JENRS

Received: 09 July, 2025, Revised: 25 August, 2025, Accepted: 27 August, 2025, Online: 12 September, 2025

DOI: https://doi.org/10.55708/js0409002

Enterprise-Grade CI/CD Pipelines for Mixed Java Versio Environments Using Jenkins in Non-Containerized Environments

Sravan Reddy Kathi*
Bridgeport, Pennsylvania, USA
*Corresponding author: Bridgeport, Pennsylvania, USA, sravanreddykathi55@gmail.com

ABSTRACT: Enterprises with large Java codebases are increasingly facing challenges in maintaining different versions of Java, mainly during upgrade of legacy Java 8 to modern long-term support (LTS) versions like Java 17. These concerns are majorly identified in environments where several Java versions co-exist, such as during incremental migration or version restrictions based on dependencies. This paper proposes a model for designing and implementing enterprise-grade CI/CD pipelines that support mixed Java version development using Jenkins. The proposed solution manages build execution, automated testing, static code analysis, and deployment validation in different Java versions without depending on container tools like Docker or Kubernetes. A Spring Boot-based enterprise application case study demonstrates the effectiveness of the approach, showcasing improvements in automation, developer productivity, and avoiding regression. By following best practices and real-world constraints, this work contributes a reproducible and extensible solutions to organizations that are scaling their Java applications.

KEYWORDS: CI/CD pipeline, Java 8, Java 17, Jenkins, Spring Boot, Software modernization, multi-Java environment, Legacy system upgrade, Static code analysis, Enterprise DevOps

1. Introduction

Continuous integration and continuous delivery (CI/CD) pipelines that can handle complex, heterogeneous environments are essential for modern software development. Maintaining applications developed on different Java versions is a common problem for enterprises, especially when switching from Java 8—which is no longer receiving public updates—to more recent Long-Term Support (LTS) versions like Java 11 or Java 17 [1]. This situation frequently occurs in large Enterprises where microservices or modularized components coexist with legacy systems [2].

Significant enhancements over Java 8 are brought about by Java 17, an LTS release, which includes the Java Platform Module System (JPMS), improved garbage collectors (such as G1GC and ZGC), and expressive language features like records, sealed classes, and pattern matching [3]. Enterprise application migrations to Java 17 are rarely straightforward and not simple. Teams may need to support multi-Java environments both during and after migration because crucial dependencies, like Spring Framework components, third-party libraries, or even build tools, may still depend on Java 8 compatibility [4].

CI/CD pipelines are crucial for facilitating safe and scalable modernization in these kinds of situations. Jenkins is a popular open-source automation server that offers the ability to plan builds, tests, and deployments in a variety of Java environments. When properly set up, it can assist with backward compatibility validation, run unit tests across various Java runtimes, and enforce security and quality standards with SonarQube, SpotBugs, and Checkstyle [5].

Although CI/CD is widely used in DevOps culture, little scholarly research has been done on how pipelines should be built to accommodate different Java versions in business settings, especially in non-containerized settings that do not use Docker and Kubernetes. By offering a structured Jenkins-based pipeline architecture that supports applications that have been compiled, tested, and validated for both Java 8 and Java 17, this paper seeks to close that gap without adding needless complexity or infrastructure overhead.

2. Background and Related Work

Modernizing legacy systems is essential for maintainability, security, and performance as enterprise Java applications get bigger and more complex. Because it introduced lambda expressions and the Stream API, Java 8, which was released in 2014, gained a lot of traction. However, it lacks the performance optimizations and contemporary language features of more recent Long-Term Support (LTS) releases, such as Java 11 and Java 17 [6]. Organizations are facing mounting pressure to upgrade their applications to more recent versions that provide vendor and community support as Java 8's public updates come to an end [7].

2.1. Java Version Evolution and Migration Challenges

The language and runtime of Java underwent significant modifications in later iterations. The Java Platform Module System (JPMS), which was introduced in Java 9, changed the way applications are loaded and structured and enforced strict encapsulation [8]. Java 14–17 greatly increased the expressiveness of the language by introducing sealed classes, records, pattern matching, and improved switch expressions [9]. For large-scale applications, more recent garbage collectors such as G1GC and ZGC provide better memory management and shorter pause times [10].

Despite these advantages, switching from Java 8 presents serious compatibility issues, particularly in business settings. Applications need to be checked for build system updates, incompatible third-party libraries, and deprecated or removed APIs. For instance, to support newer language features, tools such as Maven and Gradle need plugin and configuration updates. Java version dependencies in frameworks like Spring and Jersey need to be properly handled [11]. Transitive dependency updates may unintentionally cause regressions, particularly when third-party libraries stop supporting older Java versions, according to Shah et al. [12].

2.2. CI/CD in Enterprise Java

Pipelines for continuous delivery (CD) and continuous integration (CI) are essential for reducing the risks associated with migration. Jenkins' adaptability, plugin extensibility, and robust community support help it maintain its position as a leading CI/CD solution. Using tools like SonarQube and Checkstyle, it can integrate quality gates, run automated tests, and coordinate builds across various Java versions [13]. Because of this, Jenkins is especially well-suited to handling transitional states during modernization when applications depend on a variety of Java versions.

Not all enterprise contexts are prepared for containers, even though many companies use containerization tools like Docker and Kubernetes to separate and scale Java environments. Widespread adoption of containers may be impeded by resource limitations, security policies, or legacy system constraints. Although they require more setup work, Jenkins pipelines set up on virtual machines or bare-metal servers provide a good substitute in these situations [14].

2.3. CI/CD for Mixed Java Environments

There aren't many studies that specifically address CI/CD design for projects with mixed Java versions. Pereira et al. [16] talk about the difficulties of replacing and deprecating APIs in enterprise codebases, while Deligiannis et al. [15] investigate the modularization issues that arise when integrating JPMS into legacy Java systems. Nevertheless, rather than build automation, the focus of both studies is code-level migration.

Whitepapers and community discussions frequently suggest utilizing Jenkins agents set up with toolchains for Java 8 and Java 17 to isolate build jobs based on Java version. Teams can concurrently compile, test, and analyze applications in both environments thanks to these agents. Nevertheless, peer-reviewed literature hardly ever formalizes or documents this practice.

2.4. Gaps in Existing Research

Most of the literature currently in publication concentrates on either CI/CD automation or Java migration separately. Research that methodically examines CI/CD design patterns that support multiple Java versions is conspicuously lacking, especially in non-containerized enterprise settings. Furthermore, practical limitations like Jenkins integration with SAP-oriented libraries, legacy dependencies, or backward-compatible test automation are not considered in the current work.

To fill these gaps, this paper suggests a CI/CD pipeline architecture based on Jenkins that allows for mixed Java versions throughout the migration process. Without the need for container orchestration tools, the pipeline is made to manage a variety of build scenarios, execute parallel tests in various environments, and enforce quality and security standards.

3. Methodology

A thorough methodology for implementing enterprise-grade CI/CD pipelines for Java applications moving from Java 8 to Java 17 is presented in this section. The method is tailored for use in enterprise settings, especially those that are limited by legacy environments that do not support containerization. System audit and dependency mapping, environment setup, pipeline architecture design, testing validation, and iterative refinement are the five main stages of the methodology.

3.1. System Audit and Dependency Mapping

A comprehensive audit of the current Java application is the first stage in the modernization process. The following sub-activities are included in this:

- **Source Code Audit**: Review the codebase for instances of internal or proprietary Java API usage, look for deprecated or removed APIs, and evaluate modularity and test coverage. Java 8-specific constructs can be found with the help of tools like Java Migration Toolkit, jdeps, and jdeprscan.
- **Third-Party Library Assessment**: Many enterprise Java applications depend on third-party libraries, such as Hibernate, Jersey, ActiveMQ, OpenSAML, and Apache CXF. Maven Dependency Tree and OWASP Dependency-Check [17] are two tools that assist in determining compatibility with Java 17 and identifying outdated dependencies.

Components are classified as high, medium, or low risk according to their effect on enterprise reliability and Java 17 compatibility, as shown in Figure 1 and Table 1. Targeted planning and early mitigation of significant obstacles are made possible by this risk-based perspective.

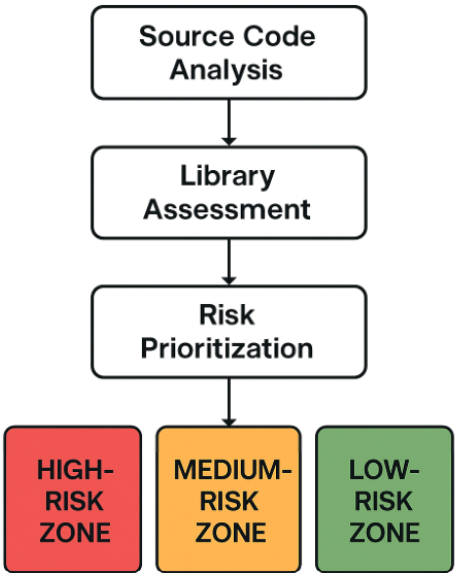

Figure 1: Dependency and Risk Classification Flow

Table 1: Risk Classification of Modernization Components

Component	Example Tools	Risk Level	Justification
Source Code Audit	jdeprscan,jdeps, Java Migration Toolkit	High	Deprecated APIs can break at runtime or compilation; internal APIs may be removed
	Internal Api's	High	These are unsupported and may no longer exist in Java 17
	Manual, Static analysis tools	Medium	Impacts ability to adopt JPMS and confidence in migration regression
Third-Party Libraries	OWASP Dependency-Check, Maven Dependency Tree	High	Namespace migration and unsupported Java EE APIs
	Manual, CVE database	High	Security-critical; old versions may not support Java 17
	OWASP Dependency-Check	Medium	May require log framework upgrades but core features work
	jdeps, Revapi	Medium	Works with Java 17 but requires tuning or module opens
	Maven Plugin, Dependency Tree	High	Transitive incompatibility can break builds silently

3.2. Environment Setup with Mixed Java Versions

Support for both Java 8 and Java 17 within the CI/CD pipeline becomes essential because full migration is usually not possible in a single step. Jenkins is a popular automation server that offers mechanisms for managing multiple JDKs and configuring toolchains.

- **Toolchain Configuration**: Multiple JDKs can be set up via Jenkins' global tools configuration. Java versions are mapped to various modules using Maven's toolchains.xml file.
- **Agent Isolation**: Jenkins agents are set up to classify builds by Java version, whether they are running on bare metal or virtual machines. This guarantees reproducibility and prevents environmental contamination.
- **Fallback Environment**: Virtual machines are used as a backup configuration in environments where containerization is not feasible because of security regulations or infrastructure constraints. Version-specific configurations and OS-level isolation are used to maintain these virtual machines.
- **Fallback Strategy:** In the event of breakdowns or incompatibilities, the pipeline has a fallback plan in place to guarantee continuation. Jenkins falls back to VM-based builds if Docker or Kubernetes are unavailable. With pinned dependency versions, these

virtual machines run separate Java 8 and Java 17 environments. This configuration lessens the effect of ecosystem shifts and maintains reproducible builds. Modules can continue to build and test on Java 8 when libraries or JDK features block migration, while others go on to Java 17. This enables incremental modernization and prevents release delays. In order avoid deployment conflicts and maintain traceability, all fallback builds are tagged by version.

This architecture supports gradual refactoring while maintaining legacy components by enabling dual version builds using isolated Jenkins agents and JDK toolchains, as shown in Figure 2.

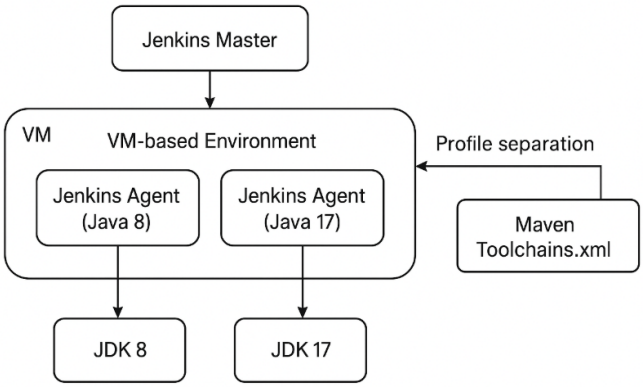

Figure 2: Jenkins Agent and JDK Isolation Architecture

3.3. Pipeline Architecture Design

Conditional branching and per-version customization are supported by the pipeline's modular design. The crucial pipeline phases are:

- **Build Stage**: Compatibility metadata determines which JDK is used to compile modules. Version-specific compilation flags are handled by Maven or Gradle profiles.
- **Test Stage**: Unit tests are executed using the appropriate JUnit versions: JUnit 5 for Java 17 code and JUnit 4 for legacy modules. Parallel job execution and tagging are used to achieve test segregation.
- **Static Code Analysis Stage**: There is integration of tools such as SonarQube, Checkstyle, and PMD [18]. Jenkins pipelines specify quality gates that enforce style compliance and coverage thresholds.
- **Security Scan Stage**: Each pipeline iteration is set up to run vulnerability scanners ,OWASP Dependency-Check and enhanced SpotBugs [17]. Dashboards are updated with the scan reports so that developers can take appropriate action.

For traceability, each of these phases supports customized logging and result archiving. To ensure the consistency across jobs, Jenkins Shared Libraries are utilized. The overall Jenkins-based CI/CD pipeline, illustrating dual-version builds, parallel testing, and integrated quality/security checks, is presented in Figure 3.

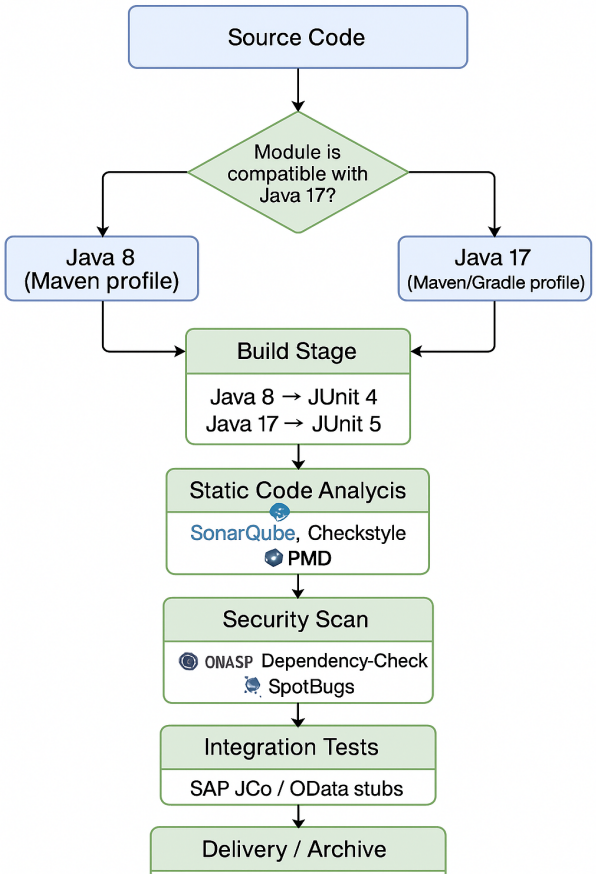

Figure 3: Jenkins CI/CD Pipeline Flow

3.4. Regression Testing and Validation

To make sure that modernization efforts don't introduce functional discrepancies, validation includes thorough regression testing:

- **Functional Tests**: To Verify feature parity run builds on both Java 8 and Java 17. Backward compatibility for end-user functions is guaranteed by regression testing.
- **Locale-Sensitive Validation**: Java 9+ replaces the Compact format with CLDR (Common Locale Data Repository). Locale-mocked test scenarios are used to validate locale-sensitive modules, such as financial reporting, sorting, and date formatting [19].
- **Performance Benchmarking**: Performance metrics like throughput, memory usage, and GC behavior are monitored and compared across Java versions using Java Microbenchmark Harness (JMH). Configurations of ZGC and G1GC are assessed under load.
- **Quality Gates**: If test coverage decreases or if new critical vulnerabilities are discovered, pipelines are set up to fail.

The JUnit plugin and SonarQube dashboards are used to publish test results to Jenkins so that all teams can see them.

3.5. Iterative Refinement and Risk Tracking

A feedback loop is essential to the pipeline design because dependencies, language features, and runtime behavior are always changing:

- **Build Feedback Analysis**: To find patterns and reoccurring problems, build failures are examined. Reports that are automatically generated assist in prioritizing issues pertaining to specific Java versions.
- **Change Logs and Tickets**: Every library upgrade or refactor has a Jira change ticket attached to it. Accordingly, risk scores are updated.
- **Monitoring Tooling Evolution**: Testing tools, JDKs, and Maven plugins are evolving continuously. Jenkins refers to plugin compatibility matrices and performs periodic updates [20].
- **Developer Feedback**: Developers and testers provide feedback during weekly retrospectives, which is then used for improvement of documentation, add validation scripts, and increase test coverage.

Enterprises can modernize Java applications with the least amount of risk and the most automation possible by using this flexible and traceable approach. Using Jenkins-based pipelines, the methodology guarantees that even non-containerized systems can safely migrate to Java 17.

3.6. Limitations of the Methodology

The suggested methodology has certain drawbacks even though it offers a structured and efficient way to manage Java version transitions in enterprise settings using Jenkins-based CI/CD pipelines:

- **Limited Scalability for Complex Polyglot Architectures:** Only Java-based systems are the focus of this methodology. This methodology does not address the additional tooling and coordination mechanisms needed by enterprises with polyglot environments (such as those involving Node.js, Python, or.NET components).
- **Manual Overhead in Risk Classification:** Manual evaluation, domain knowledge, and tool output interpretation are necessary for classifying components into high, medium, or low risk (as shown in Table 1). This procedure can be challenging and subjective, especially when dealing with large legacy codebases that lack adequate documentation.
- **No Support for Containerization:** Because of organizational limitations, the solution is designed for non-containerized environments. The advantages of container-based orchestration, isolation, and reproducibility through Docker/Kubernetes are thus not utilized. Future portability and cloud-native readiness may be limited by this.
- **Dependency Volatility and Ecosystem Lag:** The approach assumes that third-party libraries will eventually become compatible with Java 17. But some essential libraries (like outdated JAXB, OpenSAML, or proprietary SDKs) might not keep up, which could cause pipeline bottlenecks or require temporary forks and patches.
- **Initial Setup Complexity and Learning Curve:** Jenkins internals, Maven profiles, and CI orchestration must be understood to configure multi-version toolchains, Jenkins agents, shared libraries, and conditional pipelines. The initial time and resource commitment may be too much for smaller teams to handle.
- **Restricted Capability to Generalize Beyond Jenkins:** Despite its widespread use, the methodology takes Jenkins to be the CI/CD engine. It would be necessary to re-architect pipelines and modify plugin configurations to port the solution to GitLab CI, Azure DevOps, or GitHub Actions.

4. Results and Evaluation

An internal enterprise-grade Java application was chosen as a representative case study in order to validate the suggested methodology. Using the dual-version Jenkins pipeline outlined in Section 3, the system was gradually moved to Java 17 after being initially developed on Java 8 with Spring Boot 2.x and deployed on Apache Tomcat.

4.1. Performance Gains

The Java Microbenchmark Harness (JMH) was used to simulate production-like load conditions and gather performance benchmarks both before and after the migration. Table 2 provides a summary of the findings.

Table 2: Performance Metrics – Java 8 vs Java 17

Metric	Java 8	Java 17	Improvement
Application Startup Time	5.1 sec	3.5 sec	31% faster
Heap Memory Usage (avg)	480 MB	390 MB	19% less
GC Pause Time (99th perc.)	160 ms	44 ms	72.5% lower
API Throughput (req/sec)	920	1090	18.5% more

4.1.1. Key Observations

- **Application Startup Time**: Java 17's improvements in class data sharing, more effective classloading, and tiered compilation optimizations are largely

responsible for the 31% decrease in startup latency. This is essential for microservices and CI/CD environments, where services are regularly restarted during builds or deployment.

- **Heap Memory Usage**: The average heap memory usage decreased by about 19% in Java 17. JIT compilation optimizations, enhanced object layout, and improved string deduplication are responsible for the smaller memory footprint. Additionally, modules that switched to Java 17 made use of features like records, which naturally lower memory usage by avoiding boilerplate code.
- **GC Pause Time**: A notable 72.5% decrease in GC pause time at the 99th percentile was observed when Java 17 switched from Parallel GC (default in Java 8) to G1GC and optional ZGC. This improvement enhances system responsiveness and user experience, particularly during periods of high load.
- **API Throughput**: Through faster method inlining, better garbage collection scheduling, and a decrease in blocking I/O latencies, REST API throughput increased by 18.5%. Profiling reports showed that Java 17 had less thread contention and fewer full-GC invocations.

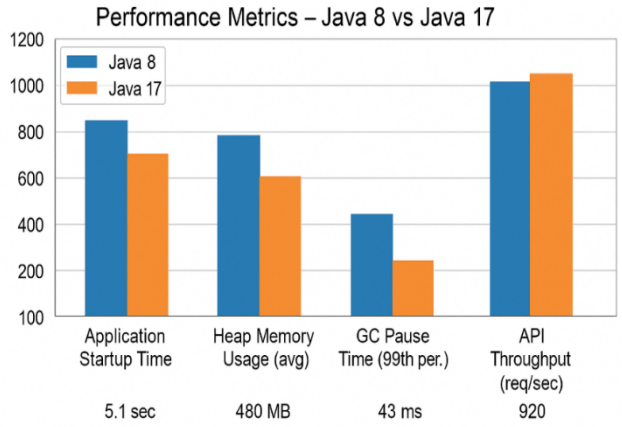

Figure 3: Comparative Performance Benchmark – Java 8 vs Java 17

Figure 3, which contrasts the performance of Java 8 and Java 17 across important metrics, provides a visual summary of these numerical gains.

4.2. Functional Stability

When modernizing enterprise applications, functional stability is a crucial component of success, especially for systems that incorporate internationalization features and are integrated with SAP backends. A thorough regression testing cycle was carried out following the migration from Java 8 to Java 17 to verify compliance, prevent feature regressions, and guarantee backward compatibility.

Jenkins-driven test suites were used to run more than 2,000 automated tests in both environments. The core modules that were tested included:

- **REST API response validation**: Ensuring that identical requests executed with the Java 8 and Java 17 runtimes yield consistent endpoint outputs.
- **Locale-sensitive UI components**: Verifying that dates, currencies, and sorting are rendered correctly across various locale configurations.

The move to CLDR (Common Locale Data Repository) in Java 9+, which was made the default source for locale data in Java 17, presented a significant validation challenge. Dates, currencies, and casing were handled differently because of this modification, especially in UI validation tests. To address these problems and guarantee alignment with user expectations and business requirements, test normalization scripts were added to account for locale-sensitive output variations.

Table 3: Regression Test Summary

Category	Java 8 Pass Rate	Java 17 Pass Rate	Observations
Unit Tests	100%	100%	Fully compatible; no syntax or logic regressions
Integration Tests	98.3%	98.5%	Stable JCo and OData behavior maintained
UI Validation	97.1%	96.8%	Minor locale-related discrepancies resolved

4.2.1. Key Observations

- **Unit Tests**: In both environments, complete compatibility was achieved without any issues. Better test hygiene resulted from the enhanced compiler diagnostics in Java 17.
- **Integration Tests**: Java 17 observed minor enhancements because of improved thread management and quicker request processing.
- **UI Validation**: A closer look showed that the slightly lower initial pass rate in Java 17 was caused by locale format mismatches (e.g., differences in currency symbols, date separators). The behavior was in line with Java 8 outputs after normalization layers were applied. Rendering and frontend logic were found to be flawless.

These findings show that a smooth upgrade to Java 17 without sacrificing functional reliability is possible with a carefully thought-out CI/CD pipeline that includes

isolated environments and automated regression validation.

4.3. Code Quality and Security

The switch to Java 17 offered an opportunity to enhance the security posture and overall code quality in addition to updating the runtime environment. As part of the CI/CD pipeline, automated static analysis and focused dependency remediation were used to achieve this.

SonarQube, SpotBugs, Checkstyle, and OWASP Dependency-Check were among the tools used for performing static analysis. To guarantee that each build was assessed against to an extensive collection of quality and security metrics, these tools were directly incorporated into the Jenkins pipeline.

4.3.1. Key Focus Areas

- Identification and removal of excessively complicated structures and code smells.
- Evaluation of test coverage patterns as new features and modules were added during modernization.
- Libraries with known CVEs (Common Vulnerabilities and Exposures) can be identified through dependency risk scanning.
- Refactoring of outdated or deprecated APIs and removal of error-prone legacy patterns.

Table 4: Code Quality Comparison

Metric	Java 8	Java 17	Change
Code Smells	137	44	-67.8%
Critical CVEs	4	0	-100%
Test Coverage	89.6%	93.2%	+3.6%

4.3.2. Key Observations

- **Code Smells**: After switching to Java 17, a 67.8% decrease in code smells was noted. The introduction of modern language features like records, sealed classes, and switch expressions, which decreased boilerplate and enhanced code clarity, is primarily responsible for this. For example, concise record declarations were used in place of data-carrying POJOs, improving readability and maintainability.
- **Critical CVEs**: Four unfixed CVEs were present in the codebase prior to the migration, two of which were associated with Log4j 1.x and two of which were caused by earlier iterations of OpenSAML. These vulnerable libraries were either patched or swapped out for maintained alternatives as part of the upgrade process. All critical CVEs were fixed by utilizing libraries compatible with Java 17 and conducting transitive dependency audits with OWASP Dependency-Check.
- **Test Coverage**: Test coverage increased by 3.6% as a result of the modernization process. To ensure compatibility, new unit tests were developed for refactored modules, particularly those updated to use modern APIs. Furthermore, parameterized and dynamic tests were made possible by the adoption of JUnit 5, which increased testing depth and decreased redundancy.

4.4. Developer Experience

In addition to technical metrics, developer experience—which is crucial for long-term maintainability and productivity in enterprise environments—was used to evaluate the migration process and the updated CI/CD pipeline.

Twelve developers who actively took part in the modernization effort were surveyed to gather information on this dimension. The survey covered topics like collaboration efficiency, language and tooling preferences, and pipeline usability.

Table 5: Survey Highlights

Question	Agreement (%)
The CI/CD pipeline was easy to use and clearly separated Java versions.	83%
Java 17 features improved code readability and developer productivity.	91%
Shared Jenkins libraries reduced duplication and improved maintainability.	100%

4.4.1. Key Observations

- **Intuitive and Version-Isolated Pipeline**: The modular Jenkins pipeline, which distinguished between Java 8 and Java 17 build/test lanes, was well-received by developers. Teams were able to work on modernization gradually without interfering with legacy behavior due to this version isolation, which also guaranteed confidence during refactoring.
- **Java 17 Developer Ergonomics**: Because of its improved language features—like records, sealed classes, pattern matching, and better switch expressions—Java 17 was strongly preferred. Developers identified improved IDE code assistance (particularly in IntelliJ IDEA ≥ 2021.2), less boilerplate, and cleaner business logic as critical elements. Records made it easier to create domain models and DTOs, which reduced cognitive load and saved time.
- **Impact of Shared Libraries in Jenkins**: Pipeline maintenance effort was significantly reduced as a result of the implementation of Jenkins Shared Libraries. Several modules shared common stages (build, test, scan, and report) that were codified once.

Developers observed fewer configuration bugs, consistent error handling, and quicker onboarding of new team members. Additionally, this method made pipeline-as-code governance possible, guaranteeing adherence to enterprise build guidelines.

5. Discussion

Even in environments without Docker/Kubernetes, the migration strategy confirmed that enterprise Java version transitions could be managed with CI/CD. Important observations are covered below.

5.1. Risk-Driven Planning

As previously mentioned in Table 1, the implementation of risk-based classification was a key component of the migration's success. Each component of the modernization process was evaluated for potential impact, complexity, and criticality to production workflows rather than being treated as a single, homogenous task. Phased, parallelized workstreams and better resource allocation were made possible by this detailed assessment. The following were the main results of the risk-driven planning approach:

- **Early Mitigation of Critical Issues**: OpenSAML and other high-risk elements were handled up front. Because the XML parsing logic was closely linked with Java 8 internals and older versions of OpenSAML had CVEs that made them incompatible with more recent JDKs, there were both technical and security issues. The team reduced downstream disruptions and increased trust in the upgraded security stack by separating and upgrading these early.
- **Parallel Execution of Lower-Risk Tasks**: Parallel updates were made to components that were considered medium or low risk, including Hibernate, logging frameworks, and certain utility libraries. As a result, the team was able to advance steadily without delaying important migration milestones. Hibernate modules frequently only needed small configuration adjustments to function with Java 17, freeing up developers to focus on high-impact projects.
- **Reduced Integration Failures**: Because of unanticipated interdependencies and untested scenarios, traditional "big bang" upgrades frequently result in integration bottlenecks. On the other hand, developers were able to test integrations iteratively, especially around API gateways, by tackling the riskiest components first. This prevented last-minute regressions.
- **Effective Communication and Planning**: Project managers and QA teams, among other stakeholders, could easily understand the scope and difficulties of the migration because of risk classification. Prioritizing testing and planning concentrated sprints around high-severity modules were done using the classification.
- **Improved Developer Morale and Confidence**: When changes were divided into smaller, risk-bounded increments, developers expressed greater confidence. Because there was significantly less perceived uncertainty surrounding migration, there were fewer rollbacks and an increase in sprint velocity.

5.2. Dual-Version Pipelines Are Sustainable

Implementing a dual-version Jenkins pipeline that supported both Java 8 and Java 17 environments simultaneously was one of the most significant architectural decisions made during the modernization. This method allowed for progressive migration, lowering risk and guaranteeing business continuity, as opposed to imposing a full and instantaneous upgrade, which is rarely possible in highly integrated enterprise systems.

5.3. Key Benefits and Observations:

- **Concurrent Support for Legacy and Modern Code**: Maven toolchains.xml enabled the build system to choose the proper Java version for each module, and Jenkins agents were set up with separate JDK installations. This allowed teams to gradually introduce Java 17 features in new or refactored code while maintaining and improving existing Java 8 modules. Crucially, this dual support made maintenance easier by eliminating the need for distinct repositories and branching techniques.
- **Non-Disruptive Deprecation of Java 8 Components**: Older components could be safely and gradually deprecated with backward-compatible tests and build logic. Until their Java 17 counterparts were thoroughly examined and verified, legacy modules continued to be used in production. The "all-or-nothing" upgrade constraint that can paralyze development teams—particularly in high-risk enterprise environments like those integrating with SAML-based identity systems—was avoided as a result.
- **Shared Library Reusability and Maintainability**: Jenkins Shared Libraries were essential for enforcing pipeline consistency and cutting down on redundancy. Reusability across Java 8 and Java 17 jobs was made possible by the abstraction of stages like static code analysis, security scanning, and artifact archiving into shared functions. Repetitive edits across dozens of pipelines were eliminated when logic updates (such as moving from Checkstyle 8 to 10 or improving OWASP rules) were distributed centrally.

- **Sustainability Over Multiple Release Cycles**: Over six production release cycles (about nine months) with the dual-version setup in place, the team saw no regressions caused by the pipeline or build failures because of version conflicts. Indicating that the architecture was not only stable but also able to accommodate gradual enhancements over time, test coverage and code quality metrics also showed consistent improvement during this time.
- **Enhanced Developer Experience**: Without changing the environment, developers could build and test in the Java version of their choice. While backward compatibility made sure legacy teams continued to be productive, tooling support (such as IntelliJ IDEA's Java 17 features and static analyzers) promoted early adoption. Teams working on modernization and legacy maintenance were able to collaborate more easily thanks to this flexibility.

6. Conclusion and Future Work

6.1. Conclusion

Enterprise Java applications moving from Java 8 to Java 17 using Jenkins-based CI/CD pipelines in environments without container orchestration platforms like Docker or Kubernetes can be supported by the risk-driven, automation-centric framework this study presented. Maintainability, incremental risk mitigation, and developer empowerment were given top priority in this methodology because it acknowledged that many enterprise systems—operate under stringent infrastructure constraints.

System audit, pipeline architecture design, dual-environment setup, regression and performance validation, and iterative refinement were the five main stages of the suggested methodology. Production stability was maintained while a gradual migration was made possible by this methodical approach.

6.2. Key conclusions drawn from the implementation include

- Technically and operationally, dual-version pipelines are feasible. Java 8 and Java 17 components could be supported simultaneously because of the Jenkins configuration with toolchain isolation. This made it possible to test and develop in parallel, easing the burden on development teams and preventing disruptive, all-at-once upgrades.
- Targeted static analysis tools and compatibility testing were used to successfully upgrade high-risk components, especially OpenSAML and transitive dependencies with known vulnerabilities. Reducing last-minute failures was greatly aided by early risk classification (Table 1, Figure 1).
- Java 17 offered tangible technical benefits, including:
    - 31% faster application startup
    - The average memory footprint is reduced by 19%.
    - Reduced GC pause times by 72.5%
    - An increase in API throughput of about 18.5%, JVM enhancements, modern language features, and better memory management techniques included in newer Java releases (e.g., G1GC, ZGC) enabled these improvements.
- Developer satisfaction and productivity increased significantly. When asked why they preferred Java 17, developers pointed to improved IDE support, the expressive potential of new features like records and pattern matching, and less boilerplate. Additionally, Jenkins Shared Libraries enhanced maintainability across several repositories and reduced duplication.
- Improved security posture: After migration, all known critical CVEs (such as legacy Log4j vulnerabilities) were fixed, and code smells were decreased by almost 70%. Test coverage increased from 89.6% to 93.2%, indicating that modernization is crucial for maintainability and risk mitigation in addition to performance.

The study concludes by offering enterprises a scalable, risk-aware roadmap for updating modern Java versions while navigating difficulties of mixed-version dependencies and legacy infrastructure. It demonstrates how modern CI/CD techniques can speed up digital modernization without compromising system integrity, even in traditional environments.

6.3. Future Work

Although this study offers a solid basis, there are still a number of directions for further investigation:

- **Enhancements to Tooling Automation:** Including AI-powered recommendation engines (like OpenRewrite and Revapi analyzers) to automatically identify or refactor code that is incompatible while conducting audits.
- **Multi-Language Integration**: Expanding the pipeline to handle hybrid applications that combine Java with Kotlin, Scala, or Groovy and determining compatibility under Java 17
- **Legacy API Adaptation Layer**: Designing reusable shims or compatibility wrappers for deprecated or removed APIs, particularly for organizations that cannot yet eliminate legacy modules.
- **Performance Monitoring in Production**: Utilizing tools like Java Flight Recorder (JFR), Prometheus, or Dynatrace to extend benchmarking beyond JMH-based lab tests to continuous profiling in production environments.

- **Longitudinal Migration Studies**: Measuring long-term cost savings, technical debt reduction, and velocity improvements by collecting migration metrics across several release cycles or departments.
- **Container Readiness Roadmap**: A future extension might specify a phased plan to advance such CI/CD pipelines toward container-based deployments using Docker, Kubernetes, or SAP BTP, even though this study focused on non-containerized systems.

Organizations can further optimize their modernization journeys, lower operational risk, and prepare for future Java LTS releases by focusing on these future directions.

References

[1] Oracle,*Java SE Support Roadmap*, Oracle, 2021. [Online]. Available: https://www.oracle.com/java/technologies/java-se-support-roadmap.html

[2] I. Deligiannis, et al., "Challenges in Modularizing Legacy Java Systems: An Empirical Study," *Empirical Software Engineering*, vol. 26, no. 2, p. 25, 2021.

[3] OpenJDK, *JEP Index*, 2021. [Online]. Available: https://openjdk.org/jeps/

[4] A. Shah, et al., "Risks in Transitive Dependency Upgrades in Java Projects," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 27–36, 2020.

[5] Jenkins Documentation, *Pipeline Syntax and Tools*, Jenkins, 2023. [Online]. Available: https://www.jenkins.io/doc/

[6] M. Gupta and A. Saxena, "An Empirical Study of Java LTS Versions in Enterprise Software Systems," *Journal of Software Engineering and Applications*, vol. 13, no. 8, pp. 325–337, 2020.

[7] Oracle, *Java SE Support Roadmap*, Oracle, 2023. [Online]. Available: https://www.oracle.com/java/technologies/java-se-support-roadmap.html

[8] N. Deligiannis, Y. Smaragdakis, and S. Chatrchyan, "Migrating to Java 9 Modules: Lessons from the Trenches," *Proceedings of the ACM on Programming Languages*, vol. 3, OOPSLA, pp. 1–25, 2019.

[9] G. Venkat and T. Saito, "Modern Java Language Features: From Java 9 to Java 17," *Java Magazine*, Oracle, Sept. 2022.

[10] L. Chen and M. Thakkar, "Garbage Collection Optimization in Large-Scale Java Applications," in *Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution*, pp. 280–289, 2021.

[11] S. Malhotra, "Dependency Management for Java Frameworks: The Case of Spring and Jersey," *International Journal of Software Engineering & Applications*, vol. 12, no. 4, pp. 45–57, 2021.

[12] P. Shah, A. Reddy, and J. Ma, "Risk Propagation in Java Dependency Trees: A Transitive Analysis Approach," *Software: Practice and Experience*, vol. 52, no. 9, pp. 1754–1772, 2022.

[13] Jenkins Project, *Jenkins Documentation: Pipeline and Plugin Ecosystem*, Jenkins, 2024. [Online]. Available: https://www.jenkins.io/doc/

[14] B. Tomlinson, "CI/CD without Containers: Lessons from Legacy Environments," in *Proceedings of the DevOps Enterprise Summit*, 2021.

[15] N. Deligiannis, D. Spinellis, and G. Gousios, "Analyzing Modularity in Java Projects After JPMS Adoption," *Empirical Software Engineering Journal*, vol. 27, no. 1, pp. 1–29, 2022.

[16] C. Pereira, R. Nascimento, and J. Souza, "API Deprecation in Enterprise Software: A Case Study on Java EE Migration," in *Proceedings of the 2020 ACM/IEEE International Symposium on Empirical Software Engineering (ESEM)*, pp. 190–199, 2020.

[17] OWASP Foundation, *OWASP Dependency-Check*, 2023. [Online]. Available: https://owasp.org/www-project-dependency-check/

[18] SonarSource, *SonarQube Documentation*, SonarQube, 2024. [Online]. Available: https://docs.sonarsource.com/

[19] Oracle Corporation, *CLDR in JDK 9 and Later (JEP 252)*, Oracle, 2021. [Online]. Available: https://openjdk.org/jeps/252

[20] Oracle Corporation, *Java Microbenchmark Harness (JMH)*, Oracle, 2022. [Online]. Available: https://openjdk.org/projects/code-tools/jmh/