Future Technology
Open Access Journal
ISSN 2832-0379

May 2026| Volume 05 | Issue 02 | Pages 297-309

Journal homepage: https://fupubco.com/futech

https://doi.org/10.55670/fpll.futech.5.2.27

Future Publishing LLC

Article

Secure migration patterns from Java 8 to Java 17 in the mission-critical ecosystem: a risk-driven approach to modernization

Sravan Reddy Kathi[1]*, Parth Joshi[2], Vani Muralidhar[3], Ajay Venugopalan[4]

[1]Independent Researcher, Bridgeport, PA, USA
[2]Independent Researcher, Dallas, TX, USA
[3]Independent Researcher, Odessa, Florida, USA
[4]Independent Researcher, Lutz, Florida, USA

ARTICLE INFO

Article history:
Received 17 November 2025
Received in revised form
15 February 2026
Accepted 26 March 2026

Keywords:
Java migration, Java 17, Modernization, Risk-driven framework, Backward compatibility, Enterprise Java

*Corresponding author
Email address:
sravanreddykathi55@gmail.com

DOI: 10.55670/fpll.futech.5.2.27

ABSTRACT

Upgrading from Java 8 to Java 17 in the enterprise setting is both challenging and an opportunity, especially for mission-critical ecosystems. Java 17 is a long-term support (LTS) version that includes numerous enhancements over Java 8, including language syntax enhancements, improved garbage collection and memory management, enhanced security models, and modularization via the Java Platform Module System (JPMS). Nevertheless, the opportunities presented by Java 17 can be fully realised with the assistance of a well-organised migration plan that assesses risks, implements mitigation strategies, and ensures the equipping of enterprise-scale systems. This paper proposes a risk-based migration framework that is SAP-based, Java environment-specific, and identifies safe migration patterns, along with a detailed case study example to demonstrate how the workflow migration methodology works. We suggest an expedient and replicable solution that accommodates modernization.

1. Introduction

Legacy enterprise application modernization has emerged as a strategic imperative due to the mounting requirements of improved security, maintainability, and runtime performance. Although Java 8 is still prevalent in the enterprise space, it is no longer supported by the active community and lacks the architectural and security improvements introduced by subsequent long-term support (LTS) releases [1]. Java 17 brings major improvements, including records, sealed classes, pattern matching, improved garbage collectors, stronger encapsulation tools, and improved security defaults [2]. Although these are important reasons to migrate, the migration is also technically challenging in mission-critical environments. The problem becomes much more complicated in the context of SAP-integrated enterprise environments. In these environments, Java applications are deeply integrated with ABAP components, SAML identity providers, SAP JCo connectors, SAP HANA drivers, and legacy third-party libraries. In these scenarios, migration is not just a matter of language-level refactoring but also involves complex considerations of ecosystem dependencies, authentication mechanisms, cryptographic settings, messaging protocols, and runtime compatibility constraints. Incremental upgrades in these environments are fraught with major operational and security risks, especially when business-critical workflows cannot be interrupted [3]. Existing approaches to Java migration primarily focus on modularization techniques, runtime compatibility, or adoption patterns for specific features [4]. Nevertheless, the current state-of-the-art literature still doesn't provide adequate support for risk quantification, exposure of dependency propagation, and operational continuity constraints for SAP-integrated mission-critical systems. Specifically, migration tools and techniques have not yet leveraged structured vulnerability analysis, authentication surface reduction, fallback-driven CI/CD pipelines, or multi-version coexistence techniques. In this context, this paper presents a risk-driven migration framework for migrating enterprise-level Java 8 applications to Java 17 in SAP-constrained mission-critical systems. Unlike the traditional upgrade process, which mainly focuses on version compatibility, our migration framework prioritizes modernization activities according to quantified security risk exposure, transitive dependency risk, ecosystem

compatibility constraints, and runtime stability. Our solution leverages:

- Dependency risk assessment and CVE exposure analysis
- Feature and API compatibility analysis for Java 8, 11, and 17
- Authentication stack restructuring (OpenSAML to Spring Security/JWT)
- Performance analysis in controlled load scenarios
- Fallback-driven CI/CD pipelines for maintaining operational continuity

Validation of the framework is achieved through a practical migration of a high-availability, high-user-volume enterprise system integrated with SAP components. The empirical validation of the framework has shown improvements in runtime performance, memory usage, vulnerability, and maintainability. The paper contributes to the software evolution body of knowledge by proposing a risk-prioritized, empirically validated modernization framework specifically designed for highly integrated enterprise environments. The contribution goes beyond syntax-level migration and provides a practical approach for organizations working in an SAP-centric environment where security, compliance, and continuity are important constraints.

2. Background and related work

2.1 Java Platform Evolution

Java has undergone major platform transformations since Java 9, and the most important aspect is ensuring it is available with high-quality modularity, fun, and improved developer productivity. The simplest structural transformation of such is the addition of the Java Platform Module System (JPMS), which provides a high degree of encapsulation and, equally, a modular architecture at the language and runtime levels. Meanwhile, the sphere of memory management was elevated to a new level with the advent and development of the modern garbage collectors, the most popular of which is G1 as the default collector and ZGC, whose mission is to bring the pause time to a minimum and be able to cope with a large amount of heap [5,6].

Java also contained a few language-level amenities that make the code easier to read and maintain, such as an improved switch expression to more compactly capture logic in a multi-branch form, pattern matching to simplify instance checks, sealed classes to limit and manage inheritance chains, and records to represent immutable transporters of information with less boilerplate [7-10]. The second major Java 9 addition was that the locale data format changed to the Common Locale Data Repository (CLDR), replacing the old compact format from Java 8 [11]. As a standard of internationalisation, upheld by the Unicode Consortium, CLDR ensures that more programs meet international locale standards, but at the expense of behavioural differences, including date, time, and number presentation, collation, and text casing policies across locales. Therefore, migrated legacy applications that leverage the locale-sensitive capabilities of Java 8 might exhibit minor UI differences or defects. All these architectural, runtime, and behavioural changes suggest that not all Java 9+ features can be easily ported to Java 8-based systems, and their use must be approached with a critical eye, rigorous validation, and extensive testing, particularly for mission-critical enterprise applications deployed in highly integrated SAPs.

2.2 SAP Java stack architecture (spring-based)

The current Java applications in the majority of enterprise SAP implementations are not developed with SAP NetWeaver but are based on less rigid, more flexible frameworks, the most prominent of which is the Spring environment. These spring-based applications are generally run in a stand-alone runtime or in containerized environments and communicate with SAP systems via standardized connectors and APIs. Architectural decoupling enables enterprises to upgrade application development without interoperability loss, even when integrated with core SAP platforms. The application comes as a Spring Boot or Spring MVC application, allowing the developer to easily create and design it with conventions that enable configuration of dependency injection and transaction processing, and the exposure of REST-like services. These Spring applications are more likely to have the business logic that is in contact with the SAP back-end system, i.e. either SAP S/4HANA or old ECC systems. This inoperability is facilitated by built-in integrations such as the SAP Java Connector (JCo), OData-based services, and the SAP Cloud SDK, through which Spring services can make scalable and manageable calls to ABAP-based logic [12].

They have frequently been architecturally broken down into microservices that interact with one another via HTTP or HTTPS. It is a microservices architecture that facilitates modularity, independent scaling, and nimbler agile deployment cycles. Meanwhile, enterprise applications are expected to rely on a wide range of third-party libraries and frameworks, some of which may be legacy components or open-source dependencies, and would impose further constraints on the platform's ability to upgrade to support newer versions of the Java runtime. In practice, the majority of businesses use Java applications built with Spring, run in Docker containers, and coordinated and managed with Kubernetes. Examples of such implementations typically run on cloud platforms such as SAP Business Technology Platform (SAP BTP), Amazon Web Services (AWS), or Microsoft Azure [13]. Although this architecture is more adaptable and scalable than traditional SAP-centric runtimes in terms of operational scale, it also introduces new problems. The largest ones include compatibility of the Spring modules with Java 17 (backward compatibility), maintainability and security of third-party dependencies, accuracy, compatibility, and reliability of the integration points with SAP systems.

2.3 Related work

Migration of old Java applications to newer versions of Java has received considerable discussion, even in academic and industry circles. The development of languages, performance differences, and the security of Java versions have been mainly discussed in previous research. As an example, Ref. [14] examines the impact of the Java module system (JPMS) on existing systems, citing insufficiencies in the enforcement of modularity. Pereira et al. also studied the depreciation process and the complexity of replacing eliminated APIs in business processes. Regarding the Java modernization of the SAP ecosystem, there is a lack of formal research. A large percentage of the migration experience was acquired through community forums, SAP Notes, and whitepapers. The SAP project itself (in which the systematic coining of Java stacks to NetWeaver to Java systems is based on the Spring Framework) has given rise to polarization between highly coupled ABAP-Java systems and other, less related microservice systems. Nevertheless, the literature on what such an evolution may imply for modernization practice is limited, particularly in enterprise settings, where SAP HANA and other vendor applications are being introduced.

Regarding the availability of third-party libraries, the studies by Ref. [3,4] highlight the threat of outdated dependencies in Java-developed applications, where the effects of the vulnerability during upgrades are transitive. It, nevertheless, has minimal rich architectures, including ecosystem-based software (e.g. ActiveMQ, Jersey, OpenSAML) and deployment software (e.g. Tomcat, Jenkins). In addition, the fact that the former Compact Number Format has replaced the CLDR in Java 9, and how this could influence internationalisation testing, is very poorly documented. Similarly, the renewal of the latter garbage collection programs (G1GC, ZGC) is not performed individually [1,2], as in the performance-sensitive world of SAP.

3. Methodology: risk-driven migration framework

The migration of enterprise SAP-integrated applications from Java 8 to Java 17 is a high-risk engineering activity because of the mission-critical nature of these applications and their tight coupling with ABAP services and frameworks like Spring, Jersey, and ActiveMQ; hence, as shown in Figure 1, we model this activity as an iterative risk-driven framework consisting of Risk Identification, Risk Classification, Mitigation Implementation, and Validation & Testing, ensuring that the migration is systematically prioritized and governed by exposure to technical risk, rather than being performed as a linear process.

3.1 Risk identification

The Java 8 to Java 17 migration was preceded by a risk identification process that ensured the system's stability, correctness, and maintainability within the SAP-integrated applications ecosystem. Due to the high sensitivity of the platform and its tight coupling with the downstream services, the risk identification process was done as a separate activity before the design of the risk remediation. A thorough dependency analysis was done using the Maven Dependency Plugin and OWASP tools to create a comprehensive list of direct and transitive dependencies [15-17].

The aim was to identify outdated dependencies, exposure to vulnerabilities, and libraries incompatible with the Java 17 runtime environment. Special focus was on legacy or unused libraries and modules that depended on internal JDK APIs that were deprecated or removed in the latest versions of Java. Concurrent with this, a delta analysis was conducted between Java 8 and Java 17 platform characteristics to determine behavioral and compatibility changes. This delta analysis covered the removal or reduction of APIs such as legacy thread control APIs, JVM configuration flags related to permanent generation memory management, and runtime defaults governing behavioral characteristics in newer JDK versions. For instance, locale processing behavior was impacted with the introduction of the Common Locale Data Repository (CLDR) as the default internationalization backend. Integration compatibility tests were conducted across enterprise communication channels, including authentication federation and backend service communication layers. Special focus was given to platform compatibility with SAP HANA database services and identity federation infrastructure, using security libraries such as OpenSAML-based authentication libraries. The scan assessed risks associated with protocol compatibility, serialization characteristics, TLS settings, and library runtime dependencies. The risk discovery procedure has identified over 35 distinct migration risks. Examples of representative risks have been documented in a migration risk register. Some examples of entries in the risk register include the following:

- Exposure of vulnerability to third-party libraries that do not have maintenance updates compatible with Java 17.
- Runtime errors due to the removal of APIs related to old thread lifecycle management.
- Degradation of serialization compatibility in the processing of authentication tokens.
- Failure of the build pipeline due to version incompatibility with the toolchain.

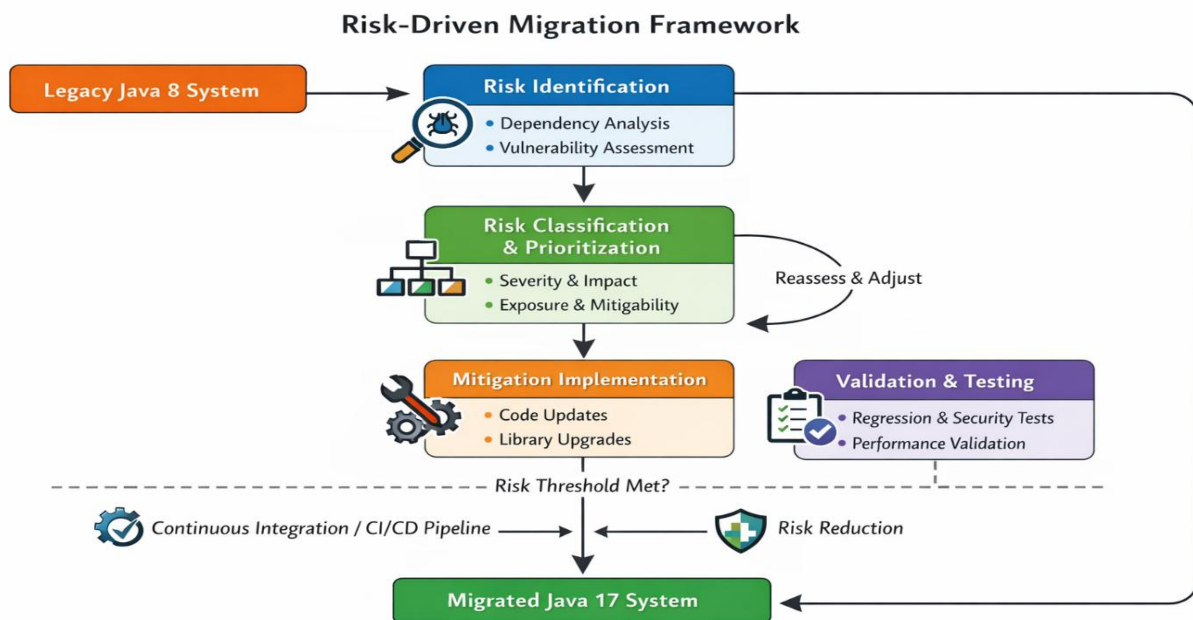

Figure 1. Conceptual risk-driven migration framework for Java 8 to Java 17 modernization in SAP-integrated enterprise systems

The distribution of risk categories is roughly shown in Table 1. The categorization of risks facilitated the prioritization of remediation tasks. High-risk vulnerabilities and compatibility risks were addressed before modifying the platform runtime.

Table 1. Migration risk classification summary for Java 8 to Java 17 transition

Risk Category	Proportion	Description
Dependency Compatibility and Vulnerabilities	~60%	Outdated libraries, insecure cryptographic modules, or APIs removed in Java 17
JVM and Language Runtime Changes	~25%	Garbage collection behavior, module system constraints, and deprecated JVM flags
Toolchain and Platform Integration Constraints	~15%	Build pipeline restrictions, authentication stack compatibility, and database connectivity

3.2 Risk classification

All available risks were systematically categorised in advance to enable analysis and prioritisation during the migration process. The multi-dimensional classification model was used to assess risks in terms of both their technical and practical effects on the enterprise's work, rather than treating them as a one-dimensional phenomenon. This plan ensured that recovery activities were coordinated with the significance of business and the soundness of engineering. The first classification dimension was severity, which approximated the extent of a risk's drawbacks to business stability. The risks were divided into critical (they can result in system outages, data corruption, or security breaches), major (they can result in outages affecting performance, reliability, or compliance), and minor (they do not result in critical behavior or cosmetic issues). The scope dimension helped determine whether a risk affected the whole system (e.g., JVM settings or shared libraries) or was a module or service risk, allowing the mitigation strategy to be tailored where possible.

Along with this, the aspect of exploitability or exposure was also considered to assess the possibility of turning a risk into a reality in real-world operations. This involved quantifying the number of running execution calls, the surface area of security exposure, and the frequency of code execution. Lastly, mitigability was considered to estimate the effort of the fix/workaround in terms of code complexity and the availability of such dependency upgrades, and to assess whether the workaround will have any side effects on the existing functionality.

This prioritised migration planning became possible through this structured classification, with Table 2 providing a summary: low-impact issues, or those easily overcome, would be migrated to the end of the development lifecycle, and those with system-wide impact and high exploitability would be addressed at the beginning of the development process. Migration effort, in turn, did not regard the stabilization of the system and business continuity as priorities and began solving less important issues step by step.

Table 2. Risk classification for Java 17 migration in SAP-integrated applications

Risk Area	Example	Severity	Scope	Mitigation Priority
Spring Upgrade	Bean post-processor changes	High	System	Medium
Locale Formatting	Java 9+ uses CLDR by default	Medium	UI Module	High
Tomcat Deprecations	ClassLoader hierarchy differences	High	Deployment	Medium
Jenkins	PermSize removal	Low	DevOps	High

To improve the objectivity of prioritisation, the multidimensional classification was codified with a semi-quantitative scoring model. Each risk Ri was evaluated across four normalised dimensions:

Severity (S) – Business and technical impact if the risk occurs

Exploitability (E) – Likelihood of the risk being triggered in real-world runtime scenarios

Exposure (X) – Frequency of the affected component

Mitigability (M) – Relative effort and complexity of remediation of the risk

This multiplicative approach was chosen to ensure that a small value in any of the key dimensions will proportionally reduce the risk priority, and that high-impact, high-likelihood risks will be exponentially prioritized. To make the results comparable, the scores were categorized into priority levels:

- $RS \geq 200$ → Immediate mitigation (Phase 1)
- $100 \leq RS < 200$ → Early migration phase
- $40 \leq RS < 100$ → Scheduled remediation
- $RS < 40$ → Deferred / monitor

This formalization, shown in Table 3, translates the former qualitative categorization into a repeatable decision-enabling tool, ensuring that migration sequencing is instead based on quantifiable technical risk exposure rather than subjective prioritisation. This improves the auditability and reproducibility of the framework in other enterprise Java modernization projects.

Table 3. Semi-quantitative risk scoring model for migration prioritization

Risk Area	S	E	X	M	Risk Score	Priority
Spring Upgrade	4	3	4	3	144	Early Phase
Locale Formatting (CLDR change)	3	4	5	2	120	Early Phase
Tomcat ClassLoader Changes	4	3	3	3	108	Early Phase
Jenkins PermSize Removal	2	2	2	2	16	Deferred

3.3 Mitigation strategies

After risk identification and prioritization, a structured set of mitigation strategies was implemented to address migration risks associated with the shift of enterprise applications to Java 17 runtime environments. The mitigation strategy focused on long-term modernization rather than short-term compatibility fixes. The goal was to improve the security posture, maintainability, and conformity with the Java ecosystem.

Dependency modernization: The core framework and third-party libraries were updated to versions known to be compatible with Java 17. This was done with special consideration for security-critical libraries and popular enterprise frameworks. The application framework stack was updated by upgrading the Spring Framework ecosystem to versions that conform to the Jakarta EE namespace refactoring. The Spring ecosystem migration to Spring 6.x and the corresponding runtime stacks align with the official migration plan to support Java 17 and Jakarta EE 9+, as described in the Spring release guidelines, specifically the Spring Boot 3.0 generation, which targets Java 17 as the baseline runtime. The Spring Boot release notes confirm that Java 17 is considered the minimum supported production runtime in this generation [18,19]. The REST integration layer was also updated from legacy Jersey code to Eclipse Jersey version 3.x or later. This was a prerequisite for migrating the javax namespace to the jakarta namespace, as required by the Jakarta EE 9 specification [20]. Security-sensitive libraries were also assessed. The authentication federation components that relied on OpenSAML were updated to comply with current cryptographic policies and Java 17 security settings. The updates ensured that the components avoided deprecated cipher algorithms and conformed to enterprise security governance. The scripting runtime environment was also enhanced by updating Groovy to version 3.x. This addressed the parser's stability compared to previous ANTLR-based versions and ensured reliable execution in dynamic scripting scenarios.

Source code refactoring and platform compatibility: The application's source code was refactored to remove dependencies on deprecated or removed Java 8 APIs. The file processing components were refactored to follow the stream-oriented I/O paradigm. For instance, the batch processing of documents was refactored from memory-inefficient patterns, such as the entire-file-loading approach, to buffered, streaming approaches using reader-based line-iteration APIs. The encapsulation code was made compliant with the Java Platform Module System (JPMS) by removing the reflective access dependencies on the internal JDK classes [21]. Where possible, the service loader APIs, explicit module exports, or public APIs were used to maintain compatibility with the latest JVM security constraints. The components of internationalization were standardized using the Common Locale Data Repository (CLDR)- based formatting behavior, which was introduced in newer versions of Java [22]. The numeric and date formatting modules were also redesigned to specify the locale's symbols and patterns.

Toolchain and DevOps pipeline modernization: Development environments were migrated to support modern toolchains that enable Java 17 features, such as pattern matching, sealed classes, and enhanced static analysis. The development IDE was harmonized to at least IntelliJ IDEA 2021.2, which offers better refactoring support and Java 17 language syntax checking.

Continuous integration pipelines were refactored to use Jenkins-based workflows to ensure build reproducibility from development through integration to production environments. Build tools were updated to support Java 17 bytecode compatibility. The Maven Compiler Plugin and Maven Surefire Plugin were updated to versions that support Java 17 bytecode compatibility for compilation and test execution [23].

Runtime deployment and JVM optimization: The deployment options were refactored, taking into account the current JVM garbage collection behavior. Controlled performance validation was performed on various garbage collectors:

- G1 Garbage Collector (G1GC) - Throughput and pause time behavior were balanced for enterprise applications.
- Z Garbage Collector (ZGC) - Tested for low-latency high-memory applications with large heap sizes.
- Parallel Garbage Collector - Served as a baseline throughput comparison during performance validation.

The results of load testing showed that G1GC maintained stable operation for business transactions, while ZGC showed better performance in applications that are less sensitive to pause time and involve large heap allocations. Legacy JVM options that were related to the eliminated memory management models were removed. The configuration flags that were related to the memory management of the permanent generation, such as -XX:PermSize, were removed and replaced with metaspace configuration options, such as -XX:MetaspaceSize.

Summary of mitigation effectiveness: The overall set of mitigation strategies has established a modernization baseline that remains stable across dependency management, application code organization, development pipeline automation, and runtime configuration. The migration has ensured compatibility with the operational semantics of Java 17 while keeping production exposure risk to a minimum.

3.4 Validation and testing

Coherent validation and testing at the unit and integration levels were undertaken to ensure functional correctness, stable performance, and consistent behaviour after the migration. The scope of the platform transformation introduced by Java 17 was so large that it was a first-class activity throughout the migration process, rather than a verification step. This approach implied that the migration-induced changes would not lead to regressions or unwanted side effects in the production-critical workflow, and that the system's operational properties would remain the same as before the migration. To ensure robust functional validation, an extensive regression testing process was conducted on the migrated codebase. Over 2,000 test scenarios were run using JUnit and TestNG testing frameworks, focusing on functional equivalence with the Java 8 baseline system. Code quality verification was also enhanced by requiring structural coverage criteria, such as branch coverage above 85%, using automated coverage analysis tools incorporated into the CI process. Regression testing was prioritized on system elements most vulnerable to Java runtime transitions. Specific functional validation efforts were made on:

- Date and time manipulation code paths influenced by CLDR-based localization defaults
- Internationalized financial calculation modules requiring deterministic decimal rounding semantics
- Multi-threaded service managers where JVM scheduling algorithm and memory model variations might cause concurrency-related anomalies

Non-functional validation criteria were also tracked to confirm production-level behavioral characteristics. A shadow deployment model was used to test runtime behavior under a realistic workload scenario without affecting users in the production environment. The Java 17 runtime environment was deployed in passive shadow mode, coexisting with the production Java 8 runtime environment. The production workload was mirrored in the shadow environment, allowing for comparative analysis of runtime behavior. Telemetry and observability data were collected using the ELK stack (Elasticsearch, Logstash, and Kibana). Sample monitoring queries and views were set up to monitor the following aspects:

- Garbage collection pause time distributions
- Heap memory usage trends
- Thread pool usage metrics
- Response latency percentiles (P95 and P99)
- Exception and error event rates for different runtime versions

The above telemetry analyses were carried out using indexed log aggregation queries in the monitoring platform provided by the Elastic ecosystem suite of tools.

Multi-layer testing architecture: The validation approach adopted a multi-layer testing pyramid strategy that included the following layers:

- Unit Testing Layer – High-level functional validation of business logic modules.
- Integration Testing Layer – Validation of service communication flows, authentication modules, and database access.
- Shadow Runtime Testing Layer – Concurrent execution monitoring under production-like loads.
- User Acceptance Testing (UAT) Layer – Final business functionality validation by domain experts.

The multi-layer validation approach adopted in this study is summarized in Figure 2, which illustrates the hierarchical testing framework used to ensure functional, integration, and runtime stability during the Java 17 migration process.

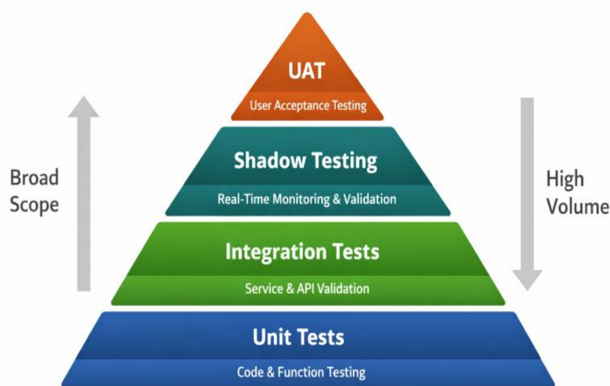

Figure 2. Multi-layer testing pyramid for Java 17 migration

The performance benchmarking was conducted to empirically assess the system's efficiency and responsiveness following the migration. The main indicators observed at this stage were API response times, heap usage, garbage collection frequency, pause time, and application startup time. Apache JMeter, VisualVM, and JVM Monitor are industry-standard tools for applying load, visualizing runtime behavior, and gathering JVM-level measurements [24]. The benchmarking outcomes provided quantitative evidence of the performance gains from the Java 17 migration, demonstrating measurable operational improvements without affecting system reliability or stability, as shown in Figure 3.

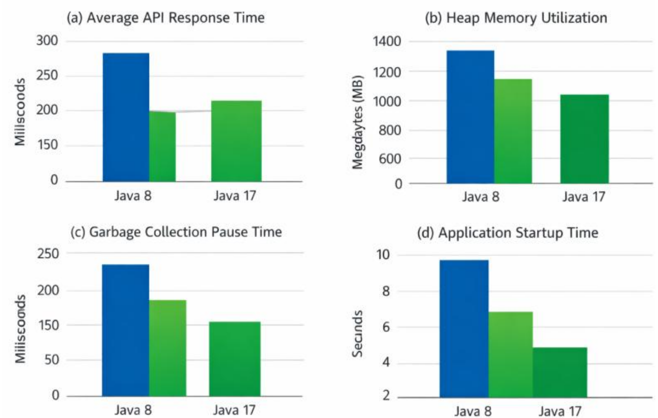

Figure 3. Performance metrics before and after Java 17 migration

4. Secure migration patterns

Enterprise Java application migration, particularly in the SAP environment, is not a technical upgrade but, in a way, a structural change, as it concerns code structure, integrations, and organizational processes. To minimize business disruption, enforce compliance, and ensure stable integration, we devised five safe migration patterns. Each pattern will address specific problems in the shift to Java 17, and the set of these patterns, in turn, will enable the construction of a risk-tolerant framework for upgrades.

4.1 Modularization-first refactoring

Problem Addressed: The overwhelming proportion of the legacy Java 8 apps is monolithic, which uses orchestrated services and dynamic shared state, and is difficult to migrate and prone to regressions or crashes.

Pattern strategy:

Module extraction: The refactorings included transforming legacy components into logical, coherent modules (e.g., core-auth, ui-layer, sap-connector, and saml-auth) to depict dependencies and achieve high functional separation.

Service interfaces: Clean interfaces were also introduced between modules in line with the dependency inversion principle, minimizing coupling and maximizing testability.

Java Platform Module System (JPMS) Transition Readiness: The Java Platform Module System was not an early adoption, though it is a modularization approach that will enable future upgrades to stronger encapsulation, service loading, and more robust runtime control.

Static code scanning pipelines using SonarQube-based analysis indicated an estimated approximate 40% reduction in vulnerability exposure signals associated with shared-state architectural paths. This estimate represents relative improvement in security indicator density rather than absolute threat elimination.

Security benefit: The capability to isolate modules reduces the number of potential vulnerabilities and provides fine-grained access control. For example, only the SAP-connector module can access SAP HANA credentials and adheres to the principle of least privilege.

4.2 Shadow testing with canary releases

Problem addressed: Completing a cutover to Java 17 carries the risk of undiagnosed runtime or integration failures in production workloads, especially with high-risk-profile enterprise systems.

Pattern strategy: Shadow Mode Implementation: The Java 17 app was deployed, and the Java 8 application already in use was also deployed. Production traffic was monitored to observe how the application behaved without interference to end users.

Read-only endpoints: The initial endpoints of the service were opened in read-only mode at the canary, as a type of functional testing, without being subjected to accruing the potential side effects.

Automated log comparison: Both versions were fed telemetry, and the log data were collected and processed using the ELK stack, where behavioral aberrations and differences were automatically identified.

Security benefit: This trend will ensure that end users are not subjected to untested modifications and that critical authentication and authorization procedures, such as single sign-on via SAML, are not destabilized or violated during the migration.

4.3 Secure library upgrade chain

Problem addressed: A transitive dependency change/unverified interaction between components will compound the risk of upgrading third-party libraries and can result in either functional regression or a security concern.

Pattern Strategy: Incremental Upgrade Strategy: The libraries were upgraded in a well-planned order, as shown in Figure 4, to ensure that each dependency could be tested before the next. This plan has reduced the number of cascading failures and ensured that any compatibility problem could be detected and fixed in a single step, as well as spring-legacy-bridge, which is a type of compatibility bridging.

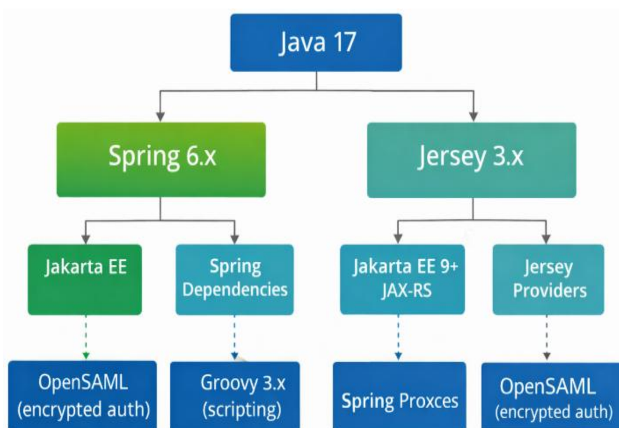

Figure 4. Dependency flow of Spring and Jersey stacks during Java 17 migration

Figure 4 shows the staged dependency upgrade order used throughout the migration process. The directed graph shows the controlled upgrade flow from critical enterprise libraries to avoid cascading compatibility issues. The nodes in the graph represent the major framework and security modules, such as the Spring Framework, the Jersey REST framework, and the OpenSAML authentication stack, while the directed arcs indicate validated upgrade paths and dependency verification routes. The upgrade process was based on a dependency-aware migration order, in which the runtime frameworks were stabilized before upgrading the application security and integration libraries. Each component upgrade was regression-tested before moving on to the next level of the migration stack. The staged upgrade approach helped avoid systemic failure by ensuring compatibility issues remained contained within each migration stage rather than cascading through interdependent components. Compatibility bridging techniques, such as legacy interoperability layers (e.g., compatibility bridge components, similar to the spring-legacy-bridge patterns), were used to ensure continuity during transition periods.

Security Patch Audit: All updates in the library were compared against a list of known vulnerabilities (OWASP Dependency-Check), and it was verified that the upgrades did not introduce any new vulnerabilities and instead minimized regressions.

Security benefit: This trend will ensure that all identified vulnerabilities are managed and that the stability of the migration process is tracked throughout. The solution will minimize the risk of unforeseen implementation defects or integration failures by going through the phases of the upgrade to Java 17 and confirming the interim steps.

4.4 Backward-compatible abstractions

Problem addressed: As the new type of safety, access referential controls, and encapsulation restrictions features are introduced in Java 17, in the form of JPMS, sealed classes, and module-level access controls, the legacy code will violate them, either because the legacy code is using reflection intensively or because classes are being loaded dynamically.

Pattern Strategy:

Adapter interfaces: Not all the legacy logic could be refactored at the same time; adapter interfaces were introduced to abstract implementation classes. This also made the bare functionality accessible without violating the more stringent access controls of Java 17.

Factory-based instantiation: Factory-Based Instantiation Direct Class Practice Factories Direct replaced service loaders or custom factories, which can inject alternative implementations flexibly and do not require as much runtime reflection [25].

Legacy layer retention: The closely coupled modules implemented using Groovy-based scripts were contained within a legacy-core module. This module's operations and activities were to be transferred and phased out.

Security benefit: The pattern not only reduces the risk of runtime errors but also enables backward compatibility. At the same time, it can present a logical migration path to existing security principles, including the use of closed interfaces and module encapsulation, thereby enabling long-term service and enforceable least-privilege principles.

4.5 Security policy adaptation

Problem Addressed: Java 17 has more robust default security, including defaulting on TLS 1.3 and JEP 411-deprecation of the Security Manager, both of which are likely to make old configurations or even generate new, unknown behavior with sensitive code [26].

Pattern strategy: TLS Policy Migration: TLS 1.2 and above was explicitly enabled and applied to internal services and SAP connectors; cipher suites were configured to exclude weakened or insecure algorithms, and it was verified that all integrations would not be harmed by ensuring safe communication.

Access control files: Java.policy files that were decommissioned long ago were scrapped and replaced by code-level security managers and runtime assertions, which offer more detailed access control without disrupting the established conventions of Java security.
Third-party identity compatibility: It has been fully tested that SAML tokens can be correctly parsed and verified both with normal cryptographic policy, e.g., using SHA-256 rather than SHA-1 in OpenSAML, and should not break anything [27].
Security benefit: The pattern will ensure that the application is consistent with the existing cryptographic and access control schemes, the application attack surface is minimized on possible exploits, and the pattern is the foundation of the implementation of zero-trust security architectures.
The action plans involved in solving the most prevalent modernisation concerns listed in Table 4, as a cluster result, in the secure migration patterns that should be implemented to solve the dependency upgrades, API depreciation, integration with enterprise ecosystems, and runtime security compliance. By adhering to these trends, the migration of enterprises to Java 17 can be risk-mitigated, without compromising functional stability or security guarantees. Using these patterns, enterprises will be able to transition to Java 17 with resilience to risk while retaining functional stability and security guarantees.

Table 4. Summary of secure migration patterns for Java 17 transition

Pattern	Risk Addressed	Tools / Techniques Used	Security Value
Modularization-first	Code complexity, tight coupling	Maven modules, JPMS-ready structure	Isolation and least-privilege enforcement
Shadow Testing & Canary	Runtime breakage	ELK Stack, JMeter, Jenkins pipelines	Zero-downtime validation
Library Upgrade Chain	Dependency incompatibility	OWASP Dependency-Check, Maven Enforcer, Compatibility libraries	Patched CVEs, reduced attack vectors
Backward-Compatible Abstractions	Reflection and sealed type issues	Service loaders, interfaces, legacy bridging	Smooth transition and type safety

5. Case study: migration in a large SAP Java stack

5.1 Overview of the application environment

Application context: The case study is a mission-critical enterprise Java application in a hybrid deployment environment. The hybrid deployment environment combines internal business workflows and enterprise backend services. The application provides functionality for document processing, analytical workloads for SAP HANA, and identity federation services using SAML authentication protocols. The application migration was required to ensure reliability, security, and performance during the transition from Java 8 to Java 17. Since the application is directly connected to business-critical SAP services and authorization workflows, the migration required regression testing and a phased approach. To avoid redundancy in the description, this subsection focuses on the deployment topology's characteristics and summarizes the pre-migration stack configuration in Table 5.

Table 5. A summary of the pre-migration stack configuration

Layer	Component	Version / Specification
Runtime Platform	Oracle JDK	1.8.0_202
Application Framework	Spring Ecosystem	Spring 5.2 baseline
Web Container	Apache Tomcat	9.0 series
REST Service Layer	Jersey (JAX-RS Implementation)	Version 2.31
Security Assertion Processing	OpenSAML	Version 2.6
Scripting Engine	Groovy	2.5 runtime
Enterprise Integration	SAP Java Connector (JCo)	ABAP interoperability adapter
CI/CD Pipeline	Jenkins Automation Platform	Enterprise builds orchestration
Database Platform	SAP HANA	Hybrid transactional and analytical storage
Development Environment	IntelliJ IDEA	2020.3 edition

Architectural deployment characteristics: The application uses a microservice-oriented architecture, with a philosophy of autonomous functional services communicating via well-defined API contracts. The application has a hybrid deployment architecture that combines on-premises enterprise systems with cloud-hosted Java services. In particular, the application's backend business processes, implemented in ABAP workflow engines, are accessed through secure integration channels using SAP connector technologies. The hybrid deployment architecture adds new complexity to the application's operational aspects, which include:

- Network-level security enforcement
- Consistency in identity federation
- Communication latency management
- Governance control across environments

Due to these constraints, the migration approach was designed to be a comprehensive modernization effort rather than a straightforward runtime upgrade.
Migration design considerations: The migration from Java 8 to Java 17 involved the following considerations simultaneously:

- Application source code compatibility
- Third-party dependency lifecycle alignment
- Build pipeline modernization
- Runtime configuration rationalization
- Security protocol hardening

The migration process was executed through controlled iteration cycles to maintain functional correctness while lowering enterprise risk.

5.2 Migration preparation

The migration preparation stage involved identifying, evaluating, and mitigating potential risks to ensure that the Java 17 migration would not disrupt critical business operations. This was an active risk-mitigation phase rather than a reactive one, given the application's central position in the SAP framework. This was to be done to create a clear view of technical dependencies, platform constraints, and

behavioral changes with Java 17, and make any production-facing change before any such changes were made. To do this preparation, a full inventory and dependency audit of the entire codebase and build setup was done. To examine both transitive and direct dependencies, industry-standard tools, such as OWASP Dependency-Check and the jdeps utility, were used [15,16]. This exercise identified 57 third-party libraries that needed updating or replacement to achieve full compatibility with Java 17. Simultaneously, the use of degraded APIs and potential runtime incompatibilities was evaluated systematically using IntelliJ IDEA inspections and the jdeprscan tool. Such analyses enabled earlier identification of code segments that would not work under the tighter access controls and revised runtime limits of Java 17. Along with the dependency analysis, compatibility and impact analyses were conducted in greater detail to determine the implications of Java 17 language and platform features for the application's functional and non-functional behaviour. Special focus was on enhanced encapsulation imposed by the Java Platform Module System, the introduction of sealed classes, the elimination of logic that relied on the Security Manager, and changes to garbage collection behavior [28]. This test was conducted to assess functional correctness and operational capabilities, and the modules examined included secure document processing, SAML-based authentication, SAP HANA analytics, and the execution of ABAP jobs. This meant that by managing these effects at the preparation stage, the migration reduced downstream risks and continuity of core enterprise workflows after the transition.

5.3 Migration execution and core changes

The migration implementation stage consisted of consistent architectural, framework-level, and code-level adjustments that could be advantageous for achieving full compatibility with Java 17 and for improving runtime performance and maintainability. Table 6 summarises the most notable core modifications made during the migration, focusing on replacing legacy platform components with their Java 17-compatible versions. The combination of these upgrades created a modernized base that was in line with the long-term support (LTS) requirement and with the up-to-date enterprise Java requirements.

Table 6. Core changes implemented during Java 17 migration

Area	Before	After
Java Version	1.8.0_202	17.0.8 (LTS)
Spring Framework	5.2	6.1
Apache Tomcat	9.x	10.1
Jersey (JAX-RS)	2.x	3.1
Groovy	2.5	4
Garbage Collector	Parallel GC	G1 GC (Java 17 default)
Date / Decimal Formatting	Compact locale	CLDR

Among the changes needed during execution were refactoring Groovy-based components to be compatible with Java 17 bytecode and resolving performance and stability issues observed in previous versions. The reversion to a more recent Groovy runtime addressed the constraints in ANTLR parsing that had previously been encountered and enabled easy compatibility with the Java 17 runtime. Placing Groovy scripts in distinct modules with well-defined classloader boundaries helped prevent classpath pollution and potential interference with the core application code. This form of modularization preserved backward compatibility with the legacy codebase script but allowed the main application codebase to use all the capabilities of Java 17, including improved encapsulation and runtime optimizations.

Changes were also necessary to the SAP integration layer to support the more restrictive access controls and sealed classes introduced in Java 17. Even though the new runtime was mostly compatible with the SAP Java Connector (SAP JCo), new wrapper abstractions were implemented to control access boundaries and provide predictable behavior. In addition, transport-level security configurations were revised to align with the tighter cryptographic default settings in Java 17, including changes to supported TLS protocols and certificate trust stores. These modifications were tested thoroughly to maintain the consistency of ABAP-Java communication, the integrity of remote job execution, the SAML procedures in the authentication process, and other operations that can be performed by SAP.

The execution phase was accompanied by comprehensive testing to ensure that not only was the migrated system functioning properly, but it was also non-functionally stable. JUnit 5 and end-to-end tests were performed on more than 3,000 unit and integration tests, using Java 17, to validate the underlying business logic and inter-module interactions. Also, User Acceptance Testing (UAT) was performed with a great emphasis on business-critical and high-risk processes. The validation scenarios were focused on the accuracy of financial computations, specifically, decimal round-off and time zone management, the accuracy of generated secure PDF documents, and operational control of SAP jobs being run through ABAP integration. This multi-layered testing system provided the migration with operational integrity, met enterprise functional requirements, and delivered a stable, production-ready system.

5.4 Observed outcomes

In terms of performance Gains, the benchmarks for our SAP-integrated application are shown in Table 7, though these results may not generalize to other workloads or architectures. In Terms of Functional Stability, the migrated system was functional, as all critical business logic had been tested with historical production data. The locale-sensitive user interface drawing was also treated with special consideration, as it may be affected by the locale processing switching of locale processing in Java 17 to CLDR-based locale processing in such cases, these modules were checked and modified to prevent confusion. Jenkins continuous integration pipelines were also updated to take in Java 17 runtime changes to use the -XX: MaxMetaspaceSize property to define JVM memory options in place of the obsolete PermGen flags. The expenditure on the dimensions of deployment, as given in Table 8, indicates that these dimensions have greatly enhanced in JVM tuning, container support, and CI/CD integration. No more unpredictable performance, the most effective use of resources, and

simplified operations in the Java 17 environment resulted in these improvements, making it a stable and maintainable production deployment.

Table 7. Performance Gains observed post

Metric	Before Migration	After Migration	Change
Report Processing Time	5.2 min	3.8 min	↓ 27%
Groovy Execution Latency	850 ms	620 ms	↓ 27%
Memory Usage (Avg. Heap)	1.2 GB	950 MB	↓ 21%
Throughput (Requests/min)	2,200	2,750	↑ 25%

Table 8. Deployment dimension comparison: Java 8 vs Java 17

Deployment Dimension	Java 8	Java 17
JVM Flags	PermSize, MaxPermSize	MaxMetaspaceSize, improved GC tuning options
Garbage Collector	Parallel GC	G1 GC (default), ZGC (optional)
Remote Debugging	-agentlib:jdwp (legacy syntax)	Updated syntax compatible with JPMS
Container Support	Partial	Full (enhanced container awareness, layered class data sharing)
Jenkins Integration	Legacy JVM flags	Modern pipelines, improved memory tuning

Table 9. Development tooling and language support comparison: Java 8 vs Java 175.5

Factor	Java 8	Java 17
Language Expressiveness	Moderate (requires boilerplate code)	High (records, pattern matching, sealed classes)
IDE Compatibility	IntelliJ ≤ 2020.3	IntelliJ ≥ 2021.2
Build Tool Compatibility	Legacy Maven / Ant	Modern Maven / Gradle
Testing Frameworks	JUnit 4	JUnit 5 (modular, extension model)
Static Code Analysis	Limited	Enhanced (improved tooling support and static analysis integration)

In terms of developer experience, the transition to Java 17 significantly improved it because modern language features such as records enabled compact data modeling, switch expressions provided more natural multi-branch logic, and type inference made code cleaner. These features reduced boilerplate and made them more maintainable, and allowed developers to take advantage of better business logic. IntelliJ IDEA 2021.2 and later releases were all supported on Java 17, with the new language features and constructs being fully supported in advanced debugging, static inspection, and code inspection. This enabled developers to identify migration-related problems early, safely refactor, and utilize the capabilities of the latest tooling through this seamless integration. Java 17, as shown in Table 9, offers a more expressive and modern development ecosystem that is compatible with existing IDEs, testing systems, and statistical analysis software. Productivity for developers improved with this modernization, as did code quality and overall confidence in maintaining enterprise-grade applications compared with Java 8.

5.5 Lessons learned

The migration activity has provided several practical insights directly related to the previously identified risk categories and secure modernization goals. The insights are presented as prescriptive recommendations to facilitate future enterprise Java migration projects and relate to the secure migration patterns identified in the study.

Key takeaways and recommendations: Early Dependency Risk Auditing: Migration of Java to the enterprise level should begin with a systematic dependency lifecycle audit rather than code refactoring. Libraries with long-lived systems may have orphaned systems, such as authentication modules, outmoded XML processors, or cryptographic provider versions. These parts are used to introduce compatibility-breaking changes and latent CVE exposures during the transition to modern runtimes such as Java 17. A dependency analysis should be a formal classification of artifacts into support status, risk propagation (transitivity), API stability, and security posture. This can be used to make forward architectural decisions, such as replacing the old authentication stacks, before refactoring cascades through the codebase.

Suggestion: Introduce an implementation of the pre-migration dependency audit phase that analyzes lifecycle status, vulnerability exposure, and compatibility at runtime before modernization begins.

Categories of Risk: Dependency compatibility and vulnerability risks:

Mandated locale processing validation (CLDR Effects): The existing Java runtime environments are set to the Unicode Common Locale Data Repository (CLDR) default, which can lead to variations in how the application displays formatting, number grouping, currency symbols, and dates. These subtle formatting variations may be dragged into reconciliation bugs or even UI regression in enterprise systems - and those in particular touching upon financial reporting or compliance artifacts, too. These behavioral changes are not compilation errors but semantic runtime errors. Therefore, they cannot be determined solely through statistical analysis. Monetary calculation systems are expected to reflect locale behavior, which may include policies that govern decimal treatment, such as precision and rounding, to prevent drift when using CLDR-based formatting.

Suggestion: Implement automated locale regression pipelines to check formatting, representation of currency, and serialization products in Java 17+ runtimes. Locale validation is an essential migration point that should be incorporated into globalization-sensitive applications.

Categories of Risk: Risks in runtime behavioral change.

Shadow testing as a production risk-containment strategy: Unit and integration tests are used to verify the functional correctness of the simulated environment. The conditions, however, cannot exactly reflect real-world concurrent models and integration edge cases. Simulated Runtime Validation Layer Shadow Testing, where Java 17 workloads are run in parallel to production live traffic, is a simulated layer.

The model helps identify integration drifts, serialization errors, threading issues, and exception pattern anomalies early. The persistent comparison of telemetry data from the legacy and modern environments will help detect anomalies before the production cutover.

Recommendation: Introduce shadow runtime validation as an official stage in the upgrade lifecycle for enterprise platforms. Telemetry metrics, exception frequency analysis, and workflow integrity should be systematically compared between the pre- and post-upgrade runtime environments.

Categories of Risk: Integration and run-time risks.

Garbage collector selection and runtime stability: JVM garbage collector settings significantly affect system stability, especially in high-throughput enterprise environments connected to other systems (e.g., transient connector threads, I/O-intensive operations). Migration to Java 17 introduces additional GC options: G1 and ZGC, each with distinct throughput and pause-time characteristics. It can be more useful than raw throughput improvements in latency-sensitive integration processes with pause-time predictability. Whereas G1 provides balanced performance on medium-sized heaps, ZGC must be tested empirically in large heap deployments to ensure predictable latency under production traffic.

Recommendation: Structured load testing and heap profiling should be conducted before finalizing the choice of garbage collector. The decisions made regarding GC tuning should be workload-specific and evidence-based, rather than relying on default configuration assumptions.

Categories of risk: Toolchain/runtime optimization risks.

Security framework migration as an architectural transformation: A change in architecture, e.g., from Spring Security 5.x to 6.x, is also an upgrade of the security structure and should not be understood as an incremental configuration change. Authentication semantics and enforcement boundaries can be changed by variations in filter chain design, authorization model design, and session validation mechanisms. Security migration affects token validation, CSRF, method-level authorization, and dependency injection. Thus, treating the upgrade as an easy version bump may introduce one or more authorization regressions or unintended paths to exposure.

Recommendation: Security migration should be treated as a system-level redesign. Test the end-to-end authentication workflow, test authorization scopes under realistic conditions, and review the filter chain composition to verify that it satisfies modern security requirements.

Categories of risk: Security vulnerability mitigation pattern.

Guiding principle for enterprise migration programs: The main lesson this study has taught is that, for enterprise migration to succeed, it depends less on the target runtime version and more on the rigor of risk governance and architectural preparedness. The modernization risk is significantly reduced by using a staged validation model that includes dependency lifecycle analysis, behavioral regression testing, shadow runtime verification, GC performance evaluation, and security workflow validation. Enterprise modernization is therefore best understood as an organized, risk-managed transformation program, not a technical upgrade program.

General recommendation: Implement a gradual migration structure comprising architectural preparedness testing, automated regression pipelines, shadow-execution validation, and security testing. Risk prioritization metrics and empirical validation at every stage should govern platform modernization.

6. Comparative analysis

Performance, memory, productivity, and compatibility comparison:

The comparative analysis assesses the impact of migration by comparing the performance of the traditional Java 8 stack with that of the modernized Java 17 runtime environment. The assessment is based on performance, memory consumption, productivity, and compatibility of enterprise integration with the SAP application environment. The empirical data was collected through workload simulation on the migrated Spring-based enterprise service platform. Although performance may vary with application design and workload patterns, the empirical data in Figure 3 and Table 10 illustrate consistent benefits of modernization.

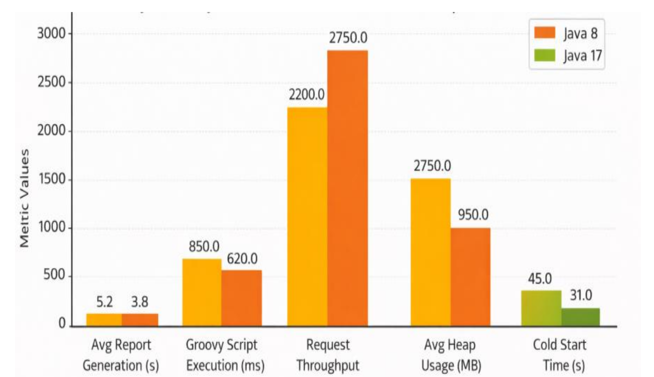

Figure 3. Java 8 vs Java 17-Performance metrics comparison

Table 10. Quantitative comparison results

Metric	Java 8 Baseline	Java 17 Migrated Stack	Improvement Indicator
Average Report Generation Time	5.2 s	3.8 s	~27% reduction
Groovy Script Execution Time	850 ms	620 ms	~27% reduction
Request Throughput	2200 requests/sec	2750 requests/sec	~25% increase
Average Heap Memory Usage	1200 MB	950 MB	~21% reduction
Application Cold Start Time	45 s	31 s	~31% reduction

The performance gains are mainly due to the improved Just-In-Time compilation optimization, efficient stream-based I/O processing, and effective memory allocation management based on the new JVM execution semantics. The runtime performance improvements are also affected by the use of new parser architectures in dynamic scripting environments like Groovy 3.x, which uses ANTLR-based parsing improvements.

Developer productivity and language modernization impact:
The impact of modern language features such as records, pattern-matching constructs, and sealed class hierarchies in Java 17 helped in simplifying the complexity of boilerplate code. The language features enable robust enforcement of type safety and facilitate modular design patterns in software development without any penalty on runtime performance [29]. Table 11 describes the transition-maintained compatibility with SAP-administered enterprise backend systems, as well as enhanced runtime security properties.

Table 11. SAP ecosystem compatibility: Java 8 vs Java 17

Aspect	Java 8	Java 17	Notes
SAP JCo Support	Yes	Yes	Binary-compatible, no code changes required
ABAP–Java Messaging	Stable	Stable	TLS handshake stricter in Java 17
Locale and Formatting	Compact locale data	CLDR	Requires regression test updates
Legacy APIs	Widely used	Deprecated / removed / relocated	Refactoring required; modern alternatives recommended
Module System	Not present	JPMS (Java Platform Module System)	Optional adoption; introduces stronger encapsulation constraints

Generalizability and external validity:
The findings of this migration study are generally relevant to enterprise Java applications with similar properties, such as:
- Spring-based microservice architectures
- Hybrid on-premise and cloud deployment scenarios
- Integration-heavy business processes
- Security-critical authentication environments

Nonetheless, performance benefits and changes may differ based on:
- Workload allocation patterns
- Garbage collector settings
- Network latency profiles
- Dependency chain composition
- Environments and hardware platforms

Hence, the findings should be regarded as signs of architectural migration rather than general performance assurances [30].

7. Conclusion

The migration of enterprise-level SAP programs from Java 8 to Java 17 is a complex task that spans the domains of architecture safety, performance, and compatibility, as well as developer efficiency. As we have been able to prove in this paper, even such a movement, with the influence of a risk-based and systemized approach, can come with substantial rewards:
- The security level will be improved by eradicating the old libraries and using Java 17 secure-by-default platform security improvements.
- 20 percent or better performance improvement: majorly in the form of garbage collection processes (G1GC), enhanced performance by Groovy over runtime, and additional thin APIs.
- Less boilerplate and modularization (with new Java features, e.g. records, sealed under classes) and new CI/CD pipelines enhanced maintainability.
- Adherence to SAP Cloud paradigms means that it is ready to be implemented as a containerized one, and be compatible with SAP Business Technology Platform (BTP) in the long-run.

In the meantime, migration did bring some issues, such as how to deal with deprecated APIs, how to handle locale and decimal behavior changes with the adoption of the CLDR, fundamental changes in deployment models, such as metaspace management, and new debugging facilities. Such intricacies confirm the fact that enterprise Java migrations are not minor upgrades, but rather modernization initiatives, which need to be properly planned, with toolset consistency, stakeholder acquisition, and extensive testing. Future directions include:
- Expanding the framework to Java 21.
- Development of the automated migration toolkits.
- Containership and microservices solutions are implications of the framework, too.

Ethical issue
The authors are aware of and comply with best practices in publication ethics, specifically regarding authorship (avoidance of guest authorship), dual submission, manipulation of figures, competing interests, and compliance with research ethics policies. The authors adhere to publication requirements that the submitted work is original and has not been published elsewhere.

Data availability statement
The manuscript contains all the data. However, more data will be available upon request from the authors.

Conflict of interest
The authors declare no potential conflict of interest.

References

[1] Oracle. (2021). Java SE 8 End of Public Updates. Oracle Technology Documentation. https://www.oracle.com/java/technologies/java8-support.html

[2] Oracle. (2021). JDK 17 Feature Specification Overview. OpenJDK Project. https://openjdk.org/projects/jdk/17/

[3] Dahanayake, D., Deelsnyder, J., & van der Weide, T. (2020). Challenges in integrating Java with enterprise SAP environments: A case-based analysis. Proceedings of the 2020 IEEE International Conference on Enterprise Systems, pp. 157–164. IEEE. https://doi.org/10.1109/ES.2020.00027

[4] Islam, M. R., Mahmood, A., & Hossain, M. (2021). Modernizing Java applications: Techniques and tools for migration. Journal of Software Engineering Research and Development, 9(1), 1–20. https://doi.org/10.1186/s40411-021-00123-4

[5] Oracle. (2016). JEP 246: Use G1 garbage collector as the default server GC. OpenJDK Project. https://openjdk.org/jeps/246

[6] Oracle. (2018). JEP 333: ZGC scalable low-latency garbage collector. OpenJDK Project. https://openjdk.org/jeps/333

[7] Oracle. (2019). JEP 361: Switch expressions (standard feature). OpenJDK Project. https://openjdk.org/jeps/361

[8] Oracle. (2021). JEP 394: Pattern Matching for instanceof. OpenJDK Project. https://openjdk.org/jeps/394

[9] Oracle. (2021). JEP 395: Records in Java. OpenJDK Project. https://openjdk.org/jeps/395

[10] Oracle. (2021). JEP 409: Sealed Classes and Interfaces. OpenJDK Project. https://openjdk.org/jeps/409

[11] Oracle. (2016). JEP 252: Default locale data based on Unicode CLDR. OpenJDK Project. https://openjdk.org/jeps/252

[12] SAP. (2024). SAP Java Connector (JCo) Technical Documentation. SAP Help Portal. https://help.sap.com/docs/JCO_LIB

[13] SAP. (2023). SAP Business Technology Platform Documentation. SAP Official Portal. https://www.sap.com/products/technology-platform/btp.html

[14] Reinhold, M. (2017). The State of the Java Module System. OpenJDK Jigsaw Project. https://openjdk.org/projects/jigsaw/spec/sotms/

[15] OWASP Foundation. (2023). OWASP Risk Rating Methodology. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[16] OWASP Foundation. (2023). OWASP Dependency-Check Documentation. Available: https://owasp.org/www-project-dependency-check/

[17] Apache Software Foundation. (2024). Apache Maven Dependency Plugin – Official Documentation. Available: https://maven.apache.org/plugins/maven-dependency-plugin/

[18] VMware. (2022). Spring Framework 6.0 Reference Documentation. Available: https://docs.spring.io/spring-framework/docs/6.0.x/reference/html/

[19] VMware. (2022). Spring Boot 3.0 Migration Guide. Available: https://docs.spring.io/spring-boot/docs/3.0.0/reference/htmlsingle/

[20] Eclipse Foundation. (2023). Jakarta EE 10 Platform Specification. Available: https://jakarta.ee/specifications/platform/10/

[21] Oracle. (2017). JEP 261: Module System (Project Jigsaw). OpenJDK Project. Available: https://openjdk.org/jeps/261

[22] Unicode Consortium. (2024). Unicode CLDR Project Documentation. https://cldr.unicode.org/

[23] Apache Software Foundation. (2024). Apache Maven Project Documentation. Available: https://maven.apache.org/

[24] Feitelson, D. G. (2015). Workload Modeling for Computer Systems Performance Evaluation. Cambridge University Press. https://doi.org/10.1017/CBO9781107280142

[25] Oracle. (2021). JEP 376: Hidden Classes. OpenJDK Project. Available: https://openjdk.org/jeps/376

[26] Oracle. (2021). JEP 411: Deprecate the Security Manager for Removal. OpenJDK Project. Available: https://openjdk.org/jeps/411

[27] NIST. (2020). Security and Privacy Controls for Information Systems and Organizations (SP 800-53 Rev. 5). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-53r5

[28] Oracle. (2021). JEP 403: Strongly Encapsulate JDK Internals. OpenJDK Project. Available: https://openjdk.org/jeps/403

[29] Li, Z., Liang, P., & Avgeriou, P. (2015). Architectural debt management in enterprise systems: A case study. Journal of Systems and Software, 110, 19–33. https://doi.org/10.1016/j.jss.2015.08.025

[30] Mens, T., & Demeyer, S. (2008). Software Evolution. Springer. https://doi.org/10.1007/978-3-540-76440-3

This article is an open-access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).