JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 39 (2026) 154–160
Received: 23 March 2026
Accepted: 5 May 2026

Comparative Analysis of Reactive Programming and Java Virtual Threads

Jakub Brzeziński*, Daniel Charlak*, Grzegorz Kozieł

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

Abstract

This study presents a comparative analysis of two concurrency paradigms in modern Java applications: reactive programming and Java virtual threads. Traditional operating system threads consume excessive resources under high concurrency, which has driven the adoption of non-blocking, event-driven frameworks such as Spring WebFlux, as well as lightweight JVM-managed virtual threads introduced as part of Project Loom in Java 21. While prior research highlights the benefits of both approaches, direct empirical comparisons under controlled conditions remain limited. To address this gap, this study evaluates the performance, scalability, and resource utilization of both paradigms through controlled experiments conducted on the Google Cloud Platform. The analysis investigates their behaviour under varying levels of concurrent load and examines the impact of different CPU and memory configurations on system efficiency. The results aim to support developers and architects in selecting appropriate concurrency models for Java applications.

Keywords: Java 21; Virtual threads; Spring WebFlux; Concurrency

*Corresponding authors

Email address: s95372@pollub.edu.pl (J. Brzeziński), s95374@pollub.edu.pl (D. Charlak)

1. Introduction

Concurrent request handling is fundamental to modern web services. Developers traditionally faced an important choice: maintaining code simplicity through traditional threads or improving resource efficiency through reactive programming [1]. This decision has shaped Java application architecture for over a decade, forcing teams to choose between development speed and system efficiency.

The introduction of virtual threads in Java 21 changes this situation by offering lightweight concurrency without sacrificing imperative programming style [2]. Unlike reactive approaches that require functional composition and complex stream abstractions, virtual threads preserve the familiar threading model while consuming significantly less memory compared to operating-system threads [3]. Virtual threads represent a structured concurrency model for scalable applications on the JVM, managed directly by the runtime rather than the operating system [4]. Despite the advantages of reactive programming paradigms in improving software comprehension and reducing boilerplate code [5], practical comparison information between virtual threads and reactive programming under realistic workloads remains limited.

This study addresses this gap by conducting an empirical comparison of two concurrency approaches within the Spring Boot framework: virtual threads and Spring WebFlux. The comparison evaluates three critical dimensions: response time, CPU utilization, and memory consumption. The experiments were performed on Google Cloud Platform using identical CRUD microservices under varying hardware configurations (4/4, 16/16, and 32/32 CPU cores with corresponding RAM) and progressive load levels (100, 500, and 1,000 concurrent requests). This approach ensures a fair comparison that captures realistic deployment scenarios faced by cloud-native applications.

2. Literature Review

Recent research on concurrency mechanisms in the Java ecosystem centers around three major areas: virtual threads, asynchronous programming, and the reactive paradigm [6]. A review of the literature reveals a significant methodological gap. Direct comparisons between the performance of virtual threads and reactive programming are absent, despite the parallel development of both paradigms.

In high-concurrency scenarios, the reactive model shows significant advantages over synchronous programming, achieving a 12% reduction in memory usage and a 56% improvement in 90th-percentile response times compared with asynchronous approaches [7]. In tests involving 1,000 concurrent users, throughput increased by 33% and response times decreased by 25%. However, the omission of virtual threads limits the assessment of their potential, while simultaneously highlighting the strengths of reactive programming in concurrency-heavy environments.

Key to understanding the capabilities of virtual threads are the studies by Lasić et al. [8], which compared their performance with traditional threads in server applications interacting with different database systems. In tests using PostgreSQL, virtual threads achieved notable latency reductions from 14.83% (GET) to 20.76% (DELETE) for CRUD operations, confirming their effectiveness in I/O-intensive environments. Tests with Oracle showed only moderate improvements (4.49–12.55%), suggesting that the efficiency of virtual threads depends heavily on the specific database system and the nature of executed operations.

Research by Iwanowski and Kozieł [9] provides important insights into the performance and stability of various concurrency paradigms in Java. In tests with 2,000 simultaneous requests, the reactive application (Spring WebFlux/Netty) processed queries 33% faster than the

imperative version (Spring MVC/Tomcat) in scenarios with delays exceeding 10 seconds, while using 18–22% less CPU. In resource-constrained environments (2 CPU cores, 4 GB RAM), the reactive server handled 2.4 times more requests per second for operations with 500 ms latency, maintaining 91% scalability across three instances. However, for operations involving large CSV files (50+ MB), the imperative model achieved 15–20% higher throughput, which the authors attribute to stream-processing overhead in Reactor. The study found no significant differences in RAM usage, but both models consumed 1.2–1.5 GB with 3,000 users.

Mochniej and Badurowicz [10] show that comparisons of reactive and imperative microservices remain a relatively new research area, with results often being inconclusive. In one experiment, a reactive application achieved 26–27% lower latency when adding a product (with 100 users), but was more than three times slower when errors occurred. In delayed-response scenarios, the reactive system performed twice as fast, underscoring its strengths in high-latency network environments. Studies comparing Spring Boot and Quarkus revealed that reactive microservices in Quarkus consumed fewer resources than their Spring Boot counterparts, highlighting the importance of framework selection [11].

The literature clearly shows that both virtual threads and reactive programming significantly improve the performance and concurrency of Java web applications, especially within the Spring ecosystem [12]. The reactive model excels in heavily loaded environments and microservices with limited resources. Virtual threads, in turn, simplify code, lower the entry barrier, and improve performance in high-I/O workloads while preserving the classical imperative programming style. However, the lack of direct comparisons makes it difficult for developers to make informed architectural choices. This study aims to fill that gap by empirically evaluating both paradigms under varied workload conditions.

3. Objective and Scope of the Study

The primary objective of this study is to empirically compare the performance characteristics of two concurrency models in the Java ecosystem: virtual threads introduced in Java 21 and reactive programming implemented using Spring WebFlux.

Based on the literature review and the architectural characteristics of the analyzed approaches, the following research hypotheses were formulated:

1. Applications utilizing virtual threads in Java 21 consume more memory than applications implemented using Spring WebFlux under equivalent workload conditions.
2. Under high levels of concurrent HTTP requests, applications using Spring WebFlux exhibit higher average response times than applications based on Java virtual threads.
3. With increasing load, applications based on virtual threads are more prone to CPU saturation compared to applications utilizing Spring WebFlux.

4. Research Methodology

This study uses an experimental comparative method to evaluate two concurrency approaches: Java virtual threads and Spring WebFlux. Both implementations were tested under identical, controlled conditions to measure the impact of each concurrency model on performance.

4.1. Study Objects

Two concurrency models widely used in modern Java systems were analyzed: Java virtual threads (Java 21) and reactive programming using Spring WebFlux. Both models were implemented as functionally equivalent Spring Boot REST services performing identical CRUD operations on product resources, ensuring comparable and repeatable workloads. The CRUD microservice scenario was selected due to its common use in enterprise systems and its I/O-intensive nature.

The applications shared the same data model, API structure, database queries, and request-processing logic. No framework-specific performance optimizations were applied. The reactive implementation relied on the default Spring WebFlux execution model and scheduler configuration, while the virtual-thread version used the default JVM virtual thread executor. This ensured that the comparison reflects typical production usage rather than manually tuned setups.

Both variants used a local PostgreSQL database hosted within the same virtual machine, executing identical SQL queries against the same schema. The virtual thread implementation used the JDBC driver for database connectivity, while the reactive implementation used the R2DBC driver to ensure a fully non-blocking connection. This configuration minimized external network variability and allowed performance differences to be attributed primarily to the concurrency model.

The shared data model consists of a single users table with three columns: id (BIGINT, primary key), username (VARCHAR), and email (VARCHAR), pre-populated with 1,000 records. The primary tested endpoint was GET /api/users, which retrieves all records from the database and returns them as a JSON array. In the virtual thread implementation, this operation was handled by a repository extending JpaRepository, executing a blocking JDBC query. In the reactive implementation, the equivalent repository extended R2dbcRepository, returning a Flux<User> via a non-blocking R2DBC connetion.

4.2. Experimental Scenarios

The experimental procedure consisted of multiple stages. First, both application variants were deployed on GCP virtual machines configured with different CPU and RAM allocations to simulate diverse infrastructure capacities.

Load was generated using Apache Benchmark (AB), which produced a controlled stream of concurrent HTTP requests without think-time delays. Although this tool does not fully model real user behavior, it provides deterministic load conditions suitable for stress-testing server-side concurrency mechanisms. Concurrency levels of 100, 500, and 1,000 simultaneous requests were applied

progressively. Each workload level was executed for a 30-second measurement interval.

Each scenario was repeated ten times, and the reported results represent average values across all runs to reduce the influence of transient fluctuations. During each test, key performance metrics were collected, including response time, throughput, CPU utilization, and memory consumption. Scalability was assessed based on the ability of the systems to maintain stable performance as concurrency increased.

4.3. Experimental Setup

All experiments were conducted in a controlled cloud environment to ensure consistency and repeatability of results. Both application variants were deployed on Google Cloud Platform e2-highcpu virtual machines running Ubuntu 22.04 LTS. Three hardware profiles were tested: e2-highcpu-4 (4 vCPU / 4 GB RAM), e2-highcpu-16 (16 vCPU / 16 GB RAM), and e2-highcpu-32 (32 vCPU / 32 GB RAM). The environment was orchestrated using Docker Compose, which managed the application containers alongside a PostgreSQL 15 database instance. Both applications were built with Spring Boot 3.5.3, using the PostgreSQL JDBC driver 42.7.7 for the virtual thread variant and the R2DBC PostgreSQL driver 1.0.7.RELEASE for the reactive variant. Each test scenario was executed on isolated instances to prevent interference from other processes.

The applications were executed using Java 21 with the default JVM configuration. No manual tuning of heap size, garbage collection, thread scheduling, or framework-level performance parameters was applied. This ensured that both concurrency models operated under comparable, production-style runtime conditions. Prior to each measurement, the JVM underwent a warm-up phase to allow just-in-time (JIT) compilation and runtime optimizations to stabilize.

The reactive implementation relied on the default Spring WebFlux execution model, while the virtual-thread version used the default JVM virtual thread executor. No custom scheduler adjustments or thread pool modifications were introduced.

A local PostgreSQL database instance served as the persistence layer for both application variants. The database was hosted within the same virtual machine as the application, and both systems executed identical SQL queries against the same schema. This configuration minimized network-induced variability and ensured that performance differences were primarily attributable to the concurrency mechanisms rather than external I/O latency.

System-level monitoring tools were used to collect CPU utilization and memory consumption metrics throughout each test. Each experimental configuration was executed ten times, and the reported values represent averages across all runs to reduce the influence of transient performance fluctuations.

4.4. Performance Metrics

The following metrics were used to evaluate system behavior and performance:

- Response time - measured in milliseconds, capturing both average latency and latency distribution under load.
- Throughput (Requests per Second) – the number of successfully processed HTTP requests per second, indicating overall system capacity.
- CPU utilization - the percentage of processor resources consumed during workload execution.
- Memory usage - the amount of RAM consumed by the application, used to assess memory efficiency under different concurrency models.

5. Results

This section presents the results of the performance evaluation across all tested metrics, hardware configurations, and concurrency levels.

5.1. Response Time

Table 1 presents the average response time per request for both concurrency models across all tested configurations.

Table 1: Average response time per request (ms)

		Hardware configuration (vCPU / RAM)		
		4/4	16/16	32/32
Number of concurrent requests	Type	Average execution time (ms) (mean ± standard deviation)		
100	VT	1377.15 ± 39.9	172.21 ± 13.9	160.35 ± 16.2
	RC	4753.77 ± 1639.6	880.00 ± 11.5	849.37 ± 5.5
500	VT	8970.22 ± 217.5	866.79 ± 9.7	793.22 ± 53.3
	RC	18815.48 ± 209.5	4741.50 ± 11.4	4337.50 ± 3.1
1000	VT	18220.45 ± 410.1	1755.88 ± 14.7	1540.12 ± 39.3
	RC	40139.13 ± 358.5	10138.85 ± 22.9	9066.90 ± 9.1

Virtual Threads achieve lower average response times across all tested scenarios. The performance gap is most pronounced on smaller instances and under high concurrency, while increasing hardware resources reduce response times for both approaches, as illustrated in Figure 1.

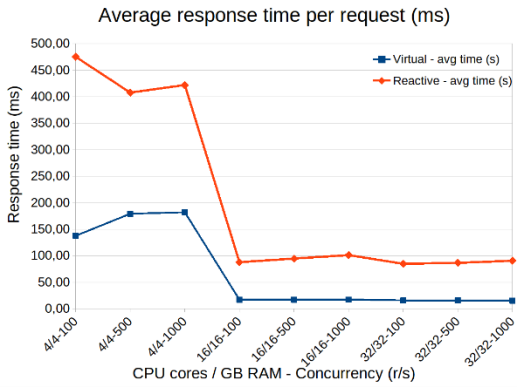

Figure 1: Average response time under increasing concurrent load

5.2. CPU Utilization

Table 2 presents the CPU utilization measurements for both models across all tested configurations.

Table 2: Average CPU usage (%)

		Hardware configuration (vCPU / RAM)		
		4/4	16/16	32/32
Number of concurrent requests	Type	Average CPU usage (%) (mean ± standard deviation)		
100	VT	70.71 ± 3.3	52.28 ± 17.8	49.84 ± 22.3
	RC	86.18 ± 3.1	70.95 ± 0.3	41.37 ± 1.9
500	VT	72.49 ± 7.7	51.08 ± 18.6	61.18 ± 17.9
	RC	81.81 ± 14.9	66.69 ± 9.5	41.08 ± 3.3
1000	VT	66.38 ± 20.7	56.75 ± 14.4	58.61 ± 26.0
	RC	81.29 ± 14.8	63.99 ± 14.2	37.97 ± 5.8

Reactive programming exhibits higher CPU utilization on smaller virtual machine configurations, particularly under high concurrency levels. In contrast, Virtual Threads maintain more stable CPU usage across increasing workloads. As available computing resources increase, CPU utilization decreases for both approaches; however, the reduction is more pronounced for Reactive programming, as illustrated in Figure 2.

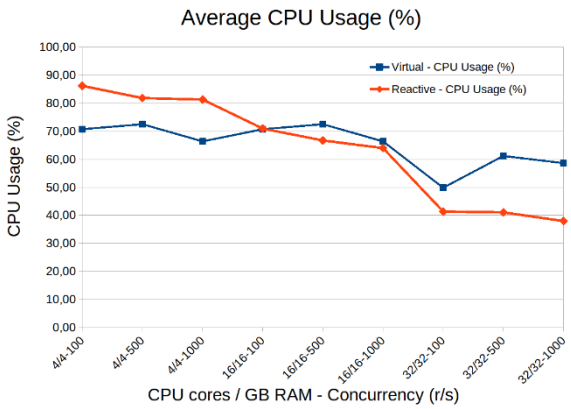

Figure 2: Average CPU utilization under increasing concurrent load

5.3. Heap Memory Consumption

Table 3 presents the heap memory usage for both concurrency models across all tested configurations.

Table 3: Average heap memory usage (MiB)

		Hardware configuration (vCPU / RAM)		
		4/4	16/16	32/32
Number of concurrent requests	Type	Average heap usage (MiB) (mean ± standard deviation)		
100	VT	980.00 ± 2.1	3105.65 ± 143.8	5381.63 ± 366.0
	RC	343.05 ± 21.4	816.50 ± 19.9	1152.66 ± 29.5
500	VT	944.15 ± 52.6	3983.87 ± 18.3	5089.77 ± 194.9
	RC	391.70 ± 32.3	1088.01 ± 55.0	1498.62 ± 257.8
1000	VT	902.77 ± 133.6	3991.65 ± 8.9	5886.98 ± 300.0
	RC	426.28 ± 50.4	1132.35 ± 54.6	1596.16 ± 147.6

Virtual Threads consistently exhibit higher heap memory consumption than Reactive programming across all tested scenarios. The difference increases with higher concurrency levels and larger virtual machine configurations. While heap usage grows for both approaches as the workload increases, Reactive programming remains below the memory consumption levels observed for Virtual Threads in all configurations, as illustrated in Figure 3.

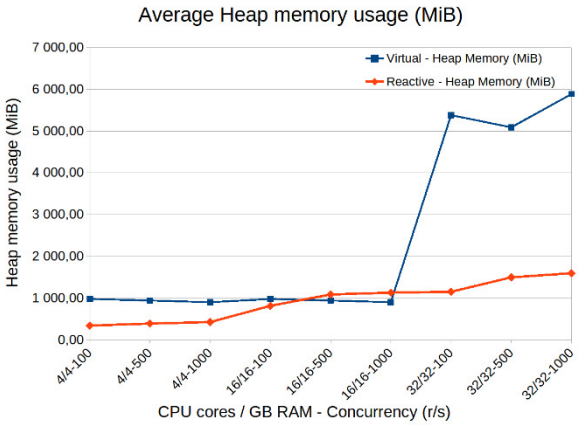

Figure 3: Average Heap memory consumption under increasing concurrent load

5.4. Throughput

Table 4 presents the throughput measurements for both models across all tested configurations.

Table 4: Average throughput (req/s)

		Hardware configuration (vCPU / RAM)		
		4/4	16/16	32/32
Number of concurrent requests	Type	Average throughput (req/s) (mean ± standard deviation)		
100	VT	72.67 ± 1.87	582.81 ± 39.55	627.72 ± 60.41
	RC	22.79 ± 4.65	113.65 ± 1.54	117.73 ± 0.91
500	VT	55.77 ± 1.33	576.92 ± 7.68	629.94 ± 26.48
	RC	24.53 ± 0.21	105.48 ± 1.72	115.28 ± 0.47
1000	VT	54.92 ± 1.43	569.55 ± 4.85	649.56 ± 18.11
	RC	23.69 ± 0.21	98.66 ± 2.45	110.30 ± 1.33

Virtual Threads achieve higher throughput than Reactive programming across all tested scenarios. The difference becomes more pronounced on larger virtual machine configurations, where throughput increases with the number of available CPU cores. In contrast, Reactive programming exhibits limited growth as infrastructure resources increase, as illustrated in Figure 4.

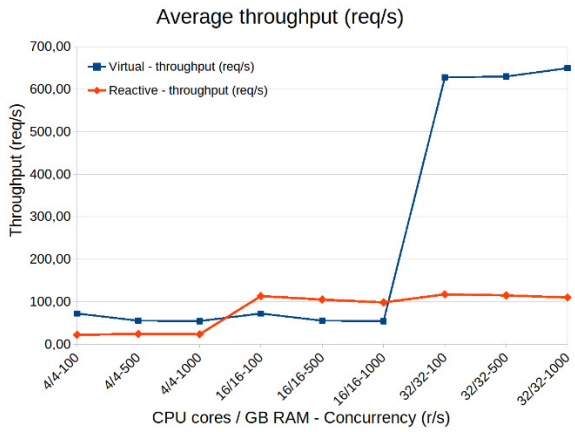

Figure 4. Average throughput under increasing concurrent load

6. Discussion

The experimental results reveal distinct performance and resource utilization characteristics associated with the two analyzed concurrency paradigms. While both models enable scalable handling of concurrent requests, their operational behavior differs substantially due to their underlying execution mechanisms.

6.1. Response Time Characteristics

Applications using virtual threads consistently demonstrated lower average response times across all infrastructure configurations and workload levels. This behavior can be attributed to the thread-per-request model, which allows blocking I/O operations to be handled without the overhead typically associated with operating system threads. The lightweight scheduling of virtual threads enables a more direct mapping between incoming requests and execution contexts, reducing coordination overhead.

In contrast, reactive applications rely on an event-driven execution model based on a limited number of event-loop threads. Under high concurrency, this approach introduces additional scheduling, context propagation, and stream-processing overhead, which may contribute to increased latency. These effects become more visible under heavy load and on smaller instances where computational resources are constrained. These findings confirm Hypothesis 2.

6.2. CPU Utilization Patterns

CPU utilization measurements indicate that reactive programming tends to consume more processor resources on smaller virtual machines. This may result from the additional processing layers associated with reactive stream operators and event-loop coordination. As hardware resources increase, CPU utilization decreases for both models, suggesting improved scalability. These results do not support Hypothesis 3. Virtual threads exhibit relatively stable CPU usage across different load levels, indicating efficient task scheduling by the JVM runtime.

6.3. Heap Memory Consumption

Heap memory consumption represents the most pronounced difference between the two paradigms. Virtual-thread-based applications consistently required substantially more heap memory. This is expected, as each virtual thread maintains its own stack and scheduling metadata. As concurrency increases, the cumulative memory footprint grows accordingly. Reactive programming, by contrast, relies on a smaller fixed thread pool and non-blocking processing, resulting in a more compact memory profile. This confirms that reactive architectures remain advantageous in memory-constrained environments or in scenarios involving extremely high numbers of concurrent requests. These findings confirm Hypothesis 1.

6.4. Throughput and Scalability Implications

Throughput measurements align with response time observations: virtual threads achieve higher request processing rates across all configurations. Their ability to handle blocking operations efficiently appears to translate directly into higher system capacity. Reactive programming exhibits more modest throughput scaling, suggesting that event-driven architectures may incur additional processing overhead that limits peak performance under the tested workload.

6.5. Limitations

The results should be interpreted in the context of the study's scope. The evaluated workload represents an I/O-intensive CRUD scenario with a local database deployment. The observed performance characteristics may differ in CPU-bound systems, highly distributed architectures, or in environments with non-blocking database drivers. Nevertheless, the findings clearly demonstrate that the choice between virtual threads and reactive programming involves a fundamental trade-off between latency/throughput performance and memory efficiency.

7. Conclusions

This study provided a controlled empirical comparison of Java virtual threads and reactive programming using Spring WebFlux under identical application logic, database configuration, and cloud infrastructure. The results demonstrate that both concurrency paradigms are capable of supporting high levels of concurrent request processing, but they emphasize different performance characteristics.

Virtual threads consistently achieved lower response times and higher throughput, particularly under high concurrency and on smaller infrastructure configurations. Their lightweight thread model allows efficient handling of blocking I/O operations while preserving a straightforward programming model. However, this advantage comes at the cost of significantly higher heap memory consumption, which may become a limiting factor in memory-constrained environments.

Reactive programming exhibited superior memory efficiency and more compact resource usage, confirming its suitability for systems where minimizing memory footprint is critical. Although latency and throughput were generally lower in the evaluated scenarios, reactive architectures provide predictable behavior under high connection volumes and remain well suited for large-scale, resource-optimized deployments.

The findings indicate that no single concurrency paradigm is universally optimal. Virtual threads are advantageous in latency-sensitive, I/O-bound services where memory resources are sufficient, whereas reactive programming remains preferable in environments prioritizing memory efficiency and high connection density. Future research should extend this comparison to CPU-bound workloads, distributed database environments, and fully non-blocking technology stacks to further clarify the applicability of both approaches.

References

[1] A. Navarro, J. Ponge, F. Le Mouël, C. Escoffier, Analysing the performance and costs of reactive programming libraries in Java, In 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS) (2021) 51-60, https://doi.org/10.1145/3486605.3486788.

[2] Oracle, Java 21 Virtual Threads Documentation, https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html, [12.10.2025].

[3] OpenJDK, JDK Enhancement Proposal 444: Virtual Threads, https://openjdk.org/jeps/444, [13.10.2025].

[4] D. Beronić, P. Pufek, B. Mihaljević, A. Radovan, On Analyzing Virtual Threads – a Structured Concurrency Model for Scalable Applications on the JVM, In 2021 44th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO) IEEE (2021) 1684–1689, https://doi.org/10.23919/MIPRO52101.2021.9596855.

[5] J. P. Briot, R. Guerraoui, K. P. Lohr, Concurrency and distribution in object-oriented programming, ACM Computing Surveys 30(3) (1998) 291–329, https://doi.org/10.1145/292469.292470.

[6] F. Myter, C. Scholliers, W. De Meuter, Distributed Reactive Programming for Reactive Distributed Systems, The Art, Science, and Engineering of Programming 3(3) (2019) 5,https://doi.org/10.22152/programming-journal.org/2019/3/5.

[7] A. Zbarcea, C. Tudose, Migrating from Developing Asynchronous Multi-Threading Programs to Reactive Programs in Java, Applied Sciences 14(24) (2024) 12062, https://doi.org/10.3390/app142412062.

[8] L. Lašić, D. Beronić, B. Mihaljević, A. Radovan, Assessing the Efficiency of Java Virtual Threads in Database-Driven Server Applications, In 2024 47th MIPRO ICT and Electronics Convention (MIPRO) IEEE (2024) 2045–2050, https://doi.org/10.1109/MIPRO60963.2024.10569754.

[9] S. Iwanowski, G. Kozieł, Comparative analysis of reactive and imperative approach in Java web application development, Journal of Computer Sciences Institute 24 (2022) 242–249, https://doi.org/10.35784/jcsi.2999.

[10] K. Mochniej, M. Badurowicz, Performance comparison of microservices written using reactive and imperative appr oaches, Journal of Computer Sciences Institute 28 (2023) 242–247, https://doi.org/10.35784/jcsi.3698.

[11] A. Navarro, J. Ponge, F. Le Mouël, C. Escoffier, Considerations for integrating virtual threads in a Java framework: A Quarkus example in a resource-constrained environment, Proceedings of the 17th ACM International Conference on Distributed and Event-Based Systems (2023) 103-114,https://doi.org/10.1145/3583678.3596895

[12] K. Lis, J. Smołka, Examination of the performance and scalability of a web application in a reactive and imperative approach using the Spring Framework, Journal of Computer Sciences Institute 32 (2024) 210–216, https://doi.org/10.35784/jcsi.6291.