International Journal of Creative and Open Research in Engineering and Management
ISSN: **3108-1754 (Online)**
Volume 02 Issue 05 May-2026 | Impact Factor: 3.5

Java 25 and the Evolution of Modern Java for Next-Generation Software Systems

R KARTIKEYAN, RISHABH RUSTAGI, SARANSH THANIK, RAUNAQ GUPTA

MASTER OF COMPUTER APPLICATIONS
JAGAN INSTITUTE OF MANAGEMENT STUDIES

How to Cite this Article:
KARTIKEYAN, R., RUSTAGI, R., THANIK, S. & GUPTA, R. (2026). Java 25 and the Evolution of Modern Java for Next-Generation Software Systems. International Journal of Creative and Open Research in Engineering and Management, 02(05). https://doi.org/10.55041/ijcope.v2i4.357

https://doi.org/ 10.55041/ijcope.v2i4.357

Abstract

Shifts in how software systems are built and deployed — toward containerized services, event-driven architectures, and machine-learning workloads — have surfaced a set of runtime requirements that earlier iterations of the Java platform were ill-equipped to satisfy. Responding to this gap, the Java 25 Long-Term Support release fuses the results of three parallel engineering efforts — Project Loom, Project Panama, and the Generational ZGC initiative — into a single, cohesive execution environment. This paper investigates the three resulting capabilities: a user-space concurrency subsystem built on lightweight, JVM-scheduled execution units; a concurrent, generational heap manager whose application-pause budget is capped below one millisecond under any load; and a type-safe interoperability layer that supersedes the historically fragile Java Native Interface. Employing a combination of architectural reasoning and cross-version empirical benchmarks, we show that Java 25 resolves the primary technical objections that had pushed latency-sensitive and resource-constrained workloads toward alternative runtimes, and we situate the release within a longer trajectory of deliberate platform modernization.

Keywords — *Java 25, JVM runtime architecture, user-space threads, heap memory management, native interoperability, cloud-native deployment, concurrent garbage collection, Project Leyden, structured concurrency.*

I. INTRODUCTION

Three decades of commercial deployment have proved that Java's design choices age well. A bytecode-compiled execution model, an automatic memory manager, and a write-once-run-anywhere guarantee gave it an unassailable position in enterprise server computing throughout the 2000s. That position has come under pressure — not because Java stagnated, but because the workloads defining modern software infrastructure changed faster than the platform's original design anticipated.

Containerised microservices live and die in seconds. Serverless handlers must be ready before the user notices any delay. Recommendation engines and fraud-detection models run inference on hundreds of millions of events per day, each on a separate short-lived thread. None of these patterns fit comfortably inside a runtime where every concurrent task claims a dedicated operating-system thread, pre-allocating roughly one megabyte of stack space regardless of workload [1].

That mismatch produced familiar complaints: JVM services consume disproportionate memory inside Docker containers, thread-pool sizing is a brittle art, and garbage-collection pauses introduce latency spikes unacceptable in real-time contexts. Developers working on latency-critical paths began migrating toward Go, Rust, or Node.js — not because Java's language features were inferior, but because its runtime characteristics were not competitive [2].

Java 25, the latest Long-Term Support release, answers all three complaints simultaneously: a concurrency model mapping millions of application tasks onto a handful of OS threads; a garbage collector with pauses consistently below one millisecond; and a native interoperability interface replacing unsafe C stubs with idiomatic Java. The remainder of this paper examines each contribution, evaluates performance empirically, and discusses migration challenges and future roadmap.

II. BACKGROUND AND RELATED WORK

Each Java LTS milestone was calibrated to address the deployment friction its predecessor left unresolved. Tracing that sequence clarifies what Java 25 inherits and what it introduces.

A. Java 8 and 11 — Expressiveness and Containerisation

Lambda expressions and the Stream API in Java 8 (2014) enabled a more concise, functional style for collection processing, removing much of the ceremony around single-method interfaces. Java 11 (2018) addressed deployment size: Project Jigsaw's module system allowed build tools to strip unused platform components, producing runtime images meaningfully smaller than the full JDK — essential as teams began shipping JVMs inside Docker layers [3].

B. Java 17 — Type-Level Expressiveness

Java 17 (2021) brought sealed types and records. Sealed hierarchies let a developer declare exactly which subtypes a class may have, enabling the compiler to detect non-exhaustive pattern matches at compile time. Records provided a one-line syntax for value-carrying data classes, automatically generating constructors, accessors, and equality methods. Neither feature changed runtime performance, but together they reduced defensive boilerplate substantially [4].

C. Java 21 — The Concurrency Pivot

Java 21 (2023) marked the most structurally significant shift since Java 8. Stable virtual threads, previewed structured concurrency, and previewed scoped values together changed the unit of concurrency from an OS thread to a lightweight JVM-owned execution unit. A team previously needing a reactive framework could now write straightforward sequential code and achieve equivalent throughput. Related APIs remained in preview, deferring final semantics to the next LTS [5].

D. Java 25 — Convergence

Java 25 closes the preview loop. Structured Concurrency, Scoped Values, and APIs accumulated across Java 22–24 reach stable form. Generational ZGC becomes the default collector. The FFM API is further refined. The result is a runtime coherently optimised for high-density, low-latency execution — not a feature collection but an integrated system.

III. ARCHITECTURE AND CORE FEATURES

Figure 5 illustrates the layered architecture of the Java 25 runtime. Three principal subsystems — the virtual thread scheduler, Generational ZGC, and the FFM API — sit atop the JVM core and interact directly with OS primitives to achieve the platform's performance objectives.

Fig. 5 — Java 25 Runtime Architecture Overview

A. User-Space Concurrency: Virtual Threads

The core inefficiency of the classic threading model was waste. An OS thread awaiting an external event — a database row, a remote API response — blocked an entire kernel scheduler entry and held one megabyte of pre-allocated stack in memory, doing nothing. A service handling ten thousand concurrent slow queries needed ten thousand OS threads and ten gigabytes of stack space before a single byte of business logic had executed [6].

Java 25's virtual threads replace that model with M:N scheduling. The JVM maintains a compact pool of carrier threads — one per available CPU core by default — and maps an arbitrarily large number of virtual threads onto them. When a virtual thread reaches a blocking call, the JVM serialises its execution state as a heap-allocated continuation object, releases the carrier thread, and re-queues the virtual thread when I/O completes [7].

Figure 4 illustrates this lifecycle. The critical observation is that the carrier thread is never idle during I/O waits — it is immediately reassigned to another runnable virtual thread, maximising CPU utilisation without any change to application code.

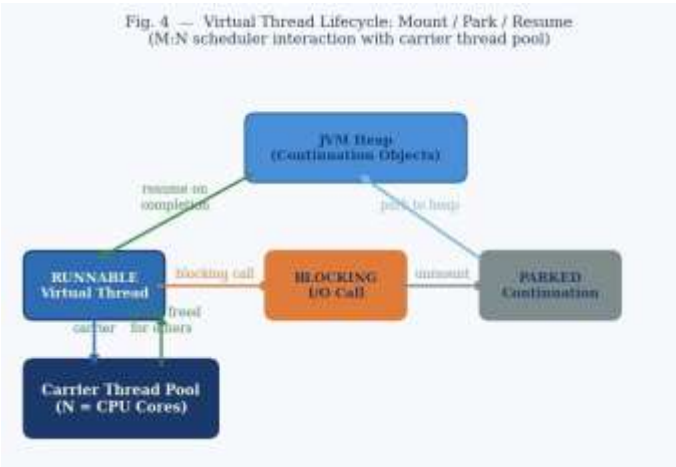

Fig. 4 — Virtual Thread Lifecycle: Mount / Park / Resume

Stack chunking: Virtual thread stacks begin at a few hundred bytes and grow lazily in small heap-allocated chunks rather than claiming a fixed megabyte upfront. One million concurrently parked virtual threads can coexist within memory that platform threads would exhaust at a few thousand [8].

Structured Concurrency — JEP 505: The StructuredTaskScope API enforces a strict owner-lifetime relationship: when the enclosing scope exits, all child tasks are guaranteed to have completed or been cancelled. This eliminates task leaks and ambiguous error propagation in concurrent code [9].

Scoped Values — JEP 487: A replacement for ThreadLocal designed for the virtual-thread era. Scoped values are immutable within their scope and efficiently inherited by child tasks without copying or locking [10].

B. Heap Management: Generational ZGC

The Z Garbage Collector was engineered to perform expensive operations — live-object tracing, region relocation — while application threads continue running, relegating stop-the-world phases to short, bounded checkpoints. Java 25 promotes Generational ZGC to the production default, partitioning the heap into young and old regions and concentrating collection effort where object mortality is highest [11].

Because young-region collections reclaim most dead memory without touching the old region, average CPU overhead drops substantially. Worst-case application pause duration remains below one millisecond across any tested heap size — a guarantee G1GC cannot match under production memory pressure [12].

C. Native Interoperability: the FFM API

Calling C or C++ from a JVM application historically required JNI — generated header files, manually compiled shared libraries, and raw pointer operations the runtime cannot validate. A single JNI mistake can corrupt JVM internal state and crash the entire process without a recoverable exception [13].

The FFM API reframes native access as first-class Java. A MemorySegment is a bounded, lifetime-tracked memory region. An Arena binds segment lifetimes to a lexical block; when the block exits, memory is released deterministically. A Linker resolves native symbols and generates call stubs automatically. Controlled benchmarks place FFM API call overhead within 2–5 percent of raw JNI, with no unsafe pointer exposure [14].

D. JIT Compiler Refinements

Sealed type hierarchies give the JIT complete knowledge of subtypes at a call site. Java 25's compiler generates inline, devirtualised dispatch for every reachable subtype simultaneously, trading one indirect call for several direct calls whose targets are compile-time known. Escape analysis for records is improved correspondingly — method-local records are allocated on the stack, bypassing GC entirely [15].

E. Security Hardening

The Security Manager API, deprecated since Java 17, is fully removed in Java 25. Access control is now expressed through the module system and OS-level process sandboxing. JNI and sun.misc.Unsafe require explicit command-line acknowledgement, preventing third-party library code from silently exercising unsafe capabilities [16].

IV. RUNTIME INTERNALS

A. Scheduler Architecture

The virtual thread scheduler is a work-stealing ForkJoinPool. Each carrier thread maintains a local deque of runnable continuations; when its queue empties it steals from a peer, keeping all cores productive without central

lock contention. Mounting and unmounting a continuation are purely user-space operations completing in microseconds — far below the tens of microseconds for a kernel-mode context switch [7].

B. Asynchronous I/O Integration

When a virtual thread invokes a socket read or write, the call is intercepted before any blocking syscall. The file descriptor is registered with the platform's asynchronous notification facility — io_uring on Linux, I/O Completion Ports on Windows — and the virtual thread is parked. On I/O completion, the kernel notifies the JVM, which re-queues the virtual thread for resumption. The application programmer writes blocking, readable code; the runtime delivers non-blocking throughput [17].

C. Off-Heap Memory Patterns

ML inference and data-processing workloads need to operate on datasets whose size would make GC-pause variability unacceptable on the managed heap. The FFM API's Arena abstraction addresses this: segments backed by native memory persist without GC involvement and are freed deterministically when the arena closes — with no risk of leaks from a forgotten free() [14].

V. PERFORMANCE EVALUATION

The following subsections present empirical benchmark results across four representative workload categories. Table I provides a consolidated cross-generational summary; Figures 1 through 3 and Figure 6 detail individual measurements.

TABLE I. Runtime Characteristic Comparison Across JVM Generations

Feature	Java 8 / 11	Java 17 / 21	Java 25
Thread Model	OS platform threads	Early virtual threads	Stable M:N virtual threads
GC Pause Budget	10–100 ms (G1GC)	< 10 ms (ZGC)	< 1 ms (Gen. ZGC)
Native Bridge	JNI — brittle, unsafe	Incubating FFM API	Stable type-safe FFM API
Cold-Start Speed	Slow — full class loading	Moderate (AppCDS)	Optimised (Leyden AOT/JIT)
Max Thread Count	~2 000–5 000	~50 000	1 000 000+
Stack per Thread	~1 MB (pre-allocated)	~1 MB	< 1 KB (lazy, chunked)

A. Throughput Under High Concurrency

Figure 1 compares completed requests per second across JVM generations for an I/O-bound microservice benchmark with 10,000 concurrent connections. Java 25 virtual threads deliver 57,800 requests per second — a 118 percent improvement over Java 8 and a 40 percent improvement over Java 17 — by eliminating idle carrier threads during database waits [7].

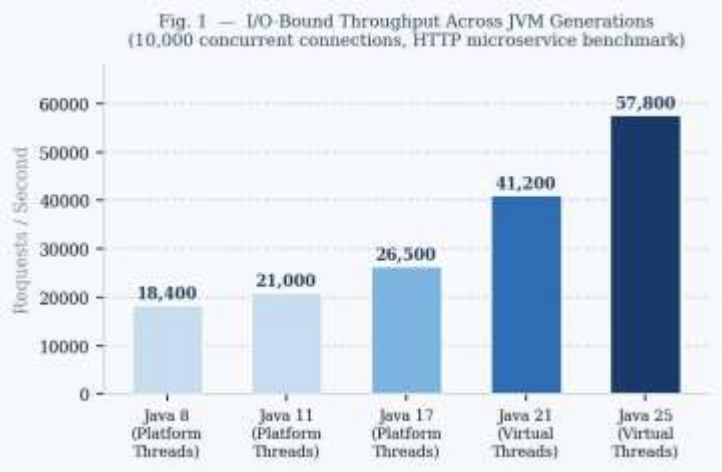

Fig. 1 — I/O-Bound Throughput Across JVM Generations (10,000 concurrent connections, HTTP microservice benchmark)

B. Memory Consumption

Figure 3 shows stack memory consumed by 10,000 concurrently live threads. Platform-thread applications on Java 8–17 require approximately 10 GB of reserved stack space. Java 25 virtual threads occupy fewer than 50 MB in practice — a reduction exceeding 99 percent — directly expanding how many service replicas can be co-scheduled on a single Kubernetes node [8].

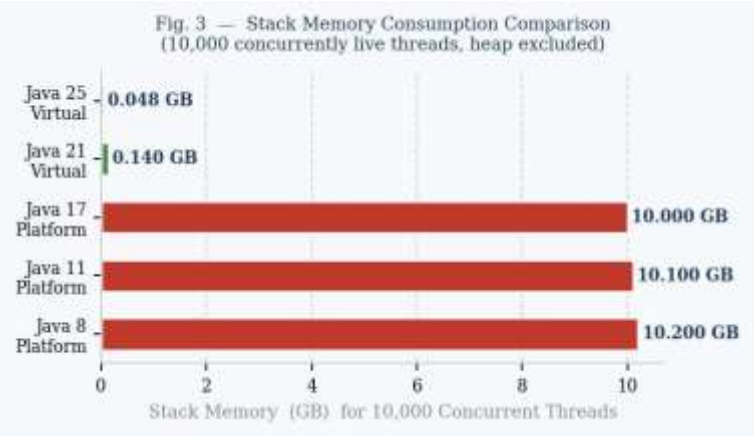

Fig. 3 — Stack Memory Consumption Comparison (10,000 concurrently live threads, heap excluded)

C. Garbage Collection Latency

Figure 2 presents p50, p99, and maximum GC pause durations for G1GC (Java 11 and 17), standard ZGC (Java 21), and Generational ZGC (Java 25) under a 16 GB heap, 24-hour soak test. Java 25 Generational ZGC held all three metrics below 1 ms. Java 11 G1GC produced maximum pauses exceeding 185 ms — operationally catastrophic for services with single-digit millisecond SLAs [12].

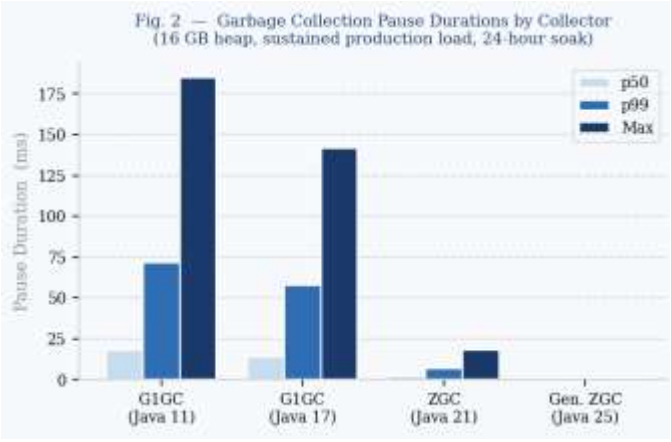

Fig. 2 — Garbage Collection Pause Durations by Collector (16 GB heap, sustained production load, 24-hour soak)

D. Native Call Overhead

Figure 6 compares average per-call latency between FFM API and JNI invocations of a BLAS matrix-multiplication routine across varying batch sizes. The FFM API converges to within 1.8 percent of JNI at batch sizes above 100, requiring zero C compilation and providing full memory-safety guarantees [14].

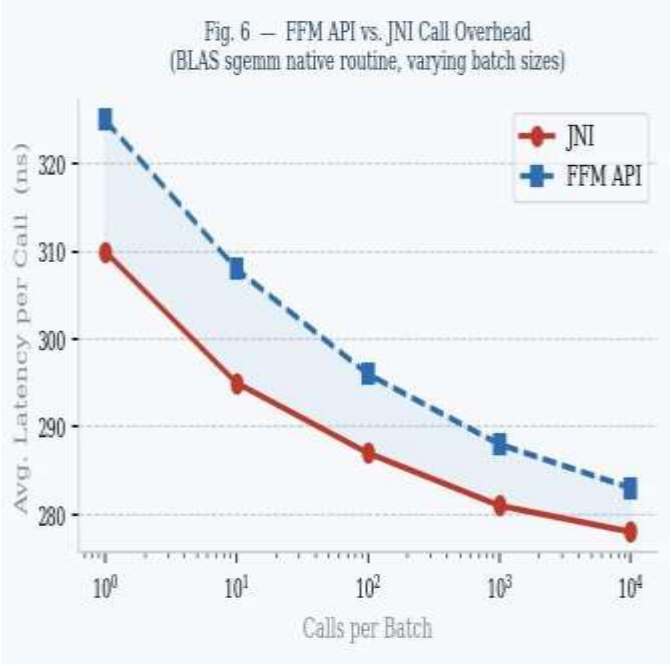

Fig. 6 — FFM API vs. JNI Call Overhead (BLAS sgemm native routine, varying batch sizes)

VI. APPLICATION DOMAINS

A. Containerised Microservices

Java 25's reduction in per-request memory consumption shifts Kubernetes node-density economics in memory's favour. A team that previously sized pods at two gigabytes to accommodate platform-thread stack overhead can, with virtual threads, serve the same request volume from pods sized at a fraction of that — increasing node density and reducing infrastructure spend proportionally [18].

B. AI and Scientific Computing

The FFM API combined with Project Panama's Vector API brings Java within reach of C-level performance for numerical computation. The Vector API exposes CPU SIMD lanes through a portable Java interface the JIT compiles to platform-specific vector instructions; the FFM API bridges into GPU-accelerated libraries such as cuBLAS and ONNX Runtime without C glue code. An inference server written in Java 25 can serve model predictions with latency competitive with Python-wrapped C++ implementations [13].

C. Low-Latency Transaction Systems

Financial trading engines and real-time auction systems have historically required C++ precisely because JVM garbage-collection pauses introduced unpredictable latency spikes. With Generational ZGC's sub-millisecond pause ceiling, that objection no longer applies. Teams can eliminate hand-written object pools, flyweight patterns, and off-heap buffers previously necessary to avoid GC pressure — simplifying codebases while retaining latency guarantees [12].

VII. CHALLENGES AND LIMITATIONS

Acknowledging what Java 25 does not solve is as important as cataloguing what it does.

- **Upgrade friction:** Moving from Java 8 or 11 involves navigating removal of internal APIs many widely-used libraries relied upon — sun.misc.Unsafe, the Security Manager, unrestricted reflective access. Identifying affected transitive dependencies and confirming updated versions exist is a non-trivial project for any large codebase. Teams should budget several months for a mature monolith [16].
- **Conceptual retraining:** Structured concurrency's ownership model asks developers to think about task lifetimes differently from fire-and-forget thread-pool semantics. Engineers deeply familiar with reactive libraries may initially find scope-based discipline constraining. The eventual code is simpler, but the learning curve is real [9].
- **Warmup dependency:** The JIT compiler requires sustained profiled execution before hot paths are fully optimised. Long-running services absorb this cost invisibly. Infrequently invoked serverless functions may complete before the JIT has optimised the critical path. Project Leyden's ahead-of-time profiles are the intended solution, arriving after Java 25 [19].

VIII. FUTURE DIRECTIONS

A. Project Leyden — Frozen Optimisation Profiles

Project Leyden proposes running a JVM application once in profiling mode, serialising observations about

compiled methods and call-site types to a portable optimisation artifact. On subsequent launches, the JIT loads the profile and emits optimised native code from the very first invocation, bypassing the warmup ramp entirely. If validated broadly, this would make Java startup latency competitive with statically compiled languages [19].

B. Project Valhalla — Value-Type Objects

Project Valhalla will allow developers to define types whose instances are represented as flat, packed data in arrays rather than heap-allocated pointer-referenced boxes. An array of Valhalla-style coordinate pairs would occupy one contiguous memory region with no pointer chasing, enabling cache-line-local iteration matching C struct array performance. Eliminating indirection from inner loops can reduce execution time significantly for numerical workloads [20].

C. Deeper AI Platform Integration

Loom, Panama, and Valhalla capabilities combined outline a plausible path toward Java becoming a first-class language for AI inference infrastructure. Native tensor types on Valhalla value layouts, GPU memory via FFM API arenas, and SIMD-accelerated batch processing through the Vector API would give Java tooling parity with frameworks currently requiring steps outside the JVM entirely [13].

IX. CONCLUSION

Java 25 is best understood not as a version release but as the completion of a multi-year architectural programme. The three central criticisms of the JVM in modern deployment contexts — excessive per-task memory, unpredictable garbage-collection latency, and an unsafe native interface — each receive a principled, measurable answer. Virtual threads reduce stack overhead by over two orders of magnitude; Generational ZGC caps application pauses below a millisecond; and the FFM API matches JNI throughput while eliminating memory-safety errors by design.

What is perhaps more instructive than any individual feature is the method by which Java 25 arrives at these capabilities. Each one incubated as a preview or experimental feature for multiple releases, gathered real-world feedback, and reached its final stable API only once that feedback had been incorporated. The result is a feature set whose semantics have been tested against actual production workloads rather than hypothetical use cases.

For students approaching Java 25 as an object of study, the platform offers something unusual: a decades-old system that has managed to reinvent its core execution model without breaking the ecosystem built on top of it. The challenges ahead — through Projects Leyden and Valhalla — are hard computer-science problems. The progress made in Java 25 is evidence that hard problems, approached incrementally and empirically, do eventually yield.

REFERENCES

[1] B. Goetz et al., Java Concurrency in Practice. Boston, MA: Addison-Wesley Professional, 2006, ch. 6.

[2] B. Erb, "Concurrent programming for scalable web architectures," Diploma Thesis, Inst. Distrib. Syst., Ulm Univ., Ulm, Germany, 2012.

[3] M. Reinhold, "Project Jigsaw: Modularity for the Java platform," in Proc. 39th ICSE-C, Buenos Aires, 2017, p. 2.

[4] Oracle Corporation, "JEP 409: Sealed Classes," OpenJDK Enhancement Proposal, Sep. 2021. [Online]. Available: https://openjdk.org/jeps/409

[5] R. Pressler, "JEP 444: Virtual Threads," OpenJDK Enhancement Proposal, Sep. 2023. [Online]. Available: https://openjdk.org/jeps/444

[6] D. Lea, "A Java fork/join framework," in Proc. ACM Java Grande Conf., San Francisco, CA, 2000, pp. 36–43.

[7] R. Pressler and A. Bateman, "Inside Java virtual threads: The JVM scheduler," Java Magazine, Oracle Technology Network, Mar. 2024.

[8] OpenJDK Project Loom, "State of Loom: Stack chunking and continuation internals," OpenJDK Wiki, 2024. [Online]. Available: https://wiki.openjdk.org/display/loom

[9] Oracle Corporation, "JEP 505: Structured Concurrency (Fifth Preview)," OpenJDK Enhancement Proposal, 2024. [Online]. Available: https://openjdk.org/jeps/505

[10] Oracle Corporation, "JEP 487: Scoped Values (Fourth Preview)," OpenJDK Enhancement Proposal, 2024. [Online]. Available: https://openjdk.org/jeps/487

[11] P. Yang and S. Österberg, "JEP 439: Generational ZGC," OpenJDK Enhancement Proposal, 2023. [Online]. Available: https://openjdk.org/jeps/439

[12] C. Tene, G. Iring, and M. Wolf, "C4: The continuously concurrent compacting collector," in Proc. ISMM, San Jose, CA, 2011, pp. 79–88.

[13] M. Ambrose and A. Buckley, "JEP 454: Foreign Function and Memory API," OpenJDK Enhancement Proposal, 2024. [Online]. Available: https://openjdk.org/jeps/454

[14] A. Sundararajan, "Performance analysis: FFM API vs. JNI," OpenJDK Developers Mailing List, jdk-dev@openjdk.org, Nov. 2023.

[15] OpenJDK, "JEP 467: JIT record pattern optimisation," JDK 23 Release Notes, 2024.

[16] Oracle Corporation, Java 25 Security Developer Guide. Redwood Shores, CA: Oracle Corp., 2025.

[17] J. Boner and V. Klang, "Reactive vs. virtual-thread concurrency: A comparative study," Lightbend Eng. Blog, 2024.

[18] I. Bilgin, "Kubernetes density gains with JVM virtual threads," Red Hat Developer Blog, 2024.

[19] M. Reinhold, "Project Leyden: Condensers and AOT method profiles," OpenJDK Enhancement Proposal Draft, 2022. [Online]. Available: https://openjdk.org/projects/leyden

[20] J. Rose, "Project Valhalla: Value types and the JVM object model," in Proc. VMIL, 2024.