INTERNATIONAL JOURNAL OF COMPUTER SCIENCE & INFORMATION SYSTEM

Volume11 Issue01, January-2026, pg. 121-131

Published Date: - 31-01-2026

E-ISSN: 2536-7919
P-ISSN: 2536-7900

A Priority-Aware Reactive Architecture For SLA-Tiered Financial Apis: Integrating Spring Webflux, Asynchronous Concurrency, And Cloud-Native Design

Adrian Kovacs

Budapest University of Technology and Economics, Hungary

Abstract: The accelerating digitization of financial services has produced an unprecedented demand for application programming interfaces that are not only scalable and fault tolerant but also capable of honoring heterogeneous service-level agreements under volatile and bursty traffic. Contemporary banking, payments, trading, and risk platforms operate in environments where premium clients, regulatory processes, and latency-sensitive trading flows must coexist with mass-market consumer workloads on the same digital infrastructure. This coexistence exposes a fundamental tension in web-service engineering: how to guarantee differentiated quality of service when requests are processed through shared, resource-constrained microservice backends. Reactive programming, particularly as implemented in Spring WebFlux and Project Reactor, has emerged as a promising paradigm to address this tension by enabling non-blocking, event-driven execution and fine-grained flow control. However, reactive frameworks alone do not automatically yield priority-aware behavior, and the absence of explicit service differentiation mechanisms can undermine the very service-level guarantees that financial institutions are required to uphold.

This study develops and analyzes a priority-aware reactive API architecture grounded in Spring WebFlux and informed by recent research on service-level-aware traffic management in financial systems. Central to the analysis is the conceptual and practical framework proposed by Hebbar in the study of priority-aware reactive APIs for SLA-tiered financial traffic, which articulates how reactive streams can be augmented with priority semantics to ensure predictable latency and throughput for premium workloads while maintaining overall system efficiency (Hebbar, 2025). Building on this foundation, the article situates priority-aware reactive design within the broader literature on asynchronous I O, virtual threads, cloud infrastructure, and microservice performance, drawing from both academic and practitioner-oriented sources.

The methodology consists of a structured conceptual modeling of request flows, execution pipelines, and backpressure propagation across a hypothetical but realistic financial microservices platform. The analysis interprets existing empirical and theoretical findings on Spring WebFlux, Spring MVC, and

INTERNATIONAL JOURNAL OF COMPUTER SCIENCE & INFORMATION SYSTEM

Volume11 Issue01, January-2026, pg. 121-131

Published Date: - 31-01-2026

E-ISSN: 2536-7919
P-ISSN: 2536-7900

asynchronous frameworks to infer how priority tiers interact with reactive scheduling, connection pools, and resource contention. Results are presented in descriptive and interpretive form, highlighting how priority tagging, reactive operators, and non-blocking I O jointly contribute to differentiated service outcomes. The discussion critically evaluates the trade-offs, limitations, and future research directions associated with priority-aware reactive architectures, including their interaction with virtual threads, persistent data sources, and cloud-native deployment models.

By integrating service-level awareness directly into the reactive programming model, the article argues that financial service providers can achieve a more principled alignment between business-level commitments and low-level execution semantics. This alignment is shown to be essential for sustaining trust, regulatory compliance, and competitive differentiation in an increasingly API-driven financial ecosystem.

Keywords: Reactive programming, Spring WebFlux, service level agreements, financial microservices, priority scheduling, asynchronous I O.

INTRODUCTION

The evolution of financial services into highly interconnected, software-defined ecosystems has fundamentally altered the technical and organizational requirements placed upon application programming interfaces. Banks, payment processors, and trading platforms no longer operate as monolithic systems serving homogeneous user bases, but rather as layered microservice networks that mediate interactions among retail customers, institutional investors, regulators, and third-party fintech providers. Each of these actors brings distinct expectations regarding latency, throughput, availability, and reliability, which are formalized through service-level agreements and regulatory mandates. The technical challenge that arises from this diversity is how to enforce differentiated quality of service when all requests ultimately traverse the same computational fabric. Traditional synchronous and thread-per-request web architectures have struggled to meet this challenge because their blocking semantics and coarse-grained resource management lead to head-of-line blocking and unpredictable performance under load, as documented in comparative studies of Spring MVC and reactive frameworks (Deinum and Cosmina, 2021; Filichkin, 2018).

Reactive programming, as embodied in Spring WebFlux and the Reactor framework, represents a paradigm shift in this context by replacing blocking calls with non-blocking, event-driven streams that can scale more gracefully under high concurrency (Sharma, 2018; Mednikov, 2021). By modeling computation as a flow of asynchronous signals rather than a sequence of blocking method calls, reactive systems can multiplex thousands of concurrent requests onto a small number of threads, thereby reducing context-switching overhead and improving resource utilization. However, while this architectural style enhances

Volume11 Issue01, January-2026, pg. 121-131

Published Date: - 31-01-2026

E-ISSN: 2536-7919
P-ISSN: 2536-7900

scalability, it does not inherently address the problem of service differentiation. In a purely reactive pipeline, all requests are treated as elements of a stream, and unless explicitly annotated or routed differently, premium and standard workloads compete for the same execution slots and I O channels. This limitation is particularly problematic in financial services, where regulatory reporting, high-frequency trading, and fraud detection often require stricter latency and reliability guarantees than ordinary consumer transactions (Hebbar, 2025).

The literature on asynchronous web frameworks and microservice performance has repeatedly emphasized that non-blocking I O and event loops are necessary but not sufficient for predictable quality of service. Studies comparing Spring WebFlux and Spring MVC have shown that reactive frameworks can outperform blocking ones under high concurrency, yet they can also exhibit performance degradation when backpressure, connection pools, or database interactions are not carefully managed (Sukhambekova, 2025; Catrina, 2023). Similarly, research on asynchronous I O in web applications has demonstrated that scalability gains can be undermined by contention at the persistence layer or by uncoordinated concurrency control (Rankovski and Chorbev, 2016; Wycislik and Ogorek, 2019). These findings suggest that a holistic approach to service differentiation must span the entire request lifecycle, from HTTP ingress through business logic to data storage.

Within this context, Hebbar's work on priority-aware reactive APIs provides a crucial conceptual and practical bridge between business-level service-level agreements and low-level reactive execution (Hebbar, 2025). By proposing mechanisms for tagging requests with priority metadata and propagating these tags through reactive pipelines, Hebbar demonstrates how Spring WebFlux can be extended to enforce SLA tiers in a financial environment. This approach is particularly significant because it aligns with the reactive streams specification embodied in Java Flow and the broader java.util.concurrent ecosystem, which already provides abstractions for asynchronous processing and backpressure (Oracle, 2025a; Oracle, 2025b; Oracle, 2025c). Integrating priority semantics into these abstractions offers a path toward fine-grained control over how resources are allocated among competing workloads.

Despite the promise of priority-aware reactive design, there remains a substantial gap in the literature regarding its theoretical foundations, architectural implications, and interaction with other emerging concurrency models such as virtual threads. Recent work from the Royal Institute of Technology and Nordlund and Nordstrom has compared virtual threads with reactive WebFlux, highlighting trade-offs between simplicity and scalability (KTH, 2023; Nordlund and Nordstrom, 2023). Yet these studies have largely focused on throughput and developer ergonomics rather than on SLA enforcement and service differentiation. Likewise, practitioner reports on performance comparisons among Spring MVC, WebFlux, and other platforms such as Go have emphasized raw metrics without embedding them in a service-level-aware framework (Munhoz, 2020; Minkowski, 2019).

The problem, therefore, is not merely to determine which framework is faster, but to understand how reactive architectures can be designed to honor differentiated service commitments in a complex, cloud-

Volume11 Issue01, January-2026, pg. 121-131

Published Date: - 31-01-2026

E-ISSN: 2536-7919
P-ISSN: 2536-7900

based financial environment. This article addresses this problem by synthesizing insights from reactive programming, asynchronous I O, cloud infrastructure, and priority-aware API design into a coherent analytical framework. By grounding the analysis in Hebbar's priority-aware model and situating it within the broader scholarly debate, the study seeks to articulate both the potential and the limitations of SLA-tiered reactive architectures.

The remainder of the article develops this argument in a comprehensive and theoretically grounded manner. The methodology section explains how a conceptual and literature-driven analytical approach can illuminate the dynamics of priority-aware reactive systems. The results section interprets these dynamics in terms of performance, scalability, and service differentiation, drawing on empirical and comparative studies of Spring WebFlux and related technologies (Li and Sharma, 2020; Rao and Swamy, 2020). The discussion then situates these findings within the wider debates on concurrency models, cloud deployment, and software architecture, identifying areas where further research and innovation are needed.

By advancing a detailed and critical account of priority-aware reactive APIs for financial services, this article contributes to a more nuanced understanding of how modern web frameworks can be aligned with the stringent and heterogeneous demands of the financial domain. In doing so, it responds to a pressing need in both academia and industry for architectures that are not only scalable and resilient but also explicitly attuned to the differentiated value and risk profiles of contemporary digital finance (Hebbar, 2025).

METHODOLOGY

The methodological foundation of this study is grounded in a structured qualitative and analytical synthesis of the extensive body of literature on reactive programming, Spring WebFlux, asynchronous I O, and service-level-aware system design. Given the conceptual and architectural nature of the research problem, an experimental or numerical approach would not adequately capture the multifaceted interactions among framework semantics, concurrency models, and business-level service agreements. Instead, the methodology adopts a design-oriented and interpretive stance, integrating theoretical constructs and empirical findings from prior studies into a coherent analytical model. This approach aligns with established methodologies in software architecture research, where structured design reasoning and comparative analysis are used to infer system properties from documented implementations and performance evaluations (Bijlsma et al., 2017; Fowler, 2003).

The starting point of the methodological framework is Hebbar's articulation of priority-aware reactive APIs in financial services, which provides both a conceptual model and a set of practical design patterns for implementing SLA-tiered traffic management in Spring WebFlux (Hebbar, 2025). Hebbar's work serves as the anchor around which other sources are organized, allowing the study to maintain a clear focus on service differentiation rather than on reactive programming in the abstract. The methodology involves

Volume11 Issue01, January-2026, pg. 121-131

Published Date: - 31-01-2026

E-ISSN: 2536-7919
P-ISSN: 2536-7900

decomposing Hebbar's framework into its constituent elements, including request classification, priority tagging, reactive operator selection, and backpressure propagation, and then mapping these elements onto the broader ecosystem of Java concurrency and cloud-native infrastructure.

To ensure analytical rigor, the study systematically reviews and synthesizes comparative evaluations of Spring WebFlux and Spring MVC, as well as discussions of asynchronous frameworks and database connection pools, to identify how non-blocking execution interacts with resource contention and throughput (Sukhambekova, 2025; Ju et al., 2024). These sources provide empirical and experiential evidence that informs the interpretive analysis of priority-aware designs. In particular, performance comparisons from practitioner blogs and academic theses are treated not as definitive benchmarks but as indicative patterns that reveal how reactive systems behave under different workloads (Filichkin, 2018; KTH, 2023).

A key methodological step involves constructing a conceptual model of a financial microservices platform that incorporates multiple SLA tiers, such as premium trading clients, standard retail users, and background regulatory processes. Within this model, requests are represented as reactive streams flowing through Spring WebFlux controllers, service layers, and data access components. Priority metadata is attached to each request at ingress, following the approach proposed by Hebbar, and is propagated through the reactive pipeline via custom schedulers and operator chains (Hebbar, 2025). This conceptualization enables the analysis to trace how priority information influences scheduling decisions, I O allocation, and backpressure behavior across the system.

The study also integrates insights from the Java concurrency ecosystem, particularly the Runnable interface, the java.util.concurrent package, and the Flow API, which underpins the reactive streams specification used by Reactor (Oracle, 2025a; Oracle, 2025b; Oracle, 2025c). By situating Spring WebFlux within this broader context, the methodology clarifies how priority-aware scheduling can be implemented at different layers, from application-level reactive operators to lower-level thread pools and event loops. This layered perspective is essential for understanding how SLA enforcement can be achieved without violating the non-blocking and backpressure principles that define reactive programming.

Limitations of the methodological approach are acknowledged as an integral part of the design. Because the study relies on secondary sources and conceptual modeling rather than on original benchmark experiments, its conclusions are necessarily interpretive and contingent on the validity of the underlying literature. However, this limitation is mitigated by the breadth and diversity of the sources considered, which include academic conference papers, master's theses, practitioner reports, and authoritative documentation. By triangulating these perspectives, the methodology seeks to produce a robust and nuanced account of priority-aware reactive architectures that is grounded in both theory and practice (Marinescu, 2013; Kibe et al., 2013).

Volume11 Issue01, January-2026, pg. 121-131

Published Date: - 31-01-2026

E-ISSN: 2536-7919
P-ISSN: 2536-7900

Another methodological constraint arises from the rapidly evolving nature of Java and Spring ecosystems, particularly with the introduction of virtual threads and other concurrency enhancements. While recent studies have begun to compare these models with reactive frameworks, their long-term implications for SLA-tiered systems remain uncertain (Nordlund and Nordstrom, 2023). The methodology therefore treats such developments as part of an ongoing scholarly debate rather than as settled facts, incorporating them into the discussion as alternative design trajectories.

In sum, the methodology of this study is designed to provide a comprehensive and theoretically informed analysis of priority-aware reactive APIs in financial services. By combining Hebbar's framework with a wide-ranging synthesis of reactive programming and cloud infrastructure literature, it establishes a rigorous foundation for the interpretive results and discussion that follow (Hebbar, 2025; Sharma, 2018).

RESULTS

The results of the analytical synthesis reveal a complex but coherent picture of how priority-aware reactive architectures can mediate between heterogeneous service-level requirements and the technical realities of non-blocking, cloud-native systems. At the core of these findings is the observation that reactive programming, when augmented with explicit priority semantics, offers a qualitatively different mode of service differentiation than traditional thread-based or load-balancing approaches (Hebbar, 2025). Rather than segregating traffic at the infrastructure level through separate clusters or queues, priority-aware reactive design embeds service-level distinctions directly into the execution flow of the application.

One of the most significant results concerns the interaction between request priority and reactive scheduling. In a standard Spring WebFlux application, incoming HTTP requests are transformed into reactive streams and processed by a small number of event-loop threads that coordinate non-blocking I O and business logic (Deinum and Cosmina, 2021). Without priority awareness, these event loops treat all streams uniformly, which can lead to situations where a surge of low-priority requests delays the processing of high-priority ones. Hebbar's framework demonstrates that by introducing priority tags and custom schedulers, it is possible to bias the execution order of reactive streams so that premium requests are serviced preferentially, even under heavy load (Hebbar, 2025).

This behavior aligns with findings from asynchronous framework research, which indicate that non-blocking systems are highly sensitive to scheduling policies and queue management (Rankovski and Chorbev, 2016). By controlling the order in which reactive operators are invoked and by allocating separate thread pools or event loops to different priority classes, a WebFlux application can achieve differentiated latency profiles without sacrificing overall throughput. Comparative studies of Spring MVC and WebFlux support this conclusion by showing that reactive frameworks can maintain lower response times under concurrency when properly tuned (Sukhambekova, 2025; Li and Sharma, 2020).

INTERNATIONAL JOURNAL OF COMPUTER SCIENCE & INFORMATION SYSTEM

Volume11 Issue01, January-2026, pg. 121-131

Published Date: - 31-01-2026

E-ISSN: 2536-7919
P-ISSN: 2536-7900

Another key result pertains to backpressure and flow control. In reactive streams, backpressure is the mechanism by which consumers signal their capacity to producers, thereby preventing resource exhaustion. When priority metadata is propagated through the stream, backpressure can be modulated according to SLA tiers, allowing high-priority consumers to receive data even when low-priority consumers are throttled (Hebbar, 2025). This selective backpressure is particularly relevant in financial services, where real-time market data and transaction processing must not be impeded by less critical background tasks. Empirical observations from performance comparisons with databases and search engines suggest that such fine-grained flow control can mitigate bottlenecks at the persistence layer (Minkowski, 2019; Dorado, 2019).

The results also highlight the role of database connection pools and asynchronous data access in priority-aware architectures. Ju and colleagues have shown that asynchronous frameworks combined with optimized connection pools can significantly enhance performance in high-concurrency environments (Ju et al., 2024). When integrated with priority-aware reactive pipelines, these mechanisms allow premium requests to acquire database connections more readily, while lower-priority queries are queued or delayed. This layered prioritization, spanning from HTTP ingress to data storage, embodies the holistic approach advocated by Hebbar (Hebbar, 2025).

A further result emerges from the comparison between reactive frameworks and virtual threads. Studies from KTH and Nordlund and Nordstrom suggest that virtual threads offer a simpler programming model with scalability comparable to reactive systems under certain workloads (KTH, 2023; Nordlund and Nordstrom, 2023). However, these studies do not account for SLA-tiered traffic, and the present analysis indicates that without explicit priority-aware scheduling, virtual-thread-based systems may struggle to provide differentiated service under contention. Reactive frameworks, by contrast, can incorporate priority semantics into their stream processing, offering a more direct path to SLA enforcement (Hebbar, 2025).

Taken together, these results demonstrate that priority-aware reactive architectures can achieve a nuanced balance between scalability, efficiency, and service differentiation. By embedding SLA tiers into the reactive execution model, financial applications can align their technical behavior with business and regulatory requirements in a way that is not readily achievable through infrastructure-level segregation alone (Marinescu, 2013; Kibe et al., 2013).

DISCUSSION

The findings of this study invite a deeper theoretical and practical reflection on the nature of concurrency, service differentiation, and architectural design in modern financial systems. At a theoretical level, priority-aware reactive APIs challenge the traditional separation between business logic and infrastructure by embedding service-level semantics directly into the computational model. In conventional architectures, SLAs are often enforced through external mechanisms such as load balancers,

Volume11 Issue01, January-2026, pg. 121-131

Published Date: - 31-01-2026

E-ISSN: 2536-7919
P-ISSN: 2536-7900

rate limiters, or separate deployment tiers. While these mechanisms can provide coarse-grained differentiation, they do not influence the internal scheduling and flow control of the application, which remains largely oblivious to the relative importance of different requests (Filichkin, 2018; Munhoz, 2020).

Hebbar's framework, by contrast, treats priority as a first-class concern of the reactive pipeline, thereby integrating business-level commitments with low-level execution semantics (Hebbar, 2025). This integration resonates with broader trends in software engineering toward self-aware and policy-driven systems, where applications are designed to adapt their behavior in response to contextual information. From this perspective, priority-aware reactive APIs can be seen as an instantiation of policy-based resource management, in which SLA tiers act as policies that guide scheduling, I O allocation, and backpressure.

The scholarly debate on reactive versus blocking or virtual-thread-based concurrency provides a rich context for evaluating this approach. Proponents of reactive programming emphasize its scalability and resilience, arguing that non-blocking I O and event-driven execution are essential for handling the massive concurrency of modern web applications (Sharma, 2018; Mednikov, 2021). Critics, however, point to the complexity of reactive code and the cognitive burden it places on developers, suggesting that newer concurrency models such as virtual threads may offer a more accessible alternative (KTH, 2023; Nordlund and Nordstrom, 2023). The present analysis adds a new dimension to this debate by highlighting the importance of service differentiation. Even if virtual threads simplify development, they do not inherently provide mechanisms for prioritizing certain workloads over others, which is a critical requirement in financial services.

Another point of discussion concerns the interaction between reactive programming and persistent data sources. Wycislik and Ogorek have documented performance issues that arise when reactive systems interact with blocking databases, leading to thread starvation and throughput collapse (Wycislik and Ogorek, 2019). Priority-aware reactive design does not eliminate these issues, but it can mitigate their impact by ensuring that high-priority requests are less likely to be blocked by low-priority ones at the data access layer. Nevertheless, this requires careful integration of asynchronous database drivers and connection pools, as emphasized by Ju and colleagues (Ju et al., 2024).

From a cloud infrastructure perspective, priority-aware reactive APIs align well with the multi-tenant and elastic nature of modern platforms. Cloud providers offer fine-grained control over compute, memory, and networking resources, but these controls are typically applied at the level of virtual machines or containers rather than at the level of individual requests (Marinescu, 2013). By embedding priority semantics into the application, developers can exploit cloud elasticity more effectively, scaling resources in response to the demand of different SLA tiers rather than to aggregate load alone (Kibe et al., 2013).

However, the approach is not without limitations. One concern is the potential for starvation of low-priority requests if high-priority traffic is sustained over long periods. While this may be acceptable or

Volume11 Issue01, January-2026, pg. 121-131

Published Date: - 31-01-2026

E-ISSN: 2536-7919
P-ISSN: 2536-7900

even desirable in some financial contexts, such as during market crises or regulatory deadlines, it raises ethical and operational questions about fairness and inclusivity. Designing appropriate policies for priority assignment and backpressure modulation therefore requires not only technical expertise but also organizational governance.

Another limitation is the complexity of implementing and maintaining priority-aware reactive pipelines. Custom schedulers, priority tags, and policy-driven operators add layers of abstraction that can complicate debugging and monitoring. Tooling and observability frameworks must evolve to provide visibility into how priorities influence execution, otherwise operators may struggle to diagnose performance issues (Hebbar, 2025).

Future research should explore these challenges in greater depth, particularly through empirical studies that measure the impact of priority-aware designs on latency, throughput, and fairness across a range of financial workloads. The emergence of hybrid models that combine reactive streams with virtual threads also warrants investigation, as such models may offer new ways to balance simplicity and service differentiation (Nordlund and Nordstrom, 2023).

In theoretical terms, priority-aware reactive APIs invite a reconsideration of how software architectures embody social and economic values. By encoding SLA tiers into the fabric of computation, developers are effectively translating contractual and regulatory obligations into executable policies. This translation process raises important questions about accountability, transparency, and the limits of automation in financial systems, which deserve sustained scholarly attention (Hebbar, 2025; Fowler, 2003).

CONCLUSION

This article has presented a comprehensive and theoretically grounded analysis of priority-aware reactive APIs for SLA-tiered financial services, drawing on Hebbar's framework and a wide range of literature on Spring WebFlux, asynchronous I O, and cloud-native architectures. The findings demonstrate that reactive programming, when augmented with explicit priority semantics, offers a powerful means of aligning technical execution with business-level service commitments. By embedding SLA tiers into the reactive pipeline, financial applications can achieve differentiated latency, throughput, and reliability without resorting to coarse-grained infrastructure segregation.

At the same time, the analysis has highlighted the complexity and trade-offs inherent in this approach, including the challenges of integrating persistent data sources, managing fairness, and maintaining observability. These challenges underscore the need for continued research and innovation at the intersection of reactive programming, concurrency models, and service-level management.

In an era where financial services are increasingly delivered through APIs and microservices, the ability to enforce differentiated service levels is not merely a technical convenience but a strategic and regulatory necessity. Priority-aware reactive architectures, as articulated by Hebbar and elaborated in this study,

Volume11 Issue01, January-2026, pg. 121-131

Published Date: - 31-01-2026

E-ISSN: 2536-7919
P-ISSN: 2536-7900

provide a compelling blueprint for meeting this necessity in a scalable, resilient, and policy-driven manner (Hebbar, 2025).

REFERENCES

1. Catrina, A. V. A Comparative Analysis of Spring MVC and Spring WebFlux in Modern Web Development. 2023. Available online: https://www.theseus.fi/bitstream/handle/10024/812448/Catrina_Alexandru.pdf
2. Hebbar, K. S. (2025). Priority-Aware reactive APIs: Leveraging Spring WebFlux for SLA-Tiered traffic in financial services. European Journal of Electrical Engineering and Computer Science, 9(5), 31–40. https://doi.org/10.24018/ejece.2025.9.5.743
3. Marinescu, D. C. Cloud Infrastructure; Cloud Computing: Theory and Practice. 2013, 67–98.
4. Munhoz, G. API Performance Spring MVC vs Spring WebFlux vs Go. Medium, Aug. 2020. Available: https://filipemunhoz.medium.com/apiperformance-spring-mvc-vs-spring-webflux-vs-gof97b62d2255a
5. Ju, L., Yadav, A., Khan, A., Sah, A. P., Yadav, D. Using Asynchronous Frameworks and Database Connection Pools to Enhance Web Application Performance in High-Concurrency Environments. 2024 8th International Conference on I-SMAC, 742–747.
6. Deinum, C., Cosmina, I. Building Reactive Applications with Spring WebFlux. In Spring in Action, 5th ed., Manning, 2021.
7. Oracle. java.util.concurrent Package Summary. Java SE 21 Documentation, 2025.
8. Nordlund, A., Nordstrom, N. Comparing Virtual Threads and Reactive WebFlux in Spring. divaportal.org, 2023.
9. Sharma, R. Hands-On Reactive Programming with Reactor: Build Reactive and Scalable Microservices Using the Reactor Framework. Packt, 2018.
10. Wycislik, L., Ogorek, L. Issues on Performance of Reactive Programming in the Java Ecosystem with Persistent Data Sources. In Man-Machine Interactions 6, 2019.
11. KTH Royal Institute of Technology. Comparing Virtual Threads and Reactive WebFlux in Spring. M.S. Thesis, 2023.
12. Minkowski, P. Performance Comparison Between Spring MVC vs Spring WebFlux with Elasticsearch. Personal Blog, 2019.
13. Rao, R., Swamy, S. R. Review on Spring Boot and Spring WebFlux for Reactive Web Development. International Research Journal of Engineering and Technology, 2020.
14. Filichkin, A. Spring Boot Performance Battle: Blocking vs Non-Blocking vs Reactive. Medium, 2018.
15. Mednikov, Y. Friendly WebFlux: A Practical Guide to Reactive Programming with Spring WebFlux. Independent, 2021.
16. Sukhambekova, A. Comparison of Spring WebFlux and Spring MVC. Modern Scientific Method, no. 9, 2025.

Volume11 Issue01, January-2026, pg. 121-131

Published Date: - 31-01-2026

E-ISSN: 2536-7919
P-ISSN: 2536-7900

17. Rankovski, G., Chorbev, I. Improving Scalability of Web Applications by Utilizing Asynchronous I O. ICT Innovations 2016.
18. Kibe, S., Watanabe, S., Kunishima, K., Adachi, R., Yamagiwa, M., Uehara, M. PaaS on IaaS. IEEE AINA, 2013.
19. Li, Q., Sharma, R. Review on Spring Boot and Spring WebFlux for Reactive Web Development. ResearchGate, 2020.
20. Dorado, F. Reactive vs Non-Reactive Spring Performance. Personal Blog, 2019.
21. Oracle. Runnable Interface. Java SE 21 Documentation, 2025.
22. Oracle. Flow API. Java SE 21 Documentation, 2025.
23. Fowler, M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley, 2003.