Performance Benefits of Reactive Frameworks

Ramesh V.
AT&T
3400 W Plano Pkwy, Plano, TX 75075

ABSTRACT
Reactive programming enables the construction of responsive and robust applications that are capable of efficiently managing asynchronous data streams and dynamic changes. Among various frameworks that support this paradigm, Spring is distinguished by its adaptability and lightweight Java-based structure, making it particularly suitable for enterprise environments. Introduced in 2003, the Spring ecosystem offers two primary web frameworks: Spring Web Model-View-Controller and Spring WebFlux. In the former, the original component of the Spring framework is optimized for the servlet API and container environments, while the latter is a newer addition that leverages a reactive stack architecture to achieve enhanced scalability and performance. This study comprehensively evaluates these frameworks through performance benchmarking across diverse scenarios. By integrating a broad spectrum of performance metrics (i.e., throughput and response time) and real-world applications, this study aims to extend the current literature and provide developers with concrete insights into selecting the appropriate Spring framework for specific enterprise needs based on synchronous and reactive programming models. [1][3]

General Terms
Performance, Algorithms, Design, Experimentation and Measurement

Keywords
Spring WebFlux, Reactive Streams, EventLoop Model, Spring MVC, Web Application Performance, High Concurrency Systems and Netty vs Servlet Container

1. INTRODUCTION
Web applications typically integrate many complex components such as databases and rest API calls that traditionally operate under a synchronous, blocking model. This approach is effective at low data scales but becomes inefficient under increased data load, leading to high latency and poor resource utilization on multicore systems. Reactive programming addresses these shortcomings by introducing a nonblocking, event-driven model, thereby enhancing system responsiveness and resource management. Within the Spring framework, this shift is represented by the contrast between Spring Web Model-View-Controller (MVC), which adheres to traditional synchronous operations, and Spring WebFlux, which employs a reactive programming model optimized for dynamic, real-time interactions among the service, database, and network layers. [2][4]

This study contributes to the ongoing evaluation of reactive- and servlet-based web frameworks by providing practical insights and reproducible Spring framework configurations. The specific goals of this research are to systematically test the performance characteristics of Spring Web MVC and Spring WebFlux under different workloads and analyze the resulting performance metrics to guide real-world architectural decisions. The original contributions of this study include (1) the implementation of realistic benchmarking using two representative scenarios—compute-bound and network-bound APIs—to evaluate the performance of Spring Web MVC and Spring WebFlux; (2) iterative performance-driven code refinement, presenting source codes and configurations developed through repeated benchmarking, with refinements guided by empirical performance outcomes; (3) side-by-side comparison of servlet and reactive stacks, presenting detailed implementation, configuration, and tuning strategies for Spring Web MVC (Tomcat, RestTemplate, and Async Executor) and Spring WebFlux (Netty, WebClient, and reactive schedulers); (4) runtime configuration guidelines through a structured analysis of runtime tuning for thread pools, database connection pools, WebClient, Netty event loops, and back pressure control; (5) performance observations and recommendations concerning when to adopt a reactive or servlet-based model depending on the nature of the workload; and (6) inclusion of code snippets, configurations files, and performance metrics to support reproducibility. [1][5]

In addition, this study builds upon and complements prior research comparing web frameworks and execution models. Prior work in this domain has explored the trade-offs between traditional thread-per-request architectures and modern event-driven approaches. This study contributes to that body of knowledge by incorporating realistic scenarios, performance-tuned configurations, and measurable outcomes. By providing practical insights and reproducible configurations, it offers a grounded perspective for developers and architects making framework-related decisions. The comparison methodology is based on widely accepted practices in performance engineering and contributes to ongoing efforts in evaluate framework suitability in real-world applications.

2. METHOD
2.1 Synchronous vs. Reactive Programming
To comprehensively assess the contrasts and functionalities of synchronous and reactive programming models, exploring how each type manages resources and user requests is essential.

The synchronous programming model, widely utilized in frameworks such as Spring Web MVC, allocates a single thread to comprehensively manage each web request, as illustrated in Figure. 1.

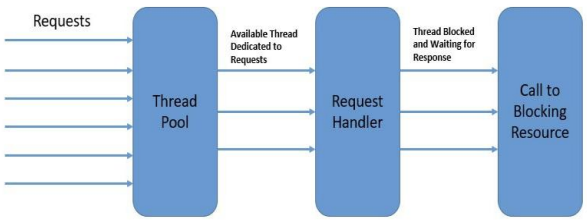

Figure 1. Block diagram of a synchronous programming model.

While the synchronous programming model functions well under minimal data load, it poses problems under high demand

owing to thread idling during blocking operations, such as database queries, inhibiting the ability of threads to perform other tasks. This inefficiency escalates in high-traffic environments, wherein managing numerous threads leads to substantial overhead due to context switching. Additionally, the inability of this model to simultaneously handle multiple requests without adding more threads complicates scalability and reduces responsiveness, particularly in dynamic, interactive applications. These limitations highlight the need for more efficient models, such as the nonblocking, event-driven ones used in reactive programming, which can better meet the demands of modern web applications. [6][8]

Reactive programming fundamentally changes how concurrency is approached by structuring applications around asynchronous data flows and event propagation, thereby improving resource utilization and concurrency, as illustrated in Figure. 2. Unlike traditional thread-based models wherein operations block threads, reactive programming employs nonblocking operations. Operations such as database reads return a publisher to which subscribers can asynchronously react, enabling event processing and generation without tying up threads, thereby enhancing overall system efficiency and responsiveness. [9]

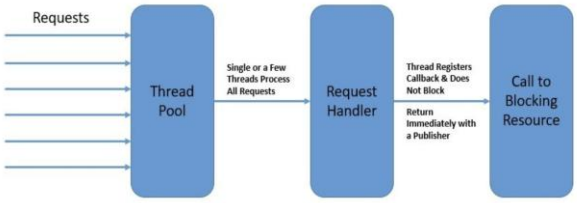

Figure 2. Block diagram of a reactive programming model.

Among several programming models that describe a reactive approach to concurrency, one such reactive asynchronous programming model is the event loop model, as illustrated in Figure. 3. The event loop model is based on the reactor library, which uses a single-threaded event loop to handle all incoming requests. Each new request is assigned to the event loop, which processes the request and returns a response. Notably, the event loop can handle other requests while one request is being processed, improving overall performance and scalability. [7]

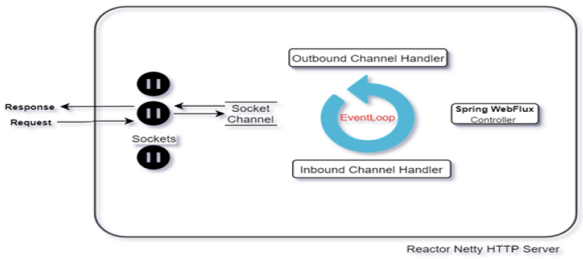

Figure 3. Schematic of the event loop working model.

2.2 Development of Spring Web MVC and Spring WebFlux Application Scenarios

Applications were constructed for the following distinct scenarios to assess how each framework handles specific operational scenarios based on response time and throughput metrics.

1. Compute-Bound Scenarios: These scenarios involve the implementation of a RESTful API that retrieves product entries from an H2 in-memory database and enriches them using in-memory lookup maps based on attributes such as category, region, and type. The response includes a transformed representation of each product, incorporating descriptive metadata through string processing and business rule evaluation.
2. Network-Bound Scenarios: These scenarios test network interaction, where the API makes a POST request to an external microservice with a JSON payload. The operation is characterized by network latency and potential thread blocking due to synchronous input/output (I/O) behavior.

The application logic and configuration parameters were finalized through multiple rounds of performance evaluation using representative workloads. Each iteration informed incremental code and configuration refinements, resulting in improved throughput, latency, and resource efficiency. [16][18]

Tables 1–2 list the design considerations for the compute-bound scenarios in Spring Web MVC.

Table 1. Design considerations for Tomcat thread pool request handling.

Setting	Consideration
max-threads	Set based on CPU cores. I have configured 120 for a 6 cores CPU. (# of cores * 20)
min-spare-threads	Keep a small pool for burst traffic to avoid ramp-up delay
accept-count	Size of the request queue - larger means more tolerance during load spikes before rejecting.
connection-timeout	Prevents idle connections from hanging too long : 5s is typical

Table 2. Design considerations for Hikari CP - database connection pool.

Setting	Consideration
maximum-pool-size	Should match expected peak concurrent DB operations (e.g. 25~30). More isn't better unnecessary DB connections consume memory
minimum-idle	Set a low value like 5~10 to avoid pool ramp-up delays
idle-timeout	Enable GZIP for large payloads.
connection-timeout	Prevent slow clients from hanging.
max-lifetime	Forces connection recyclying to avoid stale connections

Tables 3–4 list the design considerations for the compute-bound scenarios in Spring Webflux.

Table 3 lists the design considerations for the computer-bound scenarios in Spring WebFlux.

Aspect	Spring WebFlux
DB Access	Non-blocking (R2DBC)
Enrichment Logic	Offloaded via Schedulers.boundedElastic()
Concurrency Model	Event-loop + async thread pool

Table 4. Design considerations for Reactor Netty: reactive server configuration.

Setting	Consideration
Event loop threads	Set via loop resources. Use 2 × CPU cores.
Read/write timeout	Protect against stalled connections. 10s is reasonable.
Compression	Enable GZIP for large payloads.
Connection timeout	Prevent slow clients from hanging.

3. RESULTS AND DISCUSSION

This section presents the findings of the performance evaluation conducted on Spring WebMVC and Spring WebFlux under compute-bound and network-bound scenarios. The discussion highlights their behavior across key performance metrics such as response time, throughput, and resource utilization.

3.1 Compute-Bound Scenario

The JMeter Test plan included 1000 users, a ramp-up period of 1s, and a loop count of 10,000. The system configuration comprised a MacBook Pro, 6-core Intel i7, 2.6 GHz system. Spring version 5.2.32 was employed.

3.1.1. API Performance Index and Performance Statistics

As shown in Figure. 4, both applications (Spring Web MVC and Spring WebFlux) successfully responded to all API requests within a response time range of 500 ms to 1.5 s. In addition, as illustrated in Figure. 5, Spring Web MVC consistently outperformed Spring WebFlux, delivering higher throughput and faster response times under similar load conditions. [11][12] [19]

Figure 4. Application performance index of the compute-bound scenario

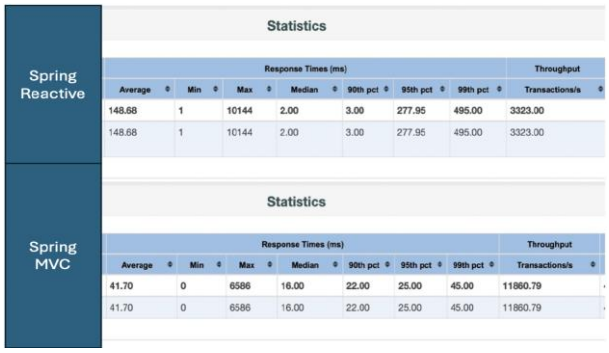

Figure 5. Performance statistics for the compute-bound scenario

3.1.2. Throughput, Average Response Times

The detailed performance metrics presented in Figure. 6 reveal that the throughput of Spring Web MVC was approximately four times greater than that of Spring WebFlux, underscoring its efficiency in request handling. Additionally, Figure. 7 shows that Spring Web MVC achieved a significantly faster average response time, also nearly four times better. This performance advantage is further validated by Figure. 8, where Spring Web MVC maintained superior responsiveness at both the 95th and 99th percentiles, contributing to a more consistent and efficient overall performance compared to Spring WebFlux.

Figure 6. Throughput values for the compute-bound scenario

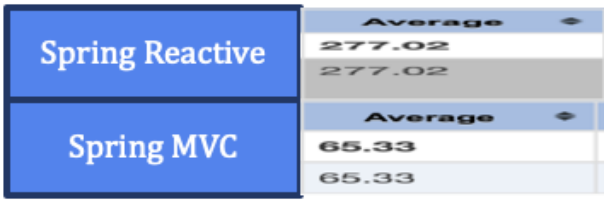

Figure 7. Average response times for the compute-bound scenario

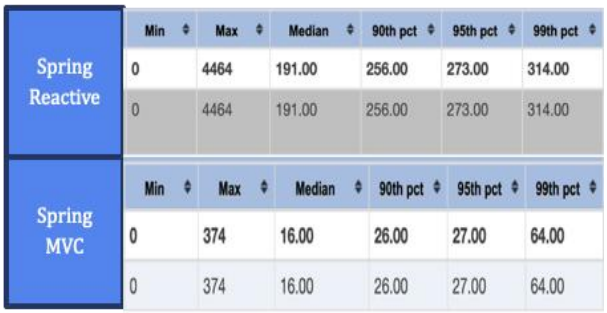

Figure 8. Average response times for compute-bound scenario 2 at the 95th and 99th percentiles.

3.2 Network-Bound Scenario

The JMeter test plan included 25 users, a ramp-up period of 1 second, and a loop count of 10,000. The system configuration comprised a MacBook Pro with a 6-core Intel i7, 2.6 GHz processor. Spring version 5.2.32 was employed.

3.2.1. API Performance Index and Performance Statistics

As shown in Figure. 9, both applications (Spring Web MVC and Spring WebFlux) responded to all API requests within a response time range of 500 ms to 1.5 s. Figure. 10 presents the statistical report, which clearly demonstrates that Spring WebFlux consistently outperformed Spring Web MVC under the given test conditions.

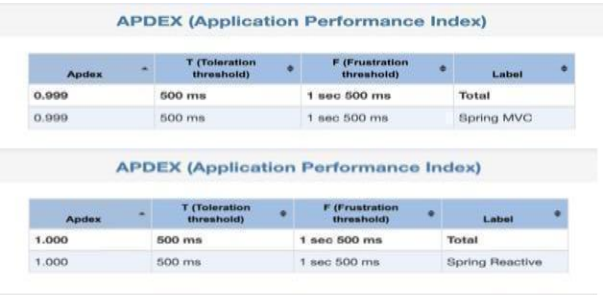

Figure 9. Application performance index for network-bound scenario

Figure 10. Performance statistics for the network-bound scenario

3.2.2. Throughput, Average Response Times

As shown in Figure. 11, Spring WebFlux achieved throughput levels nearly four times higher than Spring Web MVC, reflecting its robust handling capabilities in high-concurrency environments. Additionally, Figure. 12 indicates that the response time for Spring WebFlux was more than four times faster than that of Spring Web MVC. This performance advantage is further supported by the percentile analysis in Figure. 13, which highlights Spring WebFlux's consistent responsiveness at both the 95th and 99th percentiles.

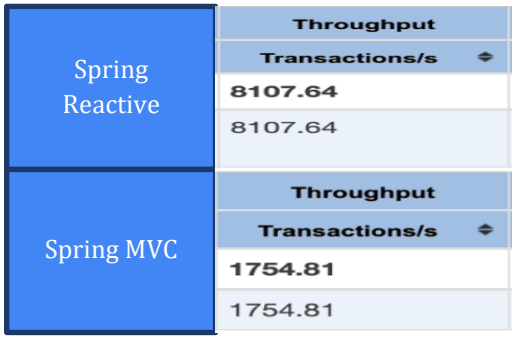

Figure 11. Throughput report for network-bound scenario

Figure 12. Average response times for the network-bound scenario

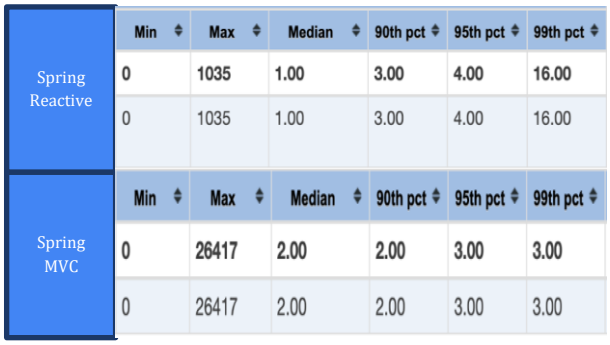

Figure 13. Average response times for network-bound scenario 2 at the 95th and 99th percentiles.

4. CONCLUSION

This study conducted a detailed, hands-on evaluation of two architectural models in the Spring ecosystem: Spring Web MVC and Spring WebFlux. The evaluation was grounded in practical implementation, wherein source code was iteratively developed and refined through repeated performance benchmarking. Spring Web MVC adopts a thread-per-request model in which each incoming request occupies a dedicated thread until the entire operation, including downstream interactions (e.g., database or external API calls), is completed. This synchronous behavior is suitable for applications that involve CPU-bound processing with limited external I/O, where parallelism can be manually introduced using mechanisms such as @Async and ThreadPoolTaskExecutor. Alternatively, Spring WebFlux, built on Reactive Streams, embraces a nonblocking event-driven model. Unlike the servlet stack, it does not assign a separate thread-per-request. Instead, Spring WebFlux utilizes a small number of event loop threads

and reactive operators (Mono and Flux) to efficiently manage I/O, making it better suited for high-concurrency scenarios that involve external service integrations or latency-prone resources.

Two realistic scenarios were implemented to empirically analyze the performance of servlet and reactive stacks, with detailed code snippets provided for both scenarios, including service logic, controller mappings, configuration files, and thread pool tuning. In Spring WebFlux, the enrichment logic was carefully offloaded to Schedulers.boundedElastic() to prevent blocking of the Netty event loop. In Spring Web MVC, similar CPU-bound processing was handled using @Async and a custom TaskExecutor. The applications were subjected to repeated load testing using Apache Jmeter to evaluate throughput, latency, and system resource utilization under varying concurrency levels, revealing the following trends:

1. In the compute-bound scenarios, Spring Web MVC with tuned @Async executors performed better than Spring WebFlux owing to direct thread allocation and reduced I/O latency from the H2 in-memory database.

2. In the network-bound scenarios, Spring WebFlux substantially outperformed Spring Web MVC owing to its nonblocking design, which allowed the system to handle more concurrent requests without increasing the thread pool size or CPU consumption.

3. WebClient configurations, including connection pooling and timeout settings, exerted a tangible impact on performance under high I/O load.

Spring Web MVC holds a slight advantage over Spring WebFlux in environments that do not demand intensive interactions with external systems owing to its ability to efficiently and simultaneously handle numerous threads, offering rapid request processing. However, Spring WebFlux performs better in more complex scenarios that require frequent file system access, database operations, or network interactions by leveraging its event loop model. Spring WebFlux avoids the typical delays found in the thread pool approach of Spring Web MVC, which is particularly evident with increasing system demand. Transitioning to a fully reactive architecture with Spring WebFlux not only maximizes resource utilization but also considerably boosts application performance. The findings of the study indicate that applications can achieve increased scalability and responsiveness by utilizing reactive drivers for databases and employing reactive HTTP client for network requests, making Spring WebFlux the preferable choice for modern, high-performance commercial applications. [14][15][20]

5. ACKNOWLEDGMENTS

I sincerely thank the AT&T research paper reviewers for their insightful feedback, which greatly contributed to improving this work.

6. REFERENCES

[1] O. Dokuka and I. Lozynskyi, *Hands-on Reactive Programming in Spring 5: Build Cloud-Ready, Reactive Systems with Spring 5 and Project Reactor*, Packt, Birmingham, UK, 2018.

[2] R. Sharma, *Hands-On Reactive Programming with Reactor: Build Reactive and Scalable Microservices Using the Reactor Framework*, Packt, Birmingham, UK, 2018.

[3] M. Srivastava, *Mastering Spring Reactive Programming for High-Performance Web Apps*, Notion Press, New York, NY, USA, 2024.

[4] Y. Mednikov, *Friendly WebFlux: A Practical Guide to Reactive Programming with Spring WebFlux*, Independent, 2021.

[5] C. Deinum and I. Cosmina, "Building Reactive Applications with Spring WebFlux," in *Spring in Action*, 5th ed., Manning, New York, NY, USA, 2021, ch. 10.

[6] A. Sukhambekova, "Comparison of Spring WebFlux and Spring MVC," *Modern Scientific Method*, no. 9, Mar. 2025.

[7] Royal Institute of Technology (KTH), "Comparing Virtual Threads and Reactive WebFlux in Spring," M.S. thesis, Stockholm, Sweden, 2023.

[8] A. Nordlund and N. Nordström, "Comparing Virtual Threads and Reactive WebFlux in Spring," *diva-portal.org*, 2023. [Online]. Available: https://www.diva-portal.org/smash/get/diva2%3A1763111/FULLTEXT01.pdf

[9] Q. Li and R. Sharma, "Review on Spring Boot and Spring WebFlux for Reactive Web Development," *ResearchGate*, 2020. [Online]. Available: https://www.researchgate.net/publication/341151097

[10] A. Filichkin, "Spring Boot Performance Battle: Blocking vs Non-Blocking vs Reactive," *Medium*, May 2018. [Online]. Available: https://filia-aleks.medium.com/microservice-performance-battle-spring-mvc-vs-webflux-80d39fd81bf0

[11] M. Piyumal, "Mastering Reactive Programming with Spring WebFlux," *Medium*, Jan. 2024. [Online]. Available: https://manjulapiyumal.medium.com/mastering-reactive-programming-with-spring-webflux-47dbf57857f0

[12] "SpringBoot MVC vs WebFlux: Performance Comparison for JWT Verify and MySQL Query," *Medium*, Jul. 2025. [Online]. Available: https://medium.com/deno-the-complete-reference/springboot-mvc-vs-webflux-performance-comparison-for-jwt-verify-and-mysql-query-10d5ff08a1ba

[13] P. Minkowski, "Performance Comparison Between Spring MVC vs Spring WebFlux with Elasticsearch," *Personal Blog*, Oct. 2019. [Online]. Available: https://piotrminkowski.com/2019/10/30/performance-comparison-between-spring-mvc-and-spring-webflux-with-elasticsearch/

[14] G. Munhoz, "API Performance — Spring MVC vs Spring WebFlux vs Go," *Medium*, Aug. 2020. [Online]. Available: https://filipemunhoz.medium.com/api-performance-spring-mvc-vs-spring-webflux-vs-go-f97b62d2255a

[15] F. Dorado, "Reactive vs Non-Reactive Spring Performance," *Personal Blog*, Jun. 2019. [Online]. Available: https://frandorado.github.io/spring/2019/06/26/spring-reactive-vs-non-reactive-performance.html

[16] T. Emanovikov, "R2DBC vs JDBC vs Vert.x – Not So Fast Benchmark," *Medium*, n.d. [Online]. Available:

https://medium.com/@temanovikov/r2dbc-vs-jdbc-vs-vert-x-not-so-fast-benchmark-c0a9fcabb274

[17] G. Gatheca, "Spring WebFlux: Load Testing Using JMeter," *Medium*, Sep. 2021. [Online]. Available: https://gathecageorge.medium.com/6-spring-webflux-load-testing-using-jmeter-b0875b09fc25

[18] "Performance Testing Strategies for Spring WebFlux Applications," *Moldstud.com*, Jul. 2025. [Online]. Available: https://moldstud.com/articles/p-performance-testing-your-spring-webflux-application-tools-strategies-and-best-practices

[19] "Mastering Spring WebFlux: Reactive APIs at Scale," *Java Code Geeks*, Jul. 2025. [Online]. Available: https://www.javacodegeeks.com/2025/07/mastering-spring-webflux-reactive-apis-at-scale.html

[20] "Spring MVC vs. Spring WebFlux: Choosing the Right Framework for Your Project," *Dev.to*, 2024. [Online]. Available: https://dev.to/jottyjohn/spring-mvc-vs-spring-webflux-choosing-the-right-framework-for-your-project-4cd2