boot's functionality and scalability but neglect systematic evaluation of its performance overhead. Unlike [9], which compared frameworks, our research isolates Spring Boot's impact by profiling its components such as authentication and messaging under load. We quantify the overhead using fine-grained latency measurements obtained through JProfiler, combined with stress testing using JMeter to simulate increasing users and requests. This approach enables us to assess the framework's suitability for high-demand services, a critical consideration often overlooked in existing literature. Our contribution supports the

development of distributed systems by revealing Spring Boot's behavior under stress and offering practical guidance to solution architects aiming to prevent deployment-time bottlenecks.

Case Study: Forex API Implementation

To evaluate the performance characteristics of the Spring Boot framework, we developed a Foreign Exchange (Forex) API that simulates real-world transactional operations within a controlled environment. This implementation serves dual purposes: it allows for structural analysis by examining Spring Boot's internal call-and-response mechanisms, and it supports performance testing through measuring the framework's overhead under controlled load conditions.

The API exposes four core endpoints (illustrated in *Figure 1*):

- **Register:** This endpoint handles the creation of new user accounts within the system. It requires input such as a valid and unique email address, a password, the user's base currency, and their date of birth. The system validates that the email is not blank and follows proper formatting, passwords are meeting the password complexity criteria.
  Upon receiving the request, the system verifies the uniqueness of the email, validates all fields, and, if successful, persists the user in the database.
- **Login:** The Login endpoint verifies a user's email and password to authenticate access. Upon successful authentication, it generates a **JWT (JSON Web Token)**, which contains user-specific information like roles and expiration. This token must be included in the Authorization header of future requests to access protected resources such as the buy and sell endpoint.
- **Buy:** The Buy endpoint allows authenticated users to initiate a purchase operation. A valid JWT token must be included in the request header to authorize the action. The user specifies the currency pair, along with the amount to be exchanged. The system retrieves current rates, verifies sufficient balance, and processes the transaction accordingly**.**
- **Sell:** This endpoint enables authenticated users to exchange currencies by selling a specified amount. A valid JWT token must be included in the Authorization header. The system verifies user balance, fetches current exchange rates, executes the transaction, updates balances, and returns a confirmation. If validation fails, such as due to insufficient funds or expired credentials, an error message is returned.

Spring boot provides a security and authentication layer, to use this authentication layer the following flow of events are required when using the API:

1. Users obtain a JWT token by logging in.
2. The token is included in subsequent requests to restricted endpoints that need authentication. (e.g., /buy, /sell).
3. Spring Boot's security module validates tokens and authorizes requests. If the validation fails an error 401 (unauthorized user) is sent back to the caller.

Figure 1. Forex API Structure.

Design Rationale

The architecture of the Forex API reflects key considerations aimed at facilitating in-depth performance analysis of the Spring Boot framework. It integrates JWT to assess the security overhead introduced by the framework's built-in authentication mechanisms. The implementation also simulates a high-frequency transactional workload through buy and sell operations, mirroring real-world financial activity to impose realistic performance stress. Moreover, the system's modular design isolates specific Spring Boot features, such as dependency injection and auto-configuration, allowing for focused profiling and precise attribution of any observed overhead.

Material and methods

To evaluate Spring Boot's performance in a realistic setting, we began by identifying critical methods along the request-handling path using call tree analysis. This allowed us to pinpoint performance-sensitive components and establish a baseline workflow. Instrumentation was then applied selectively to these methods, enabling precise measurement of runtime behavior under simulated load. The following subsections detail the techniques and tools used for method tracing, metric collection, test data generation, and load simulation.

1. Call Path Analysis with JProfiler

To understand Spring Boot's internal request-handling mechanism, we first used **JProfiler** to trace the call path of requests to the sell endpoint. This analysis identified the sequence of critical classes involved in processing a request (*Figure 2*):

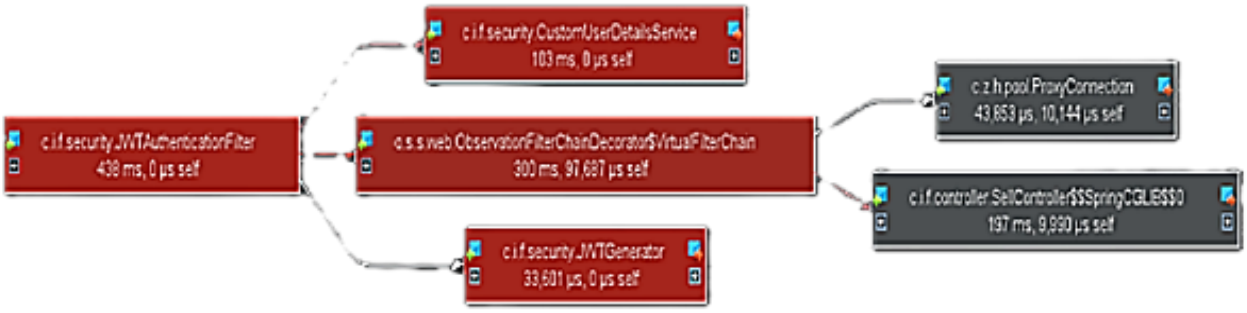

Figure 2. Resulting call tree after sending a request to the sell endpoint.

1. **JWTAuthenticationFilter:** A custom Spring Security filter that intercepts all HTTP requests to validate JWT tokens.
2. **CustomUserDetailsService:** Implements Spring Security's UserDetailsService to fetch user credentials during authentication.
3. **VirtualFilterChain:** Manages the ordered execution of security filters via Spring's FilterChainProxy.
4. **JWTGenerator:** Handles JWT creation, including signing algorithms and token expiration.
5. **SellController:** Processes authenticated sell requests; inaccessible without prior JWT validation.

This step was performed once to establish a baseline understanding of the framework's workflow.

2. Performance Instrumentation with Micrometer

To measure execution times of methods along the identified call path, we integrated Micrometer, a low-overhead metrics library for Java [10]. The implementation leverages Micrometer Timer class to capture method durations, with results stored in thread-safe arrays to prevent concurrency-related anomalies. These metrics are then flushed to a file through a dedicated /write-results endpoint, ensuring efficient collection and analysis of runtime behavior.

3. Test Data Preparation

To simulate realistic usage scenarios, we generated 10,000 unique user profiles using a custom Python script. Each of these user profiles was then issued a corresponding JWT token through separate script, allowing for authentication and authorization processes to be accurately tested within the system.

4. Load Testing with JMeter

To simulate realistic load conditions, we employed JMeter [11] across two key scenarios. Ten test iterations were conducted for each scenario, not as isolated trials, but as a deliberate strategy to progressively push the system toward its performance breaking point. The environment was reset between tests to ensure consistency and to isolate the effect of each load increase.

Scenario 1: Concurrent User Scaling
This scenario comprised ten tests, each scaling the number of concurrent users from 1,000 to 10,000 in 1,000-user increments. Every test generated approximately 500,000 total requests. Users issued requests at 1–2 second intervals to reflect real-world usage patterns, and a 10-second ramp-up period was employed to gradually reach the target load.

Scenario 2: Request Volume Scaling
In this scenario, nine tests were conducted with a constant 1,000 users, while the total request volume was scaled from 100,000 to 900,000 in 100,000-step increments. Requests were issued without delay to maximize server stress and identify throughput bottlenecks under increasing load.

Results and discussion
Our performance evaluation of the Spring Boot framework yielded several key insights across different load testing scenarios. The results demonstrate how various factors impact system behavior and where potential bottlenecks may emerge.

1. Scaling User Concurrency with Fixed Request Volume
In our initial tests, we maintained a constant load of 500,000 requests while progressively increasing concurrent users from 1,000 to 10,000. The data reveals the following:

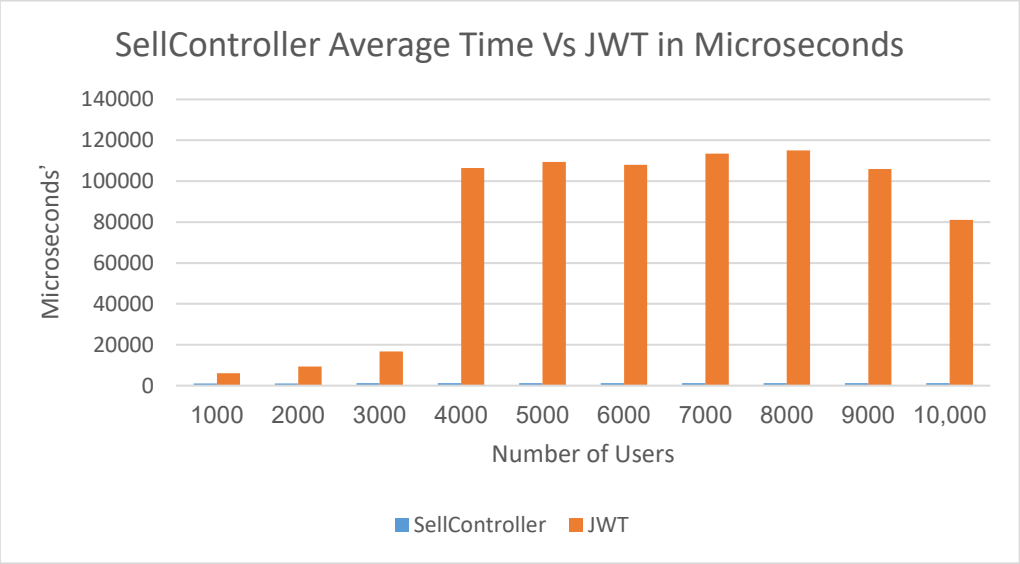

Figure 3. JWT Average Time Vs SellController Average Time.

- **Controller Stability**: As shown in Figure 3, the SellController maintained consistent average response times (1193.8µs ± 74.9µs) across all user levels. This stability stems from Spring Boot's singleton controller pattern [12], where a single instance efficiently handles all incoming requests regardless of user count.

- **Minimal Framework Overhead**: The constant performance profile indicates that Spring Boot's request routing and processing introduces negligible overhead, even at maximum concurrency. This suggests the framework's thread pooling and request handling scale mechanisms effectively within our tested range. For each test conducted (1000 users, 2000 users etc.) the average response time is stable, for example at 4000 users the average response time was (106482.00µs ± 37791.07µs). However we observe an increase in response time from 3000 users test to 4000 user test which will be explored in the next section.

2. **Request Rate Correlation Analysis**

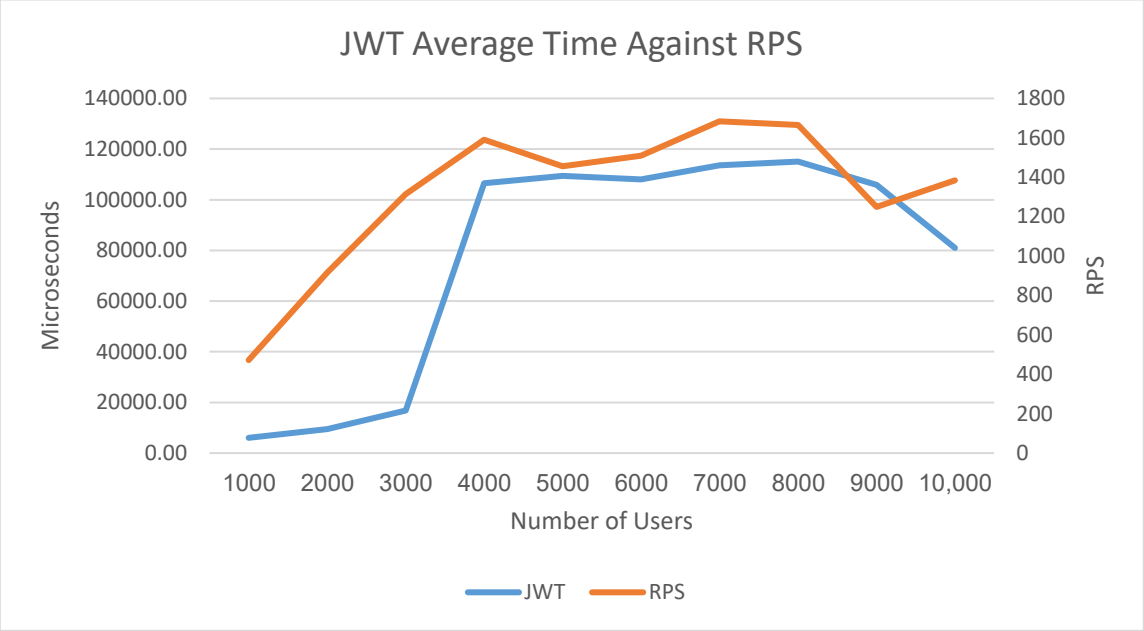

Figure 4. JWTAuthenticationFilter Average Time Against Average RPS (First Scenario).

We observe a jump in overall performance of the framework from 3000 to 4000 users and a slight decrease in performance after 8000 users, as shown in Figure 3. This finding aligns with prior observations from [8], however, no further analysis investigation was conducted to justify this increase in response time.
To explore this behavior, we seek a relation between the increase in response time and the increase in Requests per second (RPS). Figure 4 demonstrates a strong relationship between system performance and RPS. By exploring further this relationship we found a correlation factor of 0.85 between the response time of JWT and RPS, this indicates a strong positive correlation.

We can summarize the observations of the first scenario in the following factors:

A. We observed a significant performance degradation at 4,000 users (increase of 536.5% in response time) followed by partial recovery at 9,000 users. These fluctuations directly correlate with RPS measurements, indicating request rate is the primary performance determinant rather than absolute user count.
B. The near-perfect alignment between JWT authentication times and RPS suggests the security layer may represent a potential bottleneck during request bursts.

3. **Scaling Request Volume with Fixed Users**

Our second test scenario maintained 1,000 concurrent users while progressively increasing total requests from 100,000 to 900,000. The results (Figure 5) show:

A. Despite the 9x increase in total requests, system performance remained stable (112431.30µs ± 1053.3µs) throughout testing. This consistency directly results from maintaining a steady RPS throughout the test. This reinforces our finding of correlation between response time of JWT and RPS detailed in the previous section.
B. The data confirms that total request count has a minimal impact on performance when request rate remains constant. This suggests Spring Boot's architecture can reliably handle large request volumes provided the arrival rate stays within system capacity.

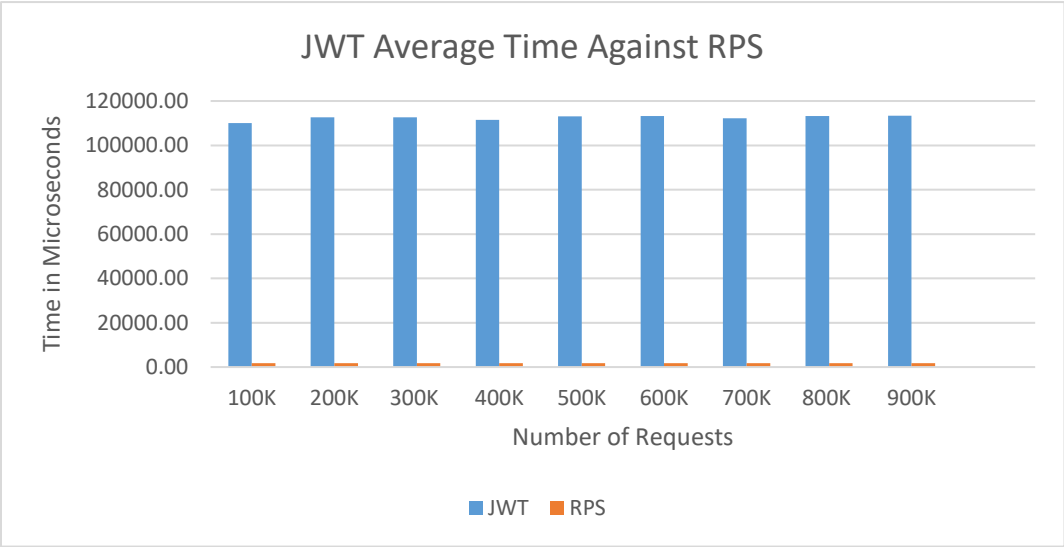

Figure 5. JWTAuthenticationFilter Average Time Against Average RPS (Second Scenario).

Conclusion

To evaluate Spring Boot's readiness for high-demand API scenarios, this study focused on two critical dimensions: performance stability under escalating traffic and architectural overhead during sustained concurrency.

The evaluation of a Forex trading API revealed that Spring Boot exhibits strong performance consistency under high-demand conditions. Across a request range of 100,000 to 900,000, the system sustained average response times around 112,431.30µs, with only a 2.9% degradation observed during burst traffic scenarios. These results suggest that Spring Boot maintains linear scalability and stable throughput, making it a robust choice for high-volume applications.

Architectural overhead introduced by Spring Boot was found to be negligible. Singleton controllers and integrated authentication mechanisms performed predictably, even at peak load, while system resource consumption remained efficient. These findings underscore the framework's suitability for stateless service architectures and reinforce its viability for rapid development of production-grade APIs operating within a concurrency ceiling of 10,000 users per server.

Practical Implications

For system architects evaluating the use of Spring Boot, our findings suggest that the framework's performance characteristics justify its adoption for most enterprise API implementations. However, as concurrent user loads approach or exceed 10,000, horizontal scaling becomes essential to maintain responsiveness and throughput. Additionally, our results indicate that monitoring request rates offers more actionable insight into system performance than simply tracking absolute user counts.

Future Research Directions

We propose extending this work by deploying the test environment on a commercial cloud infrastructure and comparing results against our local network benchmarks. Furthermore, evaluating real-world factors such as: variable bandwidth allocation, network latency fluctuations and connection stability issues

Our methodology establishes a foundation for comparative framework performance analysis. For example, we believe that immediate application to Python/Django ecosystems is possible and with some adaptation may be required for frameworks such as Node.js (Express/NestJS), .NET Core and Go framework. Moreover, the metrics serve as a standardization for a comparison between different frameworks.

This work provides both immediate practical guidance for Spring Boot implementations and a methodological foundation for broader framework evaluation in API development contexts.

References

[1] M. Goodwin, "What is an API?," IBM, 9 April 2024. [Online]. Available: https://www.ibm.com/think/topics/api. [Accessed 9 May 2025].

[2] IBM, "What is Java Spring Boot?," IBM. [Online]. [Accessed 9 May 2025].

[3] BrowserStack, "Load Testing vs Stress Testing vs Performance Testing," BrowserStack, 29 April 2025. [Online]. Available: https://www.browserstack.com/guide/load-testing-vs-stress-testing-vs-performance-testing. [Accessed 17 May 2025].

[4] Y. Yi and Z. Li, "Design and Implementation of Music Web Application based on Vue and Spring Boot," in *Proceedings of the 2020 4th International Conference on Electronic Information Technology and Computer Engineering*, Xiamen, China, 2021.

[5] P. Zhao, D. Zhang and S. Lei, "Design and Implementation of Office Automation Management System Based on Spring Boot," *EDCS '24: Proceedings of the 2024 Guangdong-Hong Kong-Macao Greater Bay Area International Conference on Education Digitalization and Computer Science*, pp. 228-233, 2024.

[6] R. Huang, M. Motwani, I. Martinez and A. Orso, "Generating REST API Specifications through Static Analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Lisbon, Portugal, 2024.

[7] X. He, D. Zhang, S. Lei, Z. Li and P. Zhao, "Design and Implementation of Personnel Management System for Colleges and Universities Based on Spring Boot Framework," in *Proceedings of the 2024 International Conference on Intelligent Education and Computer Technology*, Guilin, China, 2024.

[8] Z. Li, D. Zhang and X. He, "Design and implementation of a grassroots appraisal system based on Spring Boot framework," in *CISAI '24: Proceedings of the 2024 7th International Conference on Computer Information Science and Artificial Intelligence*, Shaoxing China, 2024.

[9] H. K. Dhalla, "A Performance Comparison of RESTful Applications Implemented in Spring Boot Java and MS.NET Core," *Journal of Physics: Conference Series*, vol. 1933, no. 1, p. 012041, jun 2021.

[10] Micrometer, "Concepts," Micrometer, [Online]. Available: https://docs.micrometer.io/micrometer/reference/concepts.html. [Accessed 10 May 2025].

[11] Apache Jmeter, "Apache JMeter™," The Apache Software Foundation, [Online]. Available: https://jmeter.apache.org/. [Accessed 10 May 2025].

[12] B. Haick, "Singleton Design Pattern in Java and Spring Boot: Understand and See it in Practice!," Linkedin, 9 Jan 2025. [Online]. Available: https://www.linkedin.com/pulse/singleton-design-pattern-java-spring-boot-understand-see-bruno-haick-z2fhf. [Accessed 26 Jan 2025].