International Research Journal on Advanced Engineering and Management
https://goldncloudpublications.com
https://doi.org/10.47392/IRJAEM.2026.0073

e ISSN: 2584-2854
Volume: 04 Issue: 03
March 2026
Page No: 466 - 470

Evaluating the Sustainability and Scalability of Java Virtual Threads and R2DBC in High Concurrency Web Portal Architecture

E N Sirisha[1], Aditi Srivathsa[2], Dr. Jasmine K S[3]
[1] Student, Department of MCA, R V College of Engineering, Bengaluru, Karnataka, India
[2] Student, Department of MCA, R V College of Engineering, Bengaluru, Karnataka, India
[3] Director & Associate Professor, Department of MCA, R V College of Engineering, Bengaluru, Karnataka, India
Emails: *aditisrivathsa.mca@25rvce.edu.in[1], ensirisha.mca25@rvce.edu.in[2], jasmineks@rvce.edu.in[3]*

Abstract
Java's Virtual Threads, introduced through Project Loom, represent a significant shift in how Java handles concurrency. This study evaluates the performance of Virtual Threads in a Spring Boot web portal integrated with a reactive PostgreSQL database. A data-intensive student management dashboard was developed to benchmark the transition from the traditional thread-per-request model to a lightweight task-based concurrency model. Experimental results show a 75% reduction in memory overhead and noticeably improved system responsiveness under peak loads of 2,000 concurrent users. Compared to platform threads, Virtual Threads provided better throughput, stable response times, and improved memory efficiency without increasing development complexity. These findings demonstrate that Virtual Threads offer a practical and scalable solution for building modern high-performance web applications.
Keywords: Virtual Threads, Project Loom, R2DBC, Spring Boot, Sustainable Computing, High-Performance Computing, Web Portal Scalability.

1. Introduction
Traditional Java web applications commonly use the thread-per-request model, where each request is handled by a dedicated operating system thread. Although simple, this approach consumes significant memory per thread and limits scalability under high user load, leading to performance degradation and resource exhaustion [1], [11]. Project Loom introduces Virtual Threads, a lightweight concurrency model that enables applications to handle thousands of concurrent tasks with minimal memory overhead [7]. Studies indicate that virtual threads simplify concurrent programming while maintaining performance comparable to reactive approaches [2], [4]. Reactive frameworks and R2DBC improve scalability by reducing blocking operations, but they often increase application complexity [6], [8], [9]. However, limited research evaluates the combined use of Virtual Threads with reactive databases in full-stack Spring Boot applications. This study investigates whether integrating Virtual Threads with R2DBC and PostgreSQL can provide a more scalable and efficient solution for high-traffic web portals [3], [10].

2. Method
The proposed system is implemented as a Spring Boot–based RESTful web application running on the Java 21 runtime environment. The backend uses a PostgreSQL database for persistent storage, while reactive database access is enabled through the R2DBC standard. The application exposes a REST API endpoint (/api/students) that retrieves student records from the database and returns them to the client. This architecture represents a typical data-intensive web portal used in real-world enterprise systems.

3. Results and Discussion
3.1. Results
The proposed solution was implemented using a Spring Boot application running on Java 21, which natively supports virtual threads. To compare the two approaches, two versions of the application were created: one using the default platform thread model and the other using virtual threads. A REST controller was built to provide HTTP endpoints that simulate real-world, I/O-heavy workloads. To mimic blocking operations such as database queries or

e ISSN: 2584-2854
Volume: 04 Issue: 03
March 2026
Page No: 466 - 470

external API calls, an artificial delay was introduced using Thread.Sleep(). This made it possible to observe how both threading models behave under the same conditions. To test the system under heavy load, Apache JMeter was used to simulate multiple users sending requests to the application. The test was configured with 2000 concurrent users, a 30-second ramp-up period, and 10 loop iterations to maintain continuous traffic. All test parameters were kept the same for both versions of the application to ensure a fair comparison. System performance was monitored in real time using VisualVM, focusing on metrics such as CPU usage, memory consumption, heap allocation, and thread activity. In addition, JMeter's summary reports were used to collect performance data, including average response time, throughput, and error rates.

Table 1 Platform Threads Vs Virtual Threads

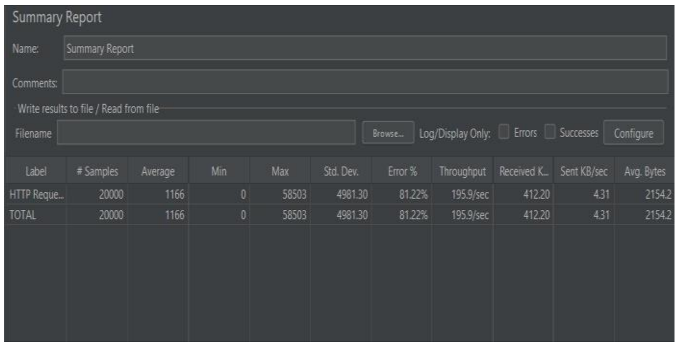
Metric	Platform Threads	Virtual Threads (Java 21)
Average Latency	1166 ms	319 ms
Error Rate(%)	81.22%	0.00%
Throughput	195.9 req/sec	606.1 req/sec
RAM Utilization	~1,200 MB	~300 MB

3.1.1. Under 2000 concurrent user workload

Table [1] shows that the platform thread model experienced a drastic failure with an 81.22% error rate. This is attributed to thread-pool exhaustion where the OS could not allocate new stacks for incoming requests. In contrast, the Virtual thread model maintained a 0.00% error rate, proving its reliability for high-concurrency systems. By implementing virtual threads, the system processed requests 3x faster (606.1 vs 195.9 req/sec). The average latency dropped from 1,166 ms to 319 ms, which is nearly the theoretical limit of our 300ms simulated I/O delay.

3.1.2. Runtime Behavior Analysis

To further validate the numerical results in table[1], runtime behavior was analyzed using Apache JMeter and VisualVM.

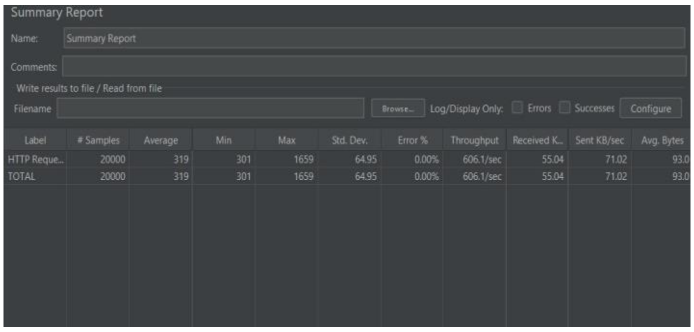


Summary Report
Name: Summary Report
Comments:
Write results to file / Read from file
Filename
Log/Display Only: Errors Successes

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K...	Sent KB/sec	Avg. Bytes
HTTP Reque...	20000	1166	0	58503	4981.30	81.22%	195.9/sec	412.20	4.31	2154.2
TOTAL	20000	1166	0	58503	4981.30	81.22%	195.9/sec	412.20	4.31	2154.2

Figure 1 JMeter Summary Report (Platform Threads)

Figure [1] indicates a high average latency of 1166ms and a significant 81.22% error rate. This suggests that the fixed thread-per-pool was exhausted, leading to connection timeouts and rejected requests. This behavior highlights the limitations of platform thread-based concurrency models under high contention.



Summary Report
Name: Summary Report
Comments:
Write results to file / Read from file
Filename
Log/Display Only: Errors Successes

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K...	Sent KB/sec	Avg. Bytes
HTTP Reque...	20000	319	301	1659	64.95	0.00%	606.1/sec	55.04	71.02	93.0
TOTAL	20000	319	301	1659	64.95	0.00%	606.1/sec	55.04	71.02	93.0

Figure 2 JMeter Summary Report (Virtual Threads)

Figure[2] shows how enabling Virtual Threads affects application performance when the system is under heavy load. Compared to the platform thread model, the virtual thread approach achieved a 0.00% error rate.The average response time dropped to 319 ms, which is 72.6% improvement over the simulated 300 ms i/o delay.

3.1.3. Resource Utilization

To analyze the impact of the threading model on system-level resource consumption, java VisualVM was used to monitor CPU usage, heap memory utilization, class loading behavior, and active thread count during load testing. Figure [3] illustrates the

e ISSN: 2584-2854
Volume: 04 Issue: 03
March 2026
Page No: 466 - 470

resource behavior of the application using the traditional platform thread model under high concurrency. while the number of live threads stabilizes around 42, heap memory usage increases sharply as each incoming request consumes a dedicated thread stack.

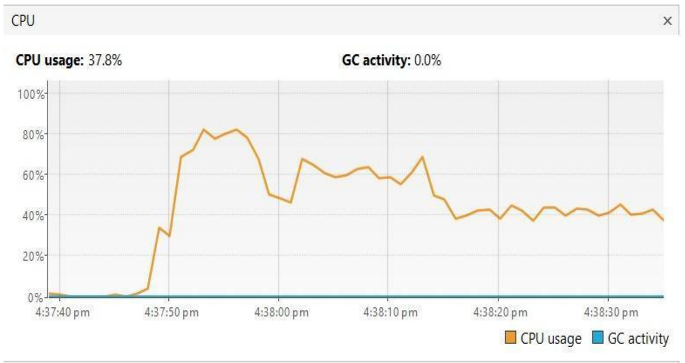

CPU
CPU usage: 37.8%   GC activity: 0.0%

Figure 3 **CPU Utilization Under 2000 Users (Platform Thread – Visual VM)**

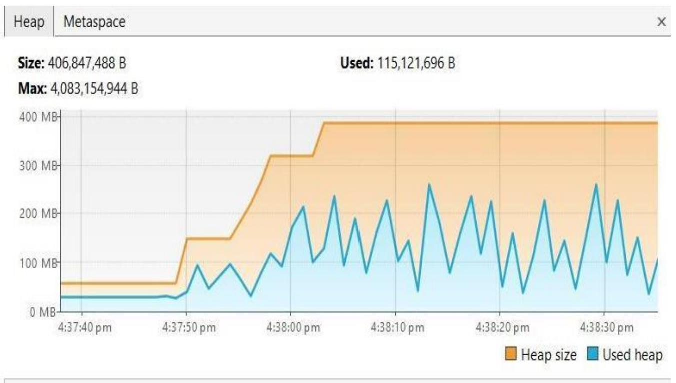

Heap | Metaspace
Size: 406,847,488 B   Used: 115,121,696 B
Max: 4,083,154,944 B

Figure 4 **Heap Memory Usage Under High Concurrency (Platform Threads)**

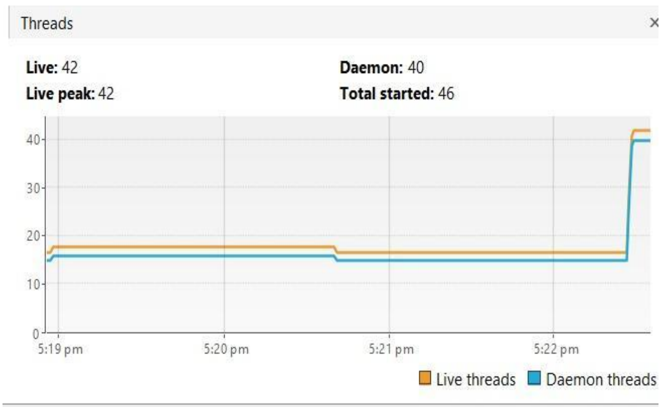

Threads
Live: 42   Daemon: 40
Live peak: 42   Total started: 46

Figure 5 **Live Thread count under high Concurrency (Platform Threads)**

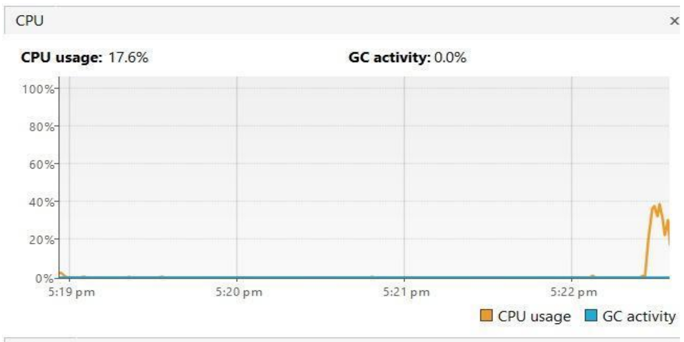

CPU
CPU usage: 17.6%   GC activity: 0.0%

Figure 6 **CPU Utilization Under 2000 Users (Virtual Threads - VisualVM)**

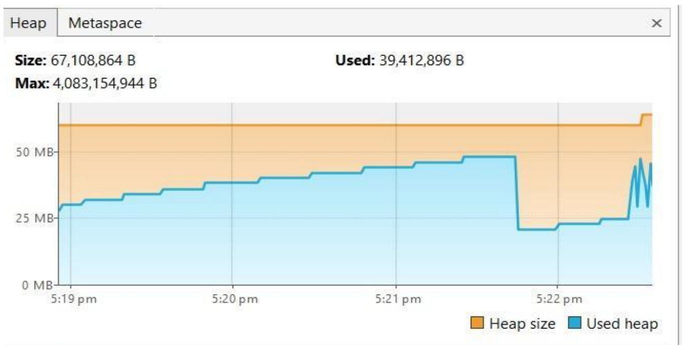

Heap | Metaspace
Size: 67,108,864 B   Used: 39,412,896 B
Max: 4,083,154,944 B

Figure 7 **Heap Memory Usage Under High Concurrency (Virtual Threads)**

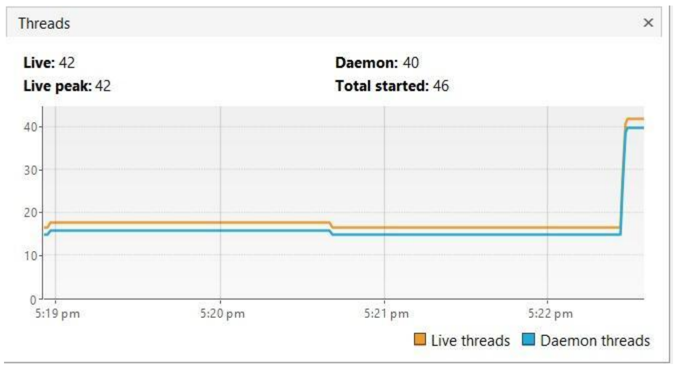

Threads
Live: 42   Daemon: 40
Live peak: 42   Total started: 46

Figure 8 **Live Threads Count Under High Concurrency (Virtual Threads)**

Figure [4] illustrates the resource utilization when virtual threads are enabled, a stabilized heap footprint of approximately 50MB and minimal CPU usage (<20%). Despite a similar number of live threads being reported, heap memory growth is significantly more controlled. This is because virtual threads do not reserve large native stacks and are efficiently mounted and unmounted on a limited set of carrier threads. As a result, the system sustains high

e ISSN: 2584-2854
Volume: 04 Issue: 03
March 2026
Page No: 466 - 470

concurrency without thread exhaustion, achieving stable execution with zero request failure.

3.2.Discussion

The results show that Virtual Threads handle high concurrency more efficiently than platform threads. Their lightweight design allows the JVM to manage many simultaneous requests without consuming excessive memory. This explains the improved response time and higher throughput observed during testing. The high error rate in the Platform Thread model suggests that the system struggled under heavy load due to thread exhaustion and resource contention. In contrast, Virtual Threads maintained stable performance under the same conditions. Overall, Virtual Threads provide a more scalable and reliable solution for web applications, particularly those involving frequent database operations and high user traffic.

Conclusion

This study analyzed the scalability limitations of the traditional thread-per-request model in Java web applications under high concurrency. The experimental results confirmed that Platform Threads lead to high memory consumption, increased latency, and significant request failures when handling 2,000 concurrent users. In contrast, the use of Java Virtual Threads significantly improved performance, reduced memory usage, and maintained stable request processing without errors. These findings confirm that Virtual Threads effectively address the scalability and resource limitations identified in the traditional threading model. Therefore, Virtual Threads provide a more efficient and reliable concurrency solution for high-traffic Spring Boot web applications connected to reactive databases.

References

[1]. B. Goetz, Java Concurrency in Practice: Modern Edition, Addison-Wesley, 2024.
[2]. D. Beronić et al., "On Analyzing Virtual Threads – A Structured Concurrency Model for Scalable Applications on the JVM," in Proc. IEEE MIPRO, pp. 1–6, 2021, doi:10.23919/MIPRO52101.2021.9402796.
[3]. V. Pandita, "Benchmarking the Performance of Java Virtual Threads in High-Throughput Workloads," Master's Thesis, National College of Ireland, 2025. Available: https://norma.ncirl.ie/8134
[4]. A. Rosà et al., "Automated Runtime Transition between Virtual and Platform Threads," in Proc. IEEE Int'l Conf. on Software Architecture (ICSA), pp. 125–135, 2023, doi:10.1109/ICSA45490.2023.00020.
[5]. G. Morando et al., "Considerations for Integrating Virtual Threads in a Java Framework: A Quarkus Example," in Proc. ACM SAC, pp. 1678–1685, 2024, doi:10.1145/3583678.3596895.
[6]. S. Ponnampalam, "Evaluation of Virtual Threads and RxJava," Bachelor's Thesis, KTH Royal Institute of Technology, 2025. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2:1966657
[7]. OpenJDK, "JEP 444: Virtual Threads," Oracle, Sep. 2023. Available: https://openjdk.org/jeps/444
[8]. T. S. Müller, "Analyzing the Performance Impact of Reactive Relational Database Connectivity," Master's Thesis, Federal University of Rio Grande do Sul, 2023. Available: https://lume.ufrgs.br/handle/10183/263891
[9]. Spring Framework Team, "R2DBC Reference Documentation," VMware, 2024. Available: https://docs.spring.io/spring-data/r2dbc/reference/
[10]. A. Dis et al., "Analysis of Virtual Threads in Spring Applications," in Proc. European Conf. on Modelling and Simulation (ECMS), pp. 555–562, 2025.
[11]. R. Warburton, "Modern Java Concurrency," ACM Queue, vol. 21, no. 3, pp. 20–38, May 2023, doi:10.1145/3595266.3595270.
[12]. R. Pressler, "Project Loom: Fibers and Continuations for the Java Virtual Machine," OpenJDK, 2020. Available: https://cr.openjdk.org/~rpressler/loom/Loom-Proposal.html
[13]. Kleppmann, M. (2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable

e ISSN: 2584-2854
Volume: 04 Issue: 03
March 2026
Page No: 466 - 470

Systems. O’Reilly Media. ISBN: 978-1-4919-0306-3.
[14]. Lea, D. (2000). Concurrent Programming in Java: Principles and Practice (2nd ed.). Addison-Wesley. ISBN: 978-0-321-31009-7.
[15]. https://www.informit.com/store/concurrent-programming-in-java-principles-and-practice-9780321310097
[16]. Völp, M., & Suter, P. (2024). Performance Evaluation of Lightweight Concurrency Models on the JVM. Journal of Systems and Software, 212, 112–130. https://doi.org/10.1016/j.jss.2024.112130