UNIVERSITÀ DEGLI STUDI DI SALERNO

DEPARTMENT OF COMPUTER SCIENCE

Software Dependability

Dependability Project – whiTee

Repository: https://github.com/rosacarota/SD-Progetto

Technical Report

Professor:
Prof.
Dario Di Nucci

Students:
Antonio D'Auria
Std. ID: NF22500063
Rosa Carotenuto
Std. ID: 0522502024
Luca Casillo
Std. ID: NF22500037

ACADEMIC YEAR 2025/2026

Contents

List of Figures

1 Context of the Project

1.1 Description of the Original Project

whiTee is a Java-based web application developed for the *Tecnologie Software per il Web* course as an academic project. The system is structured around a simple **e-commerce scenario** focused on the sale and customization of t-shirts.

The application supports different **categories of users**. Guest users can browse the product catalog, view product details, customize items, manage a shopping cart, and register an account. Registered users can finalize purchases, access their order history, manage personal information, and provide feedback on purchased products. An **administrative role** is also supported, allowing the management of products and customer orders through a dedicated interface.

These functionalities define a **complete but contained application scope**, including user management, transactional workflows, and data persistence.

1.2 Project Scope and Course Context

The project is developed within the framework of the *Software Dependability* course and consists in the application of the techniques introduced during the lectures to an existing software system. The course provides a **methodological perspective** on software dependability, covering specification, testing, performance analysis, security assessment, and operational aspects.

In this context, the application is analyzed as a **completed system** rather than as an evolving product. The scope of the project is therefore limited to the **evaluation and assessment** of the system's dependability properties.

2 Goals of the Project

The analysis addresses multiple and complementary aspects of dependability, covering the software lifecycle from specification to delivery.

Fault avoidance is tackled through the introduction of formal specifications for selected core components, with the aim of making expected behavior explicit and verifiable. **Fault detection and assessment** are addressed through an extensive testing campaign, combining unit testing, structural coverage analysis, and mutation testing to evaluate both test completeness and effectiveness. **Performance-related aspects** are analyzed by means of targeted microbenchmarks, focusing on computationally relevant portions of the codebase.

Security and data protection are addressed by introducing mechanisms for password hashing and cryptographic protection of sensitive data, as well as by performing vulnerability analysis through static analysis tools and dependency scanning techniques.

Finally, **operational reliability** is supported through the adoption of a structured build and dependency management process based on Maven, together with containerization and automated CI/CD pipelines. The use of Docker allows the application and its dependencies to be packaged into a consistent execution environment, reducing configuration-related issues and improving reproducibility. This setup enables controlled dependency resolution, reproducible builds, continuous verification, and the enforcement of quality gates throughout the development workflow.

All techniques are applied to the existing codebase and infrastructure without modifying the system's functional requirements, in order to evaluate the dependability properties of the application as it was originally designed.

3 Methodological Steps

This section documents the methodological steps followed during the project, describing how the techniques and tools selected for the dependability analysis are applied to the system.

The section is organized around distinct **methodological areas**, each addressing a specific aspect of the analysis. Depending on the case, a subsection may describe a single technique or a set of related tools used to support a given methodological approach.

For each methodological step, the description focuses on the **operational setup**, the **integration within the application or development workflow**, and the **scope of application** within the system. The discussion is limited to how the techniques and tools are configured and applied, without anticipating the outcomes of the analyses.

The results produced by the application of the described methodologies, together with the corresponding observations, are presented and discussed in section 4.

3.1 Password Hashing & Data Cryptography

The original version of the application did not include protection mechanisms for sensitive user data. User passwords were stored in clear text within the database, and billing information was persisted without any form of cryptographic protection.

As part of the dependability-oriented revision of the system, mechanisms for **password hashing** and **data cryptography** were introduced. This methodological step aims at improving the confidentiality of sensitive data stored at rest, reducing the risks associated with unauthorized access to the database.

Password hashing. Password management has been revised by introducing a hashing-based approach. User passwords are processed using the **BCrypt** algorithm before being stored, ensuring that authentication credentials are never persisted in a reversible form.

BCrypt incorporates an adaptive cost factor and automatic salting, allowing the hashing process to be computationally expensive and resistant to brute-force and precomputed dictionary attacks. As a result, password verification can be performed securely while preventing direct recovery of the original values, even in the event of database exposure.

Encryption of billing data. Sensitive billing information, such as credit card details and related fields, is protected through symmetric encryption. The application adopts an **AES-based** encryption scheme, encapsulated within a dedicated utility component responsible for encryption and decryption operations. These operations are performed at the data access layer, allowing sensitive values to be stored in encrypted form while remaining transparently usable by the application logic.

The cryptographic key used for encryption is supplied at runtime through a **configuration provided via environment variables** and injected into the application at startup.

Overall, this step introduces a systematic handling of sensitive credentials and billing data, strengthening the security posture of the application without altering its functional behavior.

3.2 Build and Dependency Management

Build automation and dependency management are handled through **Apache Maven**, which is used as the central tool for compiling, testing, packaging, and verifying the application. The adoption of Maven provides a structured and declarative approach to dependency resolution and build configuration, ensuring consistency across development and testing.

Project dependencies are explicitly declared and versioned, covering both runtime components (such as database connectors and third-party libraries) and testing frameworks. This setup allows controlled dependency resolution and simplifies vulnerability analysis and maintenance activities, as all external components are managed through a single configuration file.

The build process integrates several quality-related plugins as part of the standard Maven lifecycle. Unit tests are executed automatically during the test phase, while code coverage and mutation testing are enforced during the verification phase through dedicated plugins and coverage thresholds. By embedding these checks into the build process, the project ensures that quality constraints are continuously validated and that violations are detected early.

The application is packaged as a **WAR artifact**, suitable for deployment within a servlet container. This choice reflects the original architecture of the project and allows the same build artifact to be reused across local execution, containerization, and CI/CD pipelines.

Separation of performance benchmarks. Performance benchmarks based on JMH are managed in the **dedicated jmh development branch** of the repository. In this branch, the project build configuration is extended with additional dependencies and plugins required for microbenchmark execution.

Keeping JMH support isolated from the main branch avoids introducing benchmark-specific complexity into the standard build configuration. This separation ensures that the default build process remains focused on application compilation, testing, and verification, while allowing performance analysis to be conducted independently without affecting build reproducibility or CI workflows.

3.3 Docker & Docker Hub

The application has been containerized using Docker with the objective of ensuring a **reproducible**, **isolated**, and **consistent** execution environment. The use of containers also reduces reliance on heterogeneous local configurations.

Dockerfile and application containerization. Application containerization has been implemented through a Dockerfile based on a **multi-stage build**, which separates the build phase from the runtime environment.

(1) **Build stage.** In the first stage, the application is compiled using Maven, producing a `WAR` file. This approach isolates the build process and prevents compilation tools and build-time dependencies from being included in the final image.

(2) **Runtime stage.** In the second stage, the application is executed within a lightweight Tomcat container, where the generated `WAR` is deployed. Default Tomcat applications are removed, reducing the **attack surface** and simplifying the runtime environment.

This separation between build and runtime improves both **image efficiency** and **maintainability**.

Database access configuration is externalized through the `context.xml` file, which defines a JNDI DataSource within the Tomcat container.

Service orchestration with Docker Compose. To support the execution of the application within a complete environment, Docker Compose is used to orchestrate the services required for system operation. The configuration defines two main services: a **MySQL database service** and a **Tomcat-based application service**.

The database service is configured with **persistent volumes** to ensure data durability and includes a **healthcheck mechanism** that verifies database availability before the application is started. Both services adopt an **automatic restart policy**, improving the resilience of the environment in the presence of unexpected failures.

The application service is configured to start only when the database is operational and is parameterized through environment variables.

Docker image publishing. The application Docker image is made available through a **public registry**, allowing the reuse of a containerized artifact. The image is built from the Dockerfile and supports **multiple architectures**, increasing the portability of the application across heterogeneous environments.

Within the context of the project, image publishing is intended to provide an execution-ready container artifact and does not correspond to an actual release or deployment to a production environment.

In this context, a dedicated **override file** is provided for local development, allowing the image to be built directly from the source code rather than relying on a pre-built image. This approach enables the reuse of the same base configuration while adapting it to different execution contexts.

3.4 JML

In this project, JML has been introduced as a **fault avoidance** mechanism. The goal was to improve system robustness by identifying and addressing potential errors before they manifest at runtime. Rather than aiming for full formal verification, JML was adopted in a pragmatic way, taking into account the existing codebase and its architectural constraints.

The use of JML was therefore **selective**, focusing only on those parts of the system where formal specification could provide a concrete benefit.

Application to Bean classes. Beans represent the core of the application state: they model domain entities and act as the interface between the Control and DAO layers. As such, they naturally encode a set of domain constraints that must always hold to preserve system consistency.

Applying JML to Bean classes made these constraints explicit through the use of class invariants. In particular, JML specifications were used to guarantee that:

- identifiers and prices never assume negative values;
- tax rates remain within a valid range;
- textual fields respect predefined maximum lengths.

As a result, each valid instance of a Bean represents a semantically consistent state, independently of where it is used within the system.

Management of `null` *values.* During the annotation of Bean classes, special attention was given to the handling of `null` values. In the project, Beans may be instantiated and populated incrementally, especially during interactions with the presentation layer.

For this reason, several fields were explicitly declared as nullable in the JML specifications.

Exclusion of DAO classes. The application of JML to DAO classes was initially considered but ultimately discarded. DAO components interact directly with the database and rely on JDBC connection and SQL queries.

Experiments conducted with OpenJML showed that annotating DAO classes led to numerous static verification failures. These issues were not due to logical flaws in the code, but to the inherent limitations of formal verification when dealing with I/O operations and non-deterministic behavior.

For this reason, JML was deliberately not applied to DAO components. Their verification was instead delegated to more appropriate techniques, such as testing.

Static Verification with OpenJML. The JML specifications defined on Bean classes have been analyzed using OpenJML in **ESC** mode, adopting it as a static analysis tool. This mode enables the automatic verification of consistency between specifications and implementation, checking that class invariants, preconditions, and postconditions are not violated.

In some cases, in order to avoid excessive verification complexity, it was necessary to exclude non-critical methods, such as `toString()`, which do not affect object state but may generate computationally expensive analyses.

3.5 Testing

In the project, testing represents the primary mechanism for *fault detection*, complementing the *fault avoidance* techniques adopted in the earlier phases. The objective is to verify the correct functioning of the system and to observe the behavior of the most critical components under different inputs, through a **systematic** and **reproducible** test suite.

Testing activities, which were absent in the initial version of the project, have been introduced and developed as part of this work. They focus on the **application layer** and the **data access layer**, where most of the functional logic resides. All tests are executed automatically through Maven, using JUnit 5 as the reference testing framework and Mockito to support dependency mocking.

Starting from the test suite, additional **quantitative analyses** have been applied, including code coverage measurement and performance testing. These analyses aim to provide a more comprehensive characterization of test effectiveness and of the behavior of system components under load.

3.5.1 Unit Testing. The project includes exclusively **unit tests**, designed to verify the behavior of individual code units in isolation. Tests have been developed by applying the **Category Partition Testing** technique, which enables a systematic identification of relevant input classes, invalid cases, and boundary conditions.

Unit testing activities have been primarily focused on **Servlets** and **DAO** components, as they contain the largest portion of application logic. Servlets have been tested by simulating HTTP requests and responses, while DAO components have been verified by isolating access to external resources.

To ensure proper **test isolation**, extensive use of **Mockito** has been made to mock dependencies such as HTTP request and response objects or database access components.

Bean classes have not been tested directly, as they implement purely structural logic based on getters and setters. Their behavior is nonetheless exercised indirectly by tests targeting Servlets and DAOs that make use of them, making dedicated unit testing of Beans redundant within the scope of the project.

All tests are executed automatically through Maven during the build phase and constitute the **baseline** for subsequent analyses.

3.5.2 Code Coverage – JaCoCo & Codecov. A systematic measurement of **code coverage** has been introduced in the project with the aim of **continuously monitoring the effectiveness of the test suite**. Coverage is used as a **supporting metric for test suite evaluation**, enabling the identification of reductions in test adequacy and potential regressions introduced by code changes.

Coverage quality gate. The collection of code coverage metrics is integrated into the Maven build process through the JaCoCo Maven plugin. To make coverage measurement **operational rather than descriptive**, minimum coverage thresholds are enforced directly during the build. Specifically, a **minimum line coverage of 80%** and a **minimum branch coverage of 80%** are required. If these thresholds are not met, the build fails during the `verify` phase, explicitly signaling **insufficient test coverage**.

Exclusions and rationale. Since the project is a fork and was not originally designed with **testability and coverage measurement** in mind, the codebase includes heterogeneous components that do not contribute meaningfully to the evaluation of **test suite effectiveness**. For this reason, **targeted exclusions** have been applied in order to avoid poorly informative coverage metrics.

Specifically, data population packages (`model/populator`), selected infrastructural classes (such as database connection management and cryptographic key providers), and several **purely structural** Bean classes have been excluded. In addition, the contents of `src/main/webapp` are excluded, as they are not relevant to Java code coverage.

Exclusions have been kept **consistent across JaCoCo, PiTest and SonarQube** to ensure report alignment and to avoid discrepancies in coverage analysis.

Codecov. To complement local coverage enforcement, code coverage data produced by JaCoCo is published to **Codecov** as a centralized reporting platform. Codecov provides a consolidated view of coverage metrics, enabling inspection at different levels of granularity, including per-file coverage and historical trends.

The platform supports the comparison of coverage values across revisions, making variations in test coverage immediately visible during development. By offering continuous and centralized access to coverage information, Codecov improves the observability of test adequacy and facilitates the identification of coverage regressions over time.

Together with build-time enforcement, the use of Codecov contributes to maintaining code coverage as a **stable and monitored quality attribute** of the project.

3.5.3 Mutation Testing - PiTest. Mutation testing is adopted as part of the testing strategy to evaluate the robustness of the existing test suite beyond structural coverage metrics. The analysis is performed using **PiTest**, which is integrated into the Maven build process and executed automatically during the verification phase.

Mutation analysis is applied to the main application logic, with a focus on control and model components. **As in the structural coverage analysis**, the same classes that do not encode relevant behavioral logic—such as data beans, utility classes, and infrastructure-related components—are explicitly excluded in order to concentrate the analysis on fault-relevant code.

Mutation quality gates. Mutation testing is enforced through **quality gates** integrated into the build configuration. The build is configured to fail if the mutation analysis does not satisfy predefined minimum thresholds, ensuring that mutation testing is treated as a mandatory verification step rather than as an optional assessment.

In particular, the following constraints are enforced:

- a minimum **mutation score** of 50%;
- a minimum **mutation coverage** of 80%.

By integrating these constraints into the standard build workflow, mutation testing becomes an integral part of the continuous verification process.

3.5.4 Performance Testing - JMH. In this project, performance evaluation was conducted through microbenchmarking using **JMH (Java Microbenchmark Harness)**. The intent was to obtain a baseline characterization of component-level performance, rather than to execute system-wide load or stress tests.

Project integration and execution workflow. Benchmark code was placed under a dedicated `jmh` folder at the project root. Execution was automated through a dedicated script, `run_jmh.sh`, which runs microbenchmarks starting from the JMH-generated artifact (`benchmarks.jar`) and collects results in a structured form. Each benchmark is executed by specifying the corresponding benchmark class, producing a separate output directory for each analyzed component. This approach ensures repeatability and makes runs comparable over time.

Collected artifacts. For each executed benchmark, the workflow produces an output set composed of:

- **`jmh.log`**, containing the full benchmark execution and the summary table;
- **`results.json`**, containing the structured benchmark results;
- **`metadata.txt`**, containing information about the execution environment (e.g., CPU/OS/Java version) and the benchmark configuration.

These artifacts provide a reliable basis for micro-level performance analysis and for tracking performance evolution across subsequent runs.

Benchmark targets and scope. The scope of microbenchmarking was deliberately selective, focusing on components for which JMH measurements are meaningful and stable, such as **security utilities**, **in-memory application logic**, and small helper routines (e.g., data transformation and serialization tasks). Conversely, code paths dominated by external dependencies—including database access, filesystem interaction, and servlet-container behavior—were not treated as primary microbenchmark targets, since their performance is strongly influenced by environmental variability and is more appropriately evaluated with higher-level performance tests.

3.6 Security Mechanism

The security of the project has been analyzed through a set of **automated tools** with the objective of identifying **vulnerabilities**, **insecure configurations**, and potentially risky development practices.

The adopted tools address security from **complementary perspectives**, enabling a broader and more comprehensive assessment of potential risks. In particular, the analysis considers vulnerabilities related to **application code**, **third-party libraries**, the **execution environment**, and **secret management**.

3.6.1 SonarQube. SonarQube is used to perform **static analysis** of the source code, with particular attention to **security-related aspects**. The analysis enables the identification of known vulnerabilities, security hotspots, and code patterns that may introduce risks if used improperly.

Through static analysis, SonarQube supports the **early detection** of security issues and design weaknesses, allowing corrective actions to be taken before such problems manifest at runtime. In addition, the analysis contributes to improving **overall code quality** by highlighting development practices that are not recommended.

3.6.2 Snyk. Snyk is used to analyze the security of the project across **multiple layers**. First, static analysis of the source code is performed to identify **vulnerabilities** and insecure programming practices.

In addition to application code, Snyk is employed to analyze the project's **external dependencies**, enabling the detection of known vulnerabilities in third-party libraries.

Finally, security analysis is extended to the application's **Docker image**, allowing the identification of potential issues related to the **base image** or the **execution environment**. All analyses are configured to focus on **higher-severity vulnerabilities**, reducing noise from low-impact findings and improving the relevance of the reported results.

3.6.3 Dependabot. Dependabot is adopted as a complementary mechanism for **dependency maintenance and security hardening**. Its role is to continuously monitor the project's declared dependencies and to identify available updates that address known vulnerabilities or introduce important security fixes.

When relevant updates are detected, Dependabot automatically proposes changes through pull requests, allowing dependency upgrades to be reviewed and integrated in a controlled manner. This approach supports proactive dependency management and reduces the risk associated with outdated or vulnerable third-party components.

3.6.4 GitGuardian. GitGuardian is used to detect the presence of **secrets** and sensitive information within the repository. The analysis aims to identify the accidental exposure of credentials, API keys, or other confidential data that could compromise system security.

This type of control addresses a **distinct class of issues** compared to source code or dependency analysis, focusing instead on **development practices** and **configuration management**. As a result, the security analysis extends beyond functional aspects of the software to include risks related to the handling of sensitive information.

3.7 CI/CD Pipeline

The project adopts a structured **CI/CD pipeline** implemented using GitHub Actions, designed to automate verification, quality assessment, security analysis, and container delivery. The pipeline plays a central role in enforcing dependability requirements by integrating testing, analysis tools, and quality gates directly into the development workflow.

The CI/CD process is organized into **three coordinated workflows**, each responsible for a specific phase of verification and delivery, and executed in a strictly ordered manner.

Build, testing, and quality verification. The primary workflow is triggered on every push and pull request to the `main` branch. It builds the project using **Maven** and executes the full verification lifecycle, including unit testing and mutation testing.

Structural coverage analysis is performed using **JaCoCo**, with coverage thresholds enforced during the `verify` phase. The build fails if line or branch coverage drops below the configured limits, ensuring that insufficiently tested changes are rejected early. In parallel, **PiTest** is executed to assess test effectiveness through mutation analysis. Mutation testing is subject to dedicated **quality gates**, enforcing minimum thresholds on both mutation score and mutation coverage.

Both JaCoCo and PiTest generate HTML reports, which are automatically collected and published on **GitHub Pages**. This ensures public availability and traceability of coverage and mutation results across builds. In addition, coverage data and test results are uploaded to **Codecov**, enabling centralized visualization and historical tracking.

Test execution results are summarized using **dorny/test-reporter**, improving feedback visibility within the GitHub interface. At the end of the workflow, the application is packaged as a WAR artifact and stored as a build artifact for reuse in subsequent stages.

After a successful build and test phase, a dedicated job performs **SonarQube Cloud** analysis. The analysis reuses the compiled classes and JaCoCo coverage data produced earlier, and enforces the Sonar quality gate on new code, covering bugs, vulnerabilities, technical debt, and coverage requirements.

Security analysis. A second workflow is dedicated to security assessment and is executed only after the successful completion of the main CI pipeline, or manually when required. This workflow integrates multiple security-oriented tools, including **GitGuardian** for secret leakage detection and **Snyk** for vulnerability analysis.

Snyk is used both for static application security testing and for dependency scanning of the Maven project. To extend security checks to the execution environment, the workflow builds a Docker image using configuration generated from secret-backed templates and performs a container vulnerability scan. This separation ensures that security analysis acts as a mandatory gate before delivery, without interfering with standard build verification.

Container build and delivery. The final workflow is responsible for container delivery and is triggered only after the successful execution of the security pipeline. During this stage, the application is rebuilt in a production-ready configuration and packaged into a Docker image.

The workflow uses **Docker Buildx** to produce a **multi-architecture image** compatible with both `amd64` and `arm64` platforms. Image metadata and tags are generated automatically, and the final image is published to Docker Hub. By constraining publication to artifacts that have passed all previous quality and security gates, the pipeline enforces a strict delivery policy aligned with dependability objectives.

Public reports and dashboards. To support transparency and reproducibility, the artifacts produced by the CI/CD pipeline are made publicly accessible through dedicated reporting platforms. The following resources provide direct access to the generated reports and dashboards:

- **Codecov Coverage Dashboard:** https://app.codecov.io/gh/rosacarota/SD-Progetto
- **JaCoCo Coverage Reports:** https://rosacarota.github.io/SD-Progetto/Jacoco/
- **PiTest Mutation Reports:** https://rosacarota.github.io/SD-Progetto/PiTest/

4 Results

This section presents the results obtained from the application of the methodological steps described in the previous chapter. For each technique and tool, the outcomes are reported and discussed with the objective of highlighting the effects of the introduced interventions on the system's dependability.

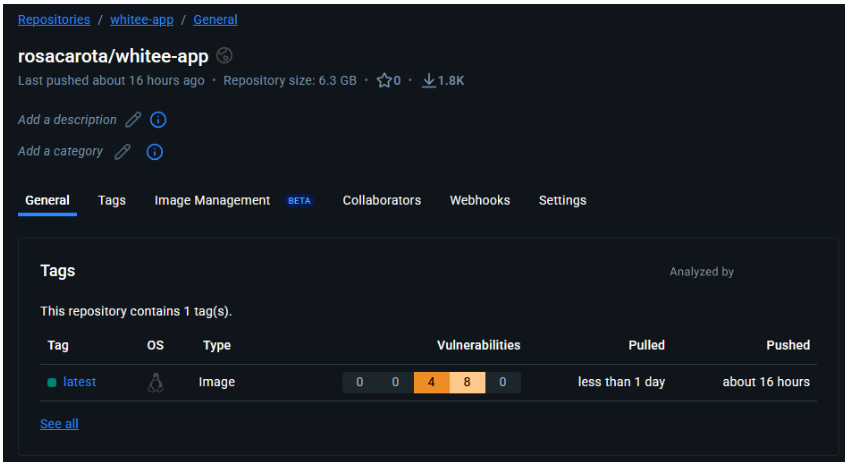
Results are organized following the same structure adopted in the methodological description. When applicable, results are supported by empirical evidence such as analysis reports, tool outputs, and visual artifacts, enabling an objective interpretation of the observed effects.

Rather than providing a detailed explanation of the employed tools, the focus of this section is on the **observable impact** of the adopted techniques. This includes improvements in **security posture**, **test effectiveness**, and overall **system characterization**. Residual issues and limitations are explicitly discussed when relevant.

4.1 Docker Results

As a result of the containerization process described in the methodological section, the application is made available as a Docker image published on a public container registry.

The published image is associated with a tagged version and is automatically analyzed by the container registry through an integrated **vulnerability scanning** mechanism. This enables continuous inspection of the security status of the containerized artifact.


Repositories / whitee-app / General

rosacarota/whitee-app

Last pushed about 16 hours ago · Repository size: 6.3 GB · 0 · 1.8K

Add a description

Add a category

Tags

Analyzed by

This repository contains 1 tag(s).

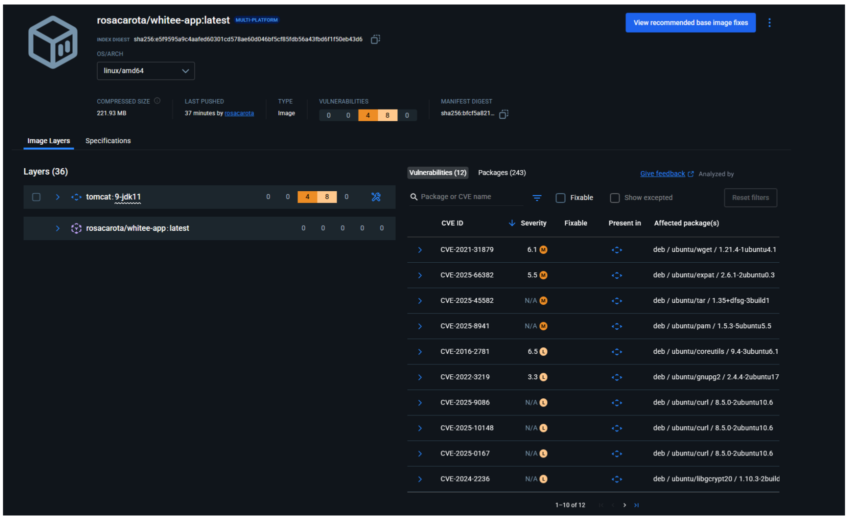
Tag	OS	Type	Vulnerabilities	Pulled	Pushed
latest		Image	0 0 4 8 0	less than 1 day	about 16 hours

See all

Fig. 4.1. Publication of the application Docker image on Docker Hub.

The vulnerability scanned a limited number of known security issues (CVE) associated with the analyzed container image. A detailed inspection shows that these vulnerabilities originate **exclusively from the underlying base image** used for the runtime environment, rather than from the application code or from dependencies introduced during the build process.

In particular, the reported vulnerabilities affect standard operating system packages included in the Linux distribution underlying the Tomcat base image, such as utilities (wget) and related system libraries. No vulnerabilities are associated with the **application layer** or with the artifacts produced during the build stage of the Dockerfile. Additionally, the identified vulnerabilities have **no known exploits**.



rosacarota/whitee-app:latest MULTI-PLATFORM

INDEX DIGEST sha256:e5f9595a9c4aafed60301cd578ae60d046bf5cf85fdb56a43fbd6f1f50eb43d6

OS/ARCH linux/amd64

COMPRESSED SIZE	LAST PUSHED	TYPE	VULNERABILITIES	MANIFEST DIGEST
221.93 MB	37 minutes by rosacarota	Image	0 0 4 8 0	sha256:bfcf5a821...

Image Layers | Specifications

Layers (36)

tomcat:9-jdk11	0	0	4	8	0
rosacarota/whitee-app:latest	0	0	0	0	0

Vulnerabilities (12) | Packages (243) | Give feedback | Analyzed by

CVE ID	Severity	Fixable	Present in	Affected package(s)
CVE-2021-31879	6.1 M			deb / ubuntu/wget / 1.21.4-1ubuntu4.1
CVE-2025-66382	5.5 M			deb / ubuntu/expat / 2.6.1-2ubuntu0.3
CVE-2025-45582	N/A M			deb / ubuntu/tar / 1.35+dfsg-3build1
CVE-2025-8941	N/A M			deb / ubuntu/pam / 1.5.3-5ubuntu5.5
CVE-2016-2781	6.5 L			deb / ubuntu/coreutils / 9.4-3ubuntu6.1
CVE-2022-3219	3.3 L			deb / ubuntu/gnupg2 / 2.4.4-2ubuntu17
CVE-2025-9086	N/A L			deb / ubuntu/curl / 8.5.0-2ubuntu10.6
CVE-2025-10148	N/A L			deb / ubuntu/curl / 8.5.0-2ubuntu10.6
CVE-2025-0167	N/A L			deb / ubuntu/curl / 8.5.0-2ubuntu10.6
CVE-2024-2236	N/A L			deb / ubuntu/libgcrypt20 / 1.10.3-2build

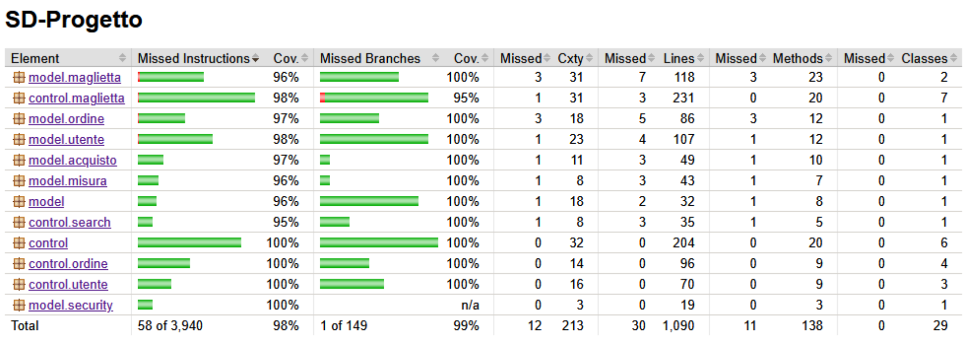
1–10 of 12

Fig. 4.2. Docker image vulnerability scan results.

These residual vulnerabilities were not addressed within the scope of the project. Their mitigation would require modifications to the base image or the adoption of a custom operating system layer, which fall outside the objectives of the conducted in this work.

4.2 Testing and JaCoCo Results

JaCoCo results confirm that the implemented unit test suite achieves a **high degree of structural coverage** on the analyzed production code. As shown in Figure 4.3, the overall coverage reaches **98% instruction coverage** and **99% branch coverage**, exceeding the minimum thresholds enforced through the build quality gate.



SD-Progetto

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
model.maglietta		96%		100%	3	31	7	118	3	23	0	2
control.maglietta		98%		95%	1	31	3	231	0	20	0	7
model.ordine		97%		100%	3	18	5	86	3	12	0	1
model.utente		98%		100%	1	23	4	107	1	12	0	1
model.acquisto		97%		100%	1	11	3	49	1	10	0	1
model.misura		96%		100%	1	8	3	43	1	7	0	1
model		96%		100%	1	18	2	32	1	8	0	1
control.search		95%		100%	1	8	3	35	1	5	0	1
control		100%		100%	0	32	0	204	0	20	0	6
control.ordine		100%		100%	0	14	0	96	0	9	0	4
control.utente		100%		100%	0	16	0	70	0	9	0	3
model.security		100%		n/a	0	3	0	19	0	3	0	1
Total	58 of 3,940	98%	1 of 149	99%	12	213	30	1,090	11	138	0	29

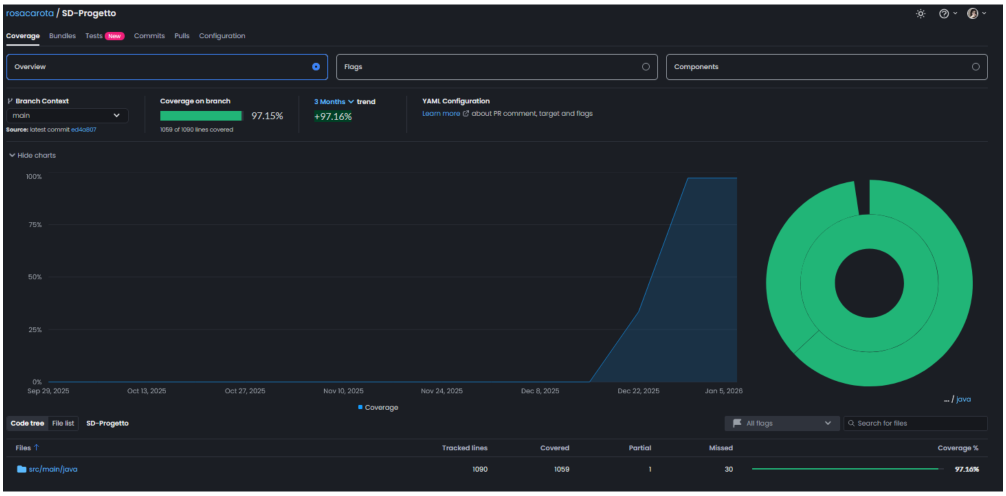
Fig. 4.3. JaCoCo coverage report with instruction and branch coverage results by package.

The package-level breakdown highlights a generally uniform coverage distribution across the codebase segments considered relevant for behavioral assessment. Most `control.*` packages reach **full coverage** on both instructions and branches, reflecting the fact that servlet logic has been systematically exercised through request/response-based unit tests. Similarly, the `model.*` packages show consistently high coverage, indicating that data access logic has been tested under the main execution paths and relevant boundary conditions.

Residual uncovered instructions are limited and concentrated in a small subset of components, and do not affect the satisfaction of the configured quality gate. These gaps can largely be attributed to defensive branches and exceptional paths that are difficult to trigger reliably in isolated unit tests, or that depend on external conditions not reproduced in the test environment. Overall, the achieved coverage values support the conclusion that the test suite provides broad structural exploration of the application logic included in the analysis scope.

4.3 Codecov Results

As shown in the Codecov dashboard, code coverage on the main branch reaches a value of **97.15%**, with **1059 covered lines out of 1090 tracked**. This result confirms the achievement of a **high and stable level of coverage**, well above the minimum thresholds defined during the verification process.



Branch Context: main
Source: latest commit ed4a807

Coverage on branch: 97.15%
1059 of 1090 lines covered

3 Months trend: +97.16%

YAML Configuration
Learn more about PR comment, target and flags

Files	Tracked lines	Covered	Partial	Missed	Coverage %
src/main/java	1090	1059	1	30	97.16%

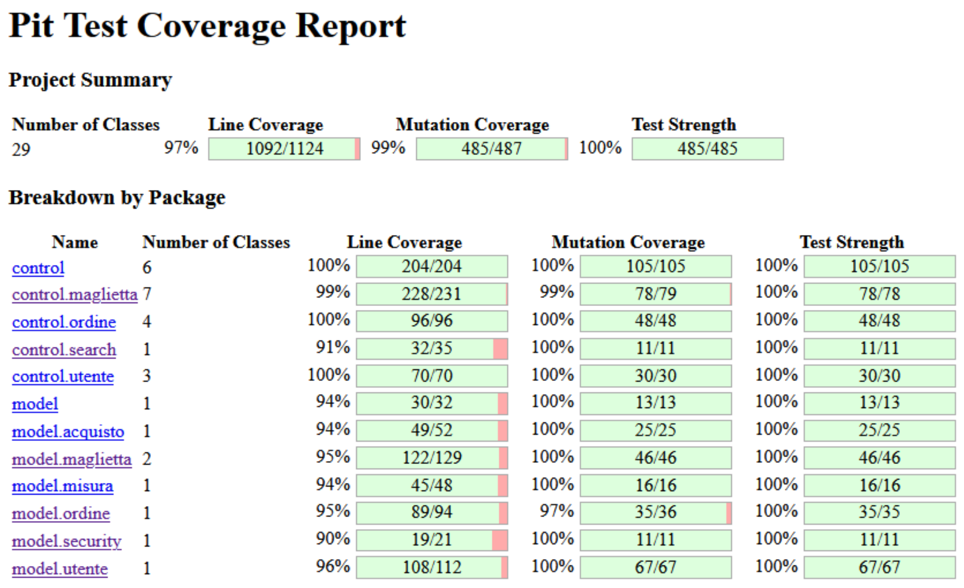
Fig. 4.4. Codecov coverage overview on main branch.

Coverage evolution over time. The analysis of the temporal trend highlights a clear increase in coverage following the introduction of automated testing. In particular, the graph shows an initial phase characterized by the absence of coverage, followed by a rapid growth.

Coverage distribution. The distribution of covered and uncovered lines confirms that the vast majority of the application code is exercised by the test suite. The few uncovered lines are concentrated in marginal portions of the codebase or in edge cases, consistently with the exclusions already justified in the methodological phase.

4.4 Pitest Results

The results produced by PiTest show a **very high level of test effectiveness** across the analyzed components. At project level, the mutation analysis reports a **mutation coverage of 99%** and a **test strength of 100%**, indicating that almost all generated mutants were exercised and successfully detected by the test suite.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
29	97%	1092/1124	99%	485/487	100%	485/485

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
control	6	100%	204/204	100%	105/105	100%	105/105
control.maglietta	7	99%	228/231	99%	78/79	100%	78/78
control.ordine	4	100%	96/96	100%	48/48	100%	48/48
control.search	1	91%	32/35	100%	11/11	100%	11/11
control.utente	3	100%	70/70	100%	30/30	100%	30/30
model	1	94%	30/32	100%	13/13	100%	13/13
model.acquisto	1	94%	49/52	100%	25/25	100%	25/25
model.maglietta	2	95%	122/129	100%	46/46	100%	46/46
model.misura	1	94%	45/48	100%	16/16	100%	16/16
model.ordine	1	95%	89/94	97%	35/36	100%	35/35
model.security	1	90%	19/21	100%	11/11	100%	11/11
model.utente	1	96%	108/112	100%	67/67	100%	67/67

Fig. 4.5. PiTest mutation testing report with overall results and package-level breakdown.

For each implemented test suite, PiTest was used to identify surviving mutants, which were then manually analyzed to understand their causes.

Surviving mutants typically highlighted missing assertions, insufficiently precise checks, or corner cases not explicitly covered by the tests. Based on this feedback, the corresponding test cases were refined and extended to better characterize the expected behavior of the affected components. This process was repeated iteratively until the remaining mutants were either eliminated or assessed as non-relevant with respect to the intended behavior.

As a result, most packages in both the control and model layers achieve **full or near-full mutation coverage**. The few remaining surviving mutants are confined to isolated cases and do not indicate systematic gaps in the testing strategy.

Overall, the PiTest results reflect the outcome of an **incremental test development process** in which mutation analysis was actively used to guide and strengthen the test suites.

4.5 JMH Results

As a result of the microbenchmarking activity described in the methodological section, the project is characterized through a set of JMH executions targeting selected backend components. The produced measurements provide an empirical view of the execution cost of representative operations under controlled workload parameters, supporting a more objective interpretation of performance-sensitive code paths.

Given the overall **limited computational complexity** of the application, the benchmarking activity was intentionally restricted to three classes, selected as the only ones exposing non-trivial, non I/O-bound logic suitable for microbenchmarking. In particular, the remaining components are largely dominated by database access or by user interactions, whose performance is primarily determined by external factors and would not yield stable or meaningful microbenchmark results.

CarrelloModel microbenchmark. Figure 4.6 summarizes the results for the shopping-cart logic by varying the cart size (`cartSize`). The measured operations exhibit a clear growth with the number of elements, consistent with the use of linear scans over an `ArrayList`.

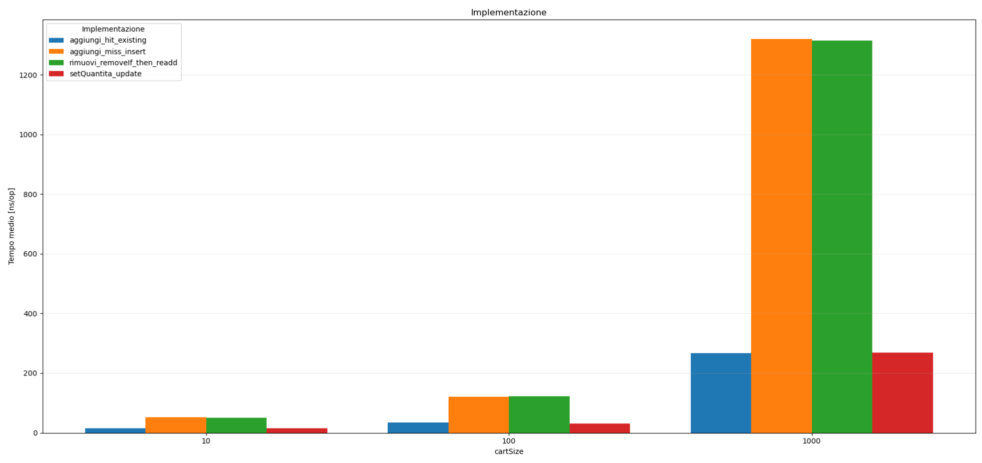


Fig. 4.6. JMH results for `CarrelloModelBenchmark` at different `cartSize`.

The following observations can be derived:

- **`aggiungi_miss_insert`** and **`rimuovi_removeIf_then_readd`** are the most expensive operations and scale sharply with `cartSize`, coherently with full-list traversal in the miss case and predicate-based removal over the collection.
- **`aggiungi_hit_existing`** and **`setQuantita_update`** remain comparatively lower, but still increase with `cartSize`, indicating that the dominant cost is the search phase rather than the update itself.

CryptoUtils microbenchmark. Figure 4.7 reports the performance of encryption, decryption, and a full encrypt-then-decrypt roundtrip for increasing plaintext sizes. The results show a stable baseline for small payloads and a visible growth for larger inputs, reflecting the increasing amount of data processed and encoded/decoded.

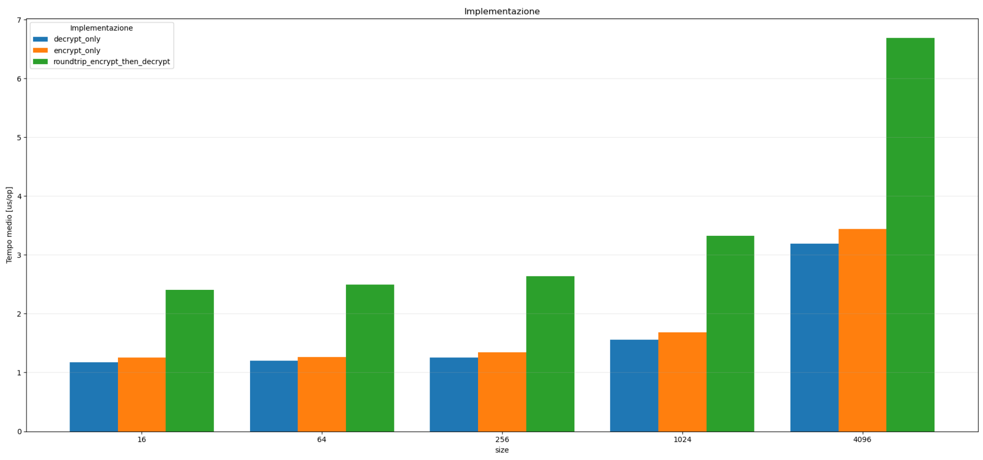


Fig. 4.7. JMH results for `CryptoUtilsBenchmark` at different input sizes.

Two trends are particularly evident:

- **Encrypt-only** and **decrypt-only** remain close for small inputs and increase for larger sizes, indicating a transition from fixed per-call overhead to payload-dominated cost.
- The **roundtrip** operation is consistently higher than individual encrypt/decrypt, since it combines both phases and includes the overhead of handling the produced encoding format.

StampaFattura microbenchmark. Figure 4.8 shows the cost of generating a PDF invoice in memory while varying the number of invoice items. The results are expressed in milliseconds per operation and show a moderate increase with the number of rendered lines.

The observed behavior suggests that:

- a **fixed setup cost** (document creation and serialization) dominates the overall execution time for small invoices;
- increasing the number of items produces an incremental overhead, consistent with additional rendering operations performed for each row.

Summary. Overall, the microbenchmark results provide a baseline performance profile of representative backend components:

1. collection-based business logic is sensitive to input size due to linear traversals;
2. cryptographic routines exhibit the expected growth with payload size;
3. PDF generation shows a dominant fixed cost with moderate per-item overhead.

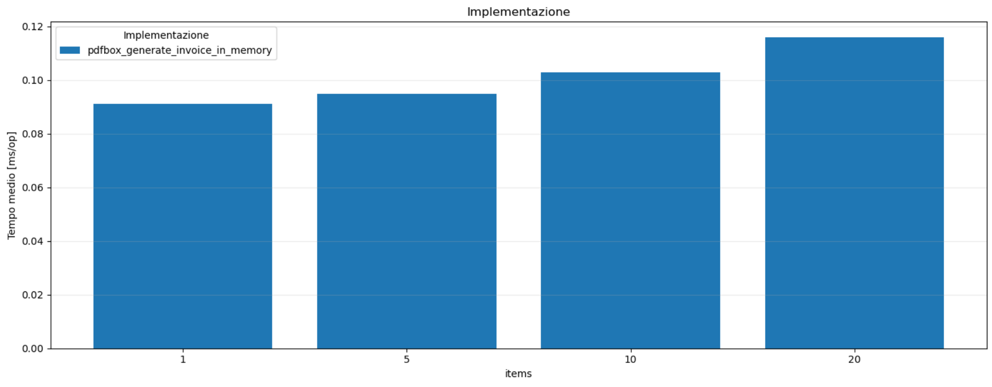


Fig. 4.8. JMH results for `StampaFatturaBenchmark` by varying the number of invoice items.

4.6 SonarQube Results

The results produced by SonarQube provide a clear and consolidated view of the effects introduced by the interventions carried out during the project. A comparison between the initial assessment and the final analysis highlights a substantial improvement across all monitored code quality dimensions, as well as full compliance with the defined quality gate.

Initial assessment. Prior to the introduction of automated testing and formal specifications, the initial SonarQube analysis reported a large number of open issues affecting **security**, **reliability**, and **maintainability**. At this stage, code coverage was effectively absent due to the lack of an executable test suite.

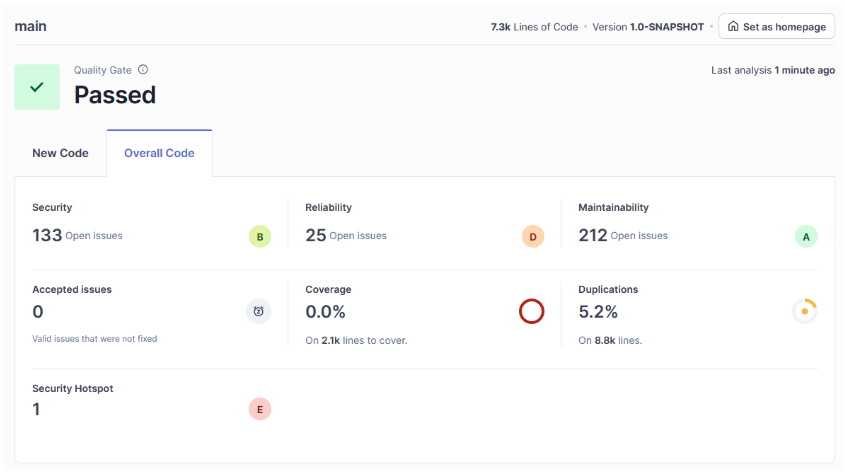
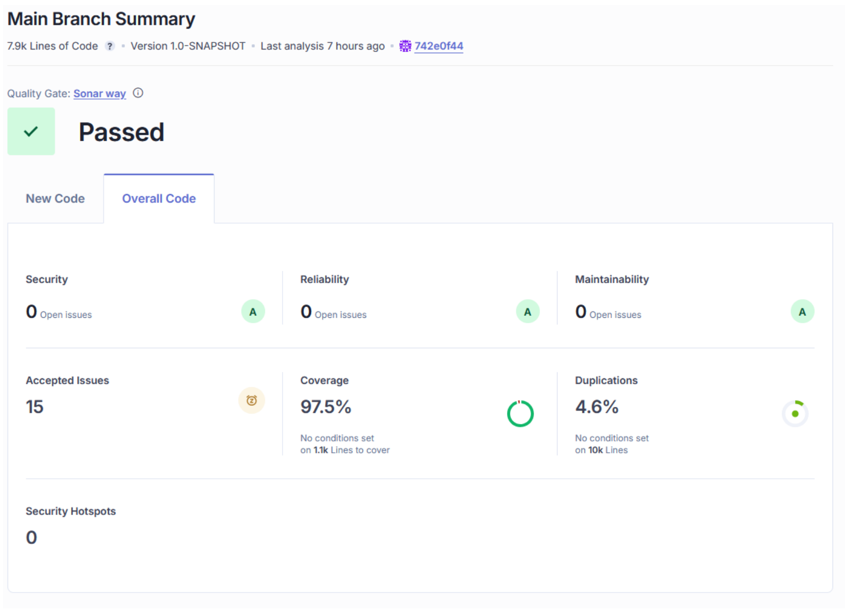


Fig. 4.9. SonarQube overview before interventions.

Post-intervention assessment. Following the introduction of unit testing and coverage enforcement, the SonarQube analysis presents a markedly different outcome. All quality gate conditions are satisfied, and no open issues are reported for security, reliability, or maintainability in the analyzed codebase.


Main Branch Summary

7.9k Lines of Code • Version 1.0-SNAPSHOT • Last analysis 7 hours ago • 742e0f44

Quality Gate: Sonar way

Passed

New Code | Overall Code

Security: 0 Open issues (A)

Reliability: 0 Open issues (A)

Maintainability: 0 Open issues (A)

Accepted Issues: 15

Coverage: 97.5% — No conditions set on 1.1k Lines to cover

Duplications: 4.6% — No conditions set on 10k Lines

Security Hotspots: 0

Fig. 4.10. SonarQube overview after interventions.

Evolution of issues and coverage. The improvement is further supported by the analysis of historical trends provided by SonarQube. The issue timeline shows a progressive reduction in the number of reported findings, culminating in their complete elimination after the stabilization of the testing and analysis pipeline.

In parallel, the coverage trend highlights a significant and sustained increase in the number of covered lines, reaching a high and stable level. These results are consistent with the measurements produced by JaCoCo and Codecov, confirming the effectiveness of the test suite and the coherence of coverage reporting across tools.

Quality gate enforcement. The SonarQube quality gate adopted in the project is primarily based on two conditions:

(1) the absence of new quality issues affecting security, reliability, or maintainability, and
(2) compliance with the defined code coverage requirements.

Passing the quality gate therefore reflects not only adequate test coverage, but also the overall consistency and quality of the analyzed code.

Issue classification and acceptance. During the analysis process, a subset of findings related to **test cases** and **JML annotations** were flagged by SonarQube. These findings

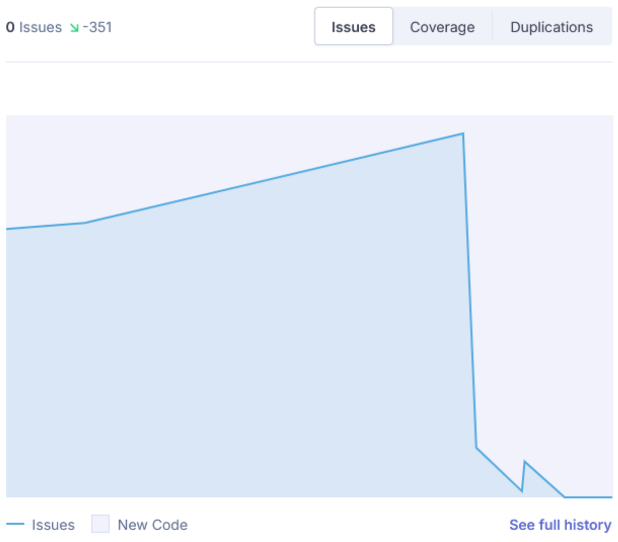


Fig. 4.11. SonarQube issues trend over time.

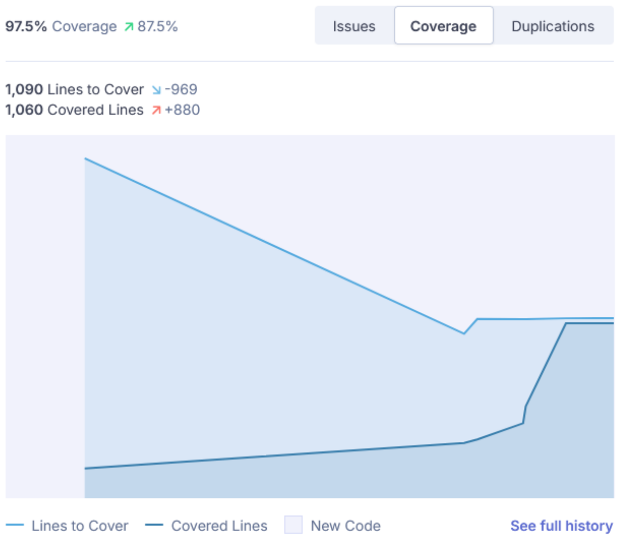


Fig. 4.12. SonarQube coverage trend over time.

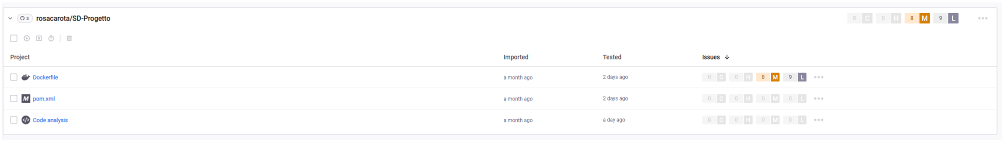
were reviewed and classified as **false positives**, as they originate from correct testing patterns and formal specifications that are not fully captured by generic static analysis rules.

In addition, a limited number of issues were explicitly **accepted**. These warnings concern specific field assignments, such as references to data sources and cryptographic keys, which were introduced to support testability. In particular, additional constructors were required to enable isolated unit testing; removing these assignments would have compromised the ability to execute the tests reliably.

4.7 Snyk Results

The application of Snyk security analysis provides insight into the security posture of the project with respect to **external dependencies** and **container configuration**. Snyk continuously scans the codebase, declared dependencies, and container artifacts, and supports remediation through the automatic generation of corrective pull requests.

Initial dependency and container analysis. The initial Snyk analysis identified several known vulnerabilities affecting both **third-party Java libraries** and **container-level dependencies**. Some of these issues overlap with vulnerabilities already detected through container registry scanning, while others are specific to application-level dependencies declared in the build configuration.

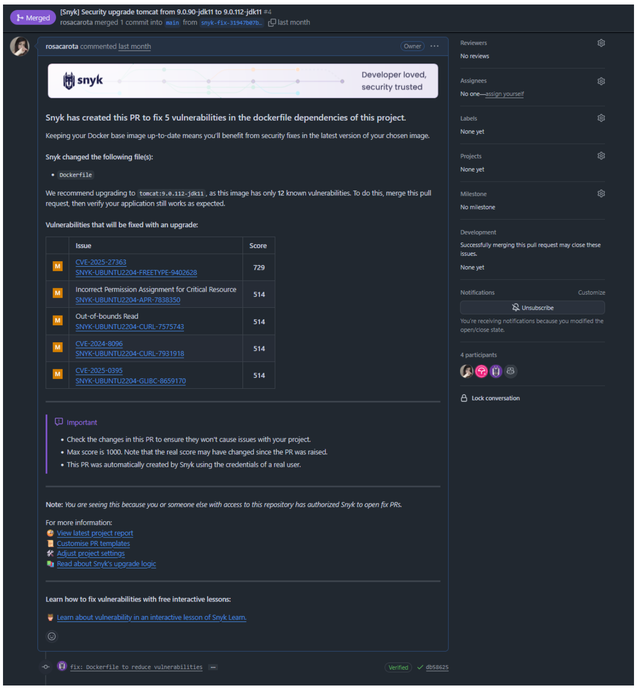


rosacarota/SD-Progetto — 0 C 0 H 8 M 9 L

Project	Imported	Tested	Issues
Dockerfile	a month ago	2 days ago	0 C 0 H 8 M 9 L
pom.xml	a month ago	2 days ago	0 C 0 H 0 M 0 L
Code analysis	a month ago	a day ago	0 C 0 H 0 M 0 L

Fig. 4.13. Snyk project vulnerability overview.

Automated remediation via pull requests. For vulnerabilities affecting **application dependencies** and **Docker configuration**, Snyk automatically generated a set of pull requests proposing secure version upgrades. These pull requests target both the `Dockerfile` (e.g., base image updates) and Maven dependencies.


Merged [Snyk] Security upgrade tomcat from 9.0.90-jdk11 to 9.0.112-jdk11 #4
rosacarota merged 1 commit into `main` from `snyk-fix-31947b07b…` last month

rosacarota commented last month Owner

Developer loved, security trusted

Snyk has created this PR to fix 5 vulnerabilities in the dockerfile dependencies of this project.

Keeping your Docker base image up-to-date means you'll benefit from security fixes in the latest version of your chosen image.

Snyk changed the following file(s):

- `Dockerfile`

We recommend upgrading to `tomcat:9.0.112-jdk11`, as this image has only **12** known vulnerabilities. To do this, merge this pull request, then verify your application still works as expected.

Vulnerabilities that will be fixed with an upgrade:

	Issue	Score
M	CVE-2025-27363 SNYK-UBUNTU2204-FREETYPE-9402628	729
M	Incorrect Permission Assignment for Critical Resource SNYK-UBUNTU2204-APR-7838350	514
M	Out-of-bounds Read SNYK-UBUNTU2204-CURL-7575743	514
M	CVE-2024-8096 SNYK-UBUNTU2204-CURL-7931918	514
M	CVE-2025-0395 SNYK-UBUNTU2204-GLIBC-8659170	514

Important

- Check the changes in this PR to ensure they won't cause issues with your project.
- Max score is 1000. Note that the real score may have changed since the PR was raised.
- This PR was automatically created by Snyk using the credentials of a real user.

Note: *You are seeing this because you or someone else with access to this repository has authorized Snyk to open fix PRs.*

For more information:
View latest project report
Customise PR templates
Adjust project settings
Read about Snyk's upgrade logic

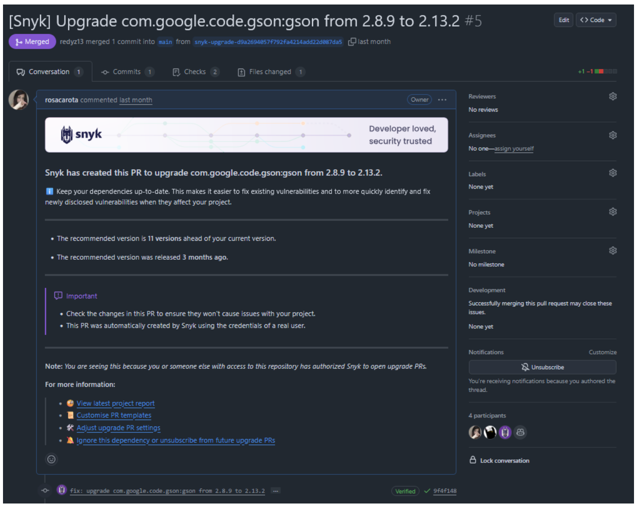
Learn how to fix vulnerabilities with free interactive lessons:

Learn about vulnerability in an interactive lesson of Snyk Learn.

`fix: Dockerfile to reduce vulnerabilities` Verified ✓ db58625

Reviewers: No reviews
Assignees: No one—assign yourself
Labels: None yet
Projects: None yet
Milestone: No milestone
Development: Successfully merging this pull request may close these issues. None yet
Notifications: Customize
Unsubscribe
You're receiving notifications because you modified the open/close state.
4 participants
Lock conversation

Fig. 4.14. Snyk upgrade of Dockerfile.


[Snyk] Upgrade com.google.code.gson:gson from 2.8.9 to 2.13.2 #5
Merged redyz13 merged 1 commit into `main` from `snyk-upgrade-d9a2694057f792fa4214add22d087da5` last month

rosacarota commented last month Owner

Developer loved, security trusted

Snyk has created this PR to upgrade com.google.code.gson:gson from 2.8.9 to 2.13.2.

Keep your dependencies up-to-date. This makes it easier to fix existing vulnerabilities and to more quickly identify and fix newly disclosed vulnerabilities when they affect your project.

- The recommended version is **11 versions** ahead of your current version.
- The recommended version was released **3 months ago.**

Important

- Check the changes in this PR to ensure they won't cause issues with your project.
- This PR was automatically created by Snyk using the credentials of a real user.

Note: *You are seeing this because you or someone else with access to this repository has authorized Snyk to open upgrade PRs.*

For more information:

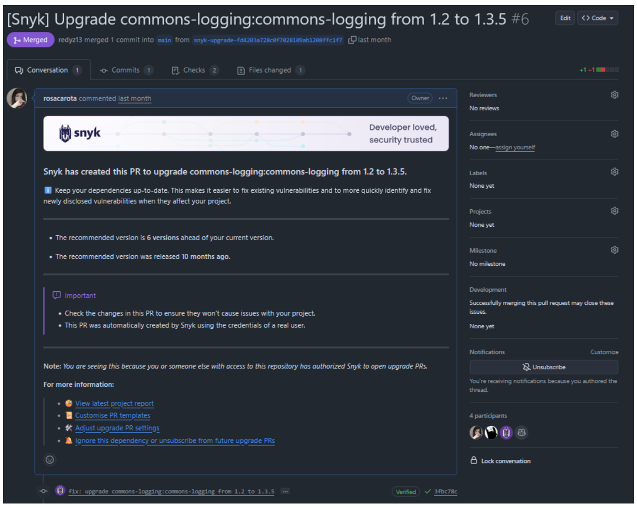
- View latest project report
- Customise PR templates
- Adjust upgrade PR settings
- Ignore this dependency or unsubscribe from future upgrade PRs

`fix: upgrade com.google.code.gson:gson from 2.8.9 to 2.13.2` Verified ✓ 9f4f148

Reviewers: No reviews
Assignees: No one—assign yourself
Labels: None yet
Projects: None yet
Milestone: No milestone
Development: Successfully merging this pull request may close these issues. None yet
Notifications: Customize
Unsubscribe
You're receiving notifications because you authored the thread.
4 participants
Lock conversation

Fig. 4.15. Snyk upgrade of `com.google.code.gson:gson`.

The proposed changes include upgrades of widely used libraries (e.g., JSON processing, logging, PDF handling, and database connectors), allowing the application to benefit from security fixes introduced in more recent releases. Each pull request explicitly documents the addressed vulnerabilities and the expected impact on the project's security posture.



[Snyk] Upgrade commons-logging:commons-logging from 1.2 to 1.3.5 #6

Merged redyz13 merged 1 commit into `main` from `snyk-upgrade-fd4203a728c0f7028109ab1208ffc1f7` last month

rosacarota commented last month — Owner

Snyk has created this PR to upgrade commons-logging:commons-logging from 1.2 to 1.3.5.

ℹ️ Keep your dependencies up-to-date. This makes it easier to fix existing vulnerabilities and to more quickly identify and fix newly disclosed vulnerabilities when they affect your project.

- The recommended version is **6 versions** ahead of your current version.
- The recommended version was released **10 months ago.**

Important

- Check the changes in this PR to ensure they won't cause issues with your project.
- This PR was automatically created by Snyk using the credentials of a real user.

Note: *You are seeing this because you or someone else with access to this repository has authorized Snyk to open upgrade PRs.*

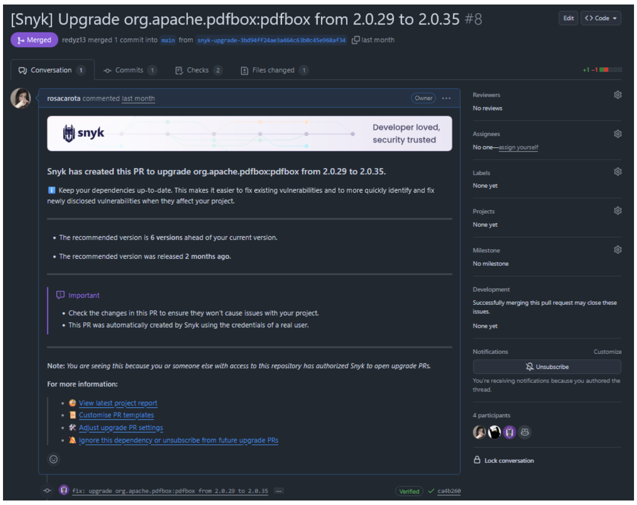
For more information:

- View latest project report
- Customise PR templates
- Adjust upgrade PR settings
- Ignore this dependency or unsubscribe from future upgrade PRs

`fix: upgrade commons-logging:commons-logging from 1.2 to 1.3.5` Verified ✓ 3fbc78c

Reviewers: No reviews
Assignees: No one—assign yourself
Labels: None yet
Projects: None yet
Milestone: No milestone
Development: Successfully merging this pull request may close these issues. None yet
Notifications — Customize
Unsubscribe
You're receiving notifications because you authored the thread.
4 participants
Lock conversation

Fig. 4.16. Snyk upgrade of `commons-logging:commons-logging`.



[Snyk] Upgrade org.apache.pdfbox:pdfbox from 2.0.29 to 2.0.35 #8

Merged redyz13 merged 1 commit into `main` from `snyk-upgrade-3bd94ff24ae3a464c63b0c45e968af34` last month

rosacarota commented last month — Owner

Snyk has created this PR to upgrade org.apache.pdfbox:pdfbox from 2.0.29 to 2.0.35.

ℹ️ Keep your dependencies up-to-date. This makes it easier to fix existing vulnerabilities and to more quickly identify and fix newly disclosed vulnerabilities when they affect your project.

- The recommended version is **6 versions** ahead of your current version.
- The recommended version was released **2 months ago.**

Important

- Check the changes in this PR to ensure they won't cause issues with your project.
- This PR was automatically created by Snyk using the credentials of a real user.

Note: *You are seeing this because you or someone else with access to this repository has authorized Snyk to open upgrade PRs.*

For more information:

- View latest project report
- Customise PR templates
- Adjust upgrade PR settings
- Ignore this dependency or unsubscribe from future upgrade PRs

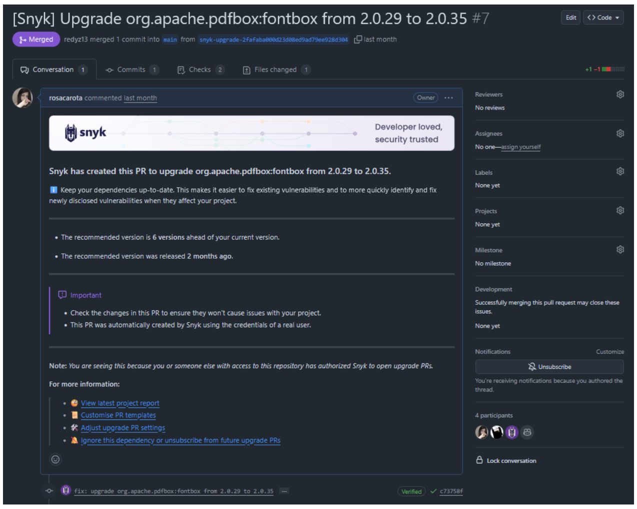
`fix: upgrade org.apache.pdfbox:pdfbox from 2.0.29 to 2.0.35` Verified ✓ ca4b260

Reviewers: No reviews
Assignees: No one—assign yourself
Labels: None yet
Projects: None yet
Milestone: No milestone
Development: Successfully merging this pull request may close these issues. None yet
Notifications — Customize
Unsubscribe
You're receiving notifications because you authored the thread.
4 participants
Lock conversation

Fig. 4.17. Snyk upgrade of `org.apache.pdfbox:pdfbox`.

Resolution and residual vulnerabilities. All Snyk-generated pull requests related to **application-level dependencies** were reviewed and merged, leading to the resolution of the corresponding vulnerabilities. After these updates, no high-severity vulnerabilities remain associated with the project's declared Java dependencies.

Residual vulnerabilities reported by Snyk are limited to **operating system packages** inherited from the Docker base image. These issues are consistent with those observed during container registry analysis and affect standard Linux utilities rather than application-specific components.


[Snyk] Upgrade org.apache.pdfbox:fontbox from 2.0.29 to 2.0.35 #7

Merged redyz13 merged 1 commit into `main` from `snyk-upgrade-2fafaba000d23d08ed9ad79ee928d304` last month

Conversation 1 · Commits 1 · Checks 2 · Files changed 1 · +1 −1

rosacarota commented last month · Owner

snyk — Developer loved, security trusted

Snyk has created this PR to upgrade org.apache.pdfbox:fontbox from 2.0.29 to 2.0.35.

ℹ️ Keep your dependencies up-to-date. This makes it easier to fix existing vulnerabilities and to more quickly identify and fix newly disclosed vulnerabilities when they affect your project.

- The recommended version is **6 versions** ahead of your current version.
- The recommended version was released **2 months ago**.

Important
- Check the changes in this PR to ensure they won't cause issues with your project.
- This PR was automatically created by Snyk using the credentials of a real user.

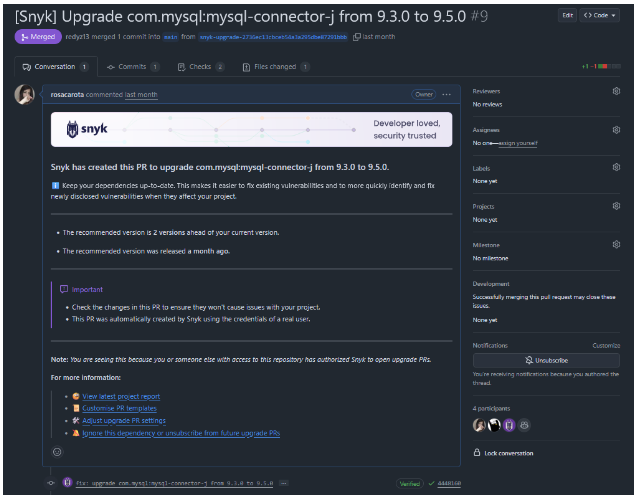
Note: *You are seeing this because you or someone else with access to this repository has authorized Snyk to open upgrade PRs.*

For more information:
- View latest project report
- Customise PR templates
- Adjust upgrade PR settings
- Ignore this dependency or unsubscribe from future upgrade PRs

`fix: upgrade org.apache.pdfbox:fontbox from 2.0.29 to 2.0.35` Verified ✓ c73758f

Reviewers: No reviews
Assignees: No one—assign yourself
Labels: None yet
Projects: None yet
Milestone: No milestone
Development: Successfully merging this pull request may close these issues. None yet
Notifications: Customize · Unsubscribe · You're receiving notifications because you authored the thread.
4 participants
Lock conversation

Fig. 4.18. Snyk upgrade of `org.apache.pdfbox:fontbox`.


[Snyk] Upgrade com.mysql:mysql-connector-j from 9.3.0 to 9.5.0 #9

Merged redyz13 merged 1 commit into `main` from `snyk-upgrade-2736ec13cbceb54a3a295dbe87291bbb` last month

Conversation 1 · Commits 1 · Checks 2 · Files changed 1 · +1 −1

rosacarota commented last month · Owner

snyk — Developer loved, security trusted

Snyk has created this PR to upgrade com.mysql:mysql-connector-j from 9.3.0 to 9.5.0.

ℹ️ Keep your dependencies up-to-date. This makes it easier to fix existing vulnerabilities and to more quickly identify and fix newly disclosed vulnerabilities when they affect your project.

- The recommended version is **2 versions** ahead of your current version.
- The recommended version was released **a month ago**.

Important
- Check the changes in this PR to ensure they won't cause issues with your project.
- This PR was automatically created by Snyk using the credentials of a real user.

Note: *You are seeing this because you or someone else with access to this repository has authorized Snyk to open upgrade PRs.*

For more information:
- View latest project report
- Customise PR templates
- Adjust upgrade PR settings
- Ignore this dependency or unsubscribe from future upgrade PRs

`fix: upgrade com.mysql:mysql-connector-j from 9.3.0 to 9.5.0` Verified ✓ 4448160

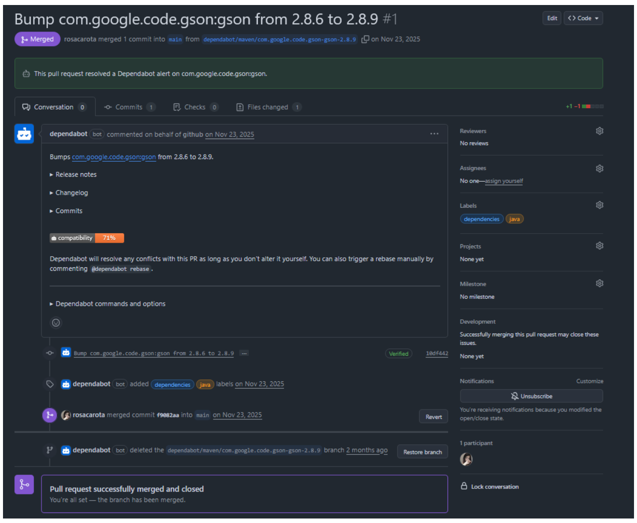
Reviewers: No reviews
Assignees: No one—assign yourself
Labels: None yet
Projects: None yet
Milestone: No milestone
Development: Successfully merging this pull request may close these issues. None yet
Notifications: Customize · Unsubscribe · You're receiving notifications because you authored the thread.
4 participants
Lock conversation

Fig. 4.19. Snyk upgrade of `com.mysql:mysql-connector-j`.

4.8 Dependabot Results

During the analysis process, Dependabot identified a security vulnerability affecting the `com.google.code.gson:gson` dependency used within the project. The library was found to be outdated with respect to the current state of the Maven ecosystem and associated with a known vulnerability.

Update and resolution. Dependabot automatically generated a pull request proposing the upgrade of the gson library from version `2.8.6` to version `2.8.9`. The pull request



Bump com.google.code.gson:gson from 2.8.6 to 2.8.9 #1

Merged rosacarota merged 1 commit into `main` from `dependabot/maven/com.google.code.gson-gson-2.8.9` on Nov 23, 2025

This pull request resolved a Dependabot alert on com.google.code.gson:gson.

dependabot bot commented on behalf of github on Nov 23, 2025

Bumps com.google.code.gson:gson from 2.8.6 to 2.8.9.

▸ Release notes

▸ Changelog

▸ Commits

compatibility 71%

Dependabot will resolve any conflicts with this PR as long as you don't alter it yourself. You can also trigger a rebase manually by commenting `@dependabot rebase`.

▸ Dependabot commands and options

`Bump com.google.code.gson:gson from 2.8.6 to 2.8.9` Verified `10df442`

dependabot bot added dependencies java labels on Nov 23, 2025

rosacarota merged commit `f9082aa` into `main` on Nov 23, 2025

dependabot bot deleted the `dependabot/maven/com.google.code.gson-gson-2.8.9` branch 2 months ago

Pull request successfully merged and closed

You're all set — the branch has been merged.

Reviewers: No reviews

Assignees: No one—assign yourself

Labels: dependencies java

Projects: None yet

Milestone: No milestone

Development: Successfully merging this pull request may close these issues. None yet

Notifications: You're receiving notifications because you modified the open/close state.

1 participant

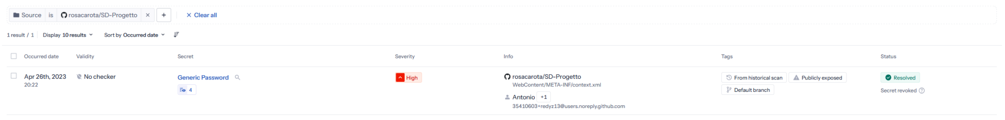
Fig. 4.20. Dependabot pull request for gson dependency upgrade.

included compatibility information, references to official changelogs, and an assessment of the confidence level associated with the update.

The proposed upgrade was reviewed and merged into the main branch without requiring additional changes to the application code.

4.9 GitGuardian Results

The analysis performed by GitGuardian identified the presence of **sensitive information** within the repository, highlighting a violation of recommended secret management practices. The detection concerned configuration files containing values that should not have been tracked in version control.



Source is rosacarota/SD-Progetto

1 result / 1 Display 10 results Sort by Occurred date

Occurred date	Validity	Secret	Severity	Info	Tags	Status
Apr 26th, 2023 20:22	No checker	Generic Password 4	High	rosacarota/SD-Progetto WebContent/META-INF/context.xml Antonio +1 35410603+redyz13@users.noreply.github.com	From historical scan; Publicly exposed; Default branch	Resolved Secret revoked

Fig. 4.21. GitGuardian alert showing detected secret exposure.

Resolution and mitigation. Following the alert, the issue was addressed by removing the exposed sensitive information from the repository and by adopting a correct configuration management strategy based on **environment variables** and non-versioned configuration files. After the corrective intervention, GitGuardian no longer reported exposed secrets.

5 Conclusions

This project applied a comprehensive set of *software dependability techniques* to an existing Java web application, with the goal of evaluating and improving its reliability, security, testability, and overall robustness, without modifying its functional requirements.

Starting from a codebase that initially lacked automated testing, formal specifications, and systematic security controls, the project introduced a structured methodology covering *fault avoidance*, *fault detection*, and *fault assessment*. Formal specifications based on JML were selectively applied to core domain components, making system constraints explicit and supporting early detection of inconsistent states.

Testing activities played a central role in the analysis. The introduction of a systematic unit test suite enabled extensive verification of application logic, while structural coverage analysis with JaCoCo and Codecov confirmed the achievement of high and stable coverage levels. Mutation testing with PiTest further demonstrated the effectiveness of the test suite in detecting behavioral faults beyond structural metrics.

Security was analyzed across multiple layers of the system. Static analysis with SonarQube, dependency and container scanning with Snyk and Dependabot, and secret detection with GitGuardian contributed to identifying and mitigating vulnerabilities, while clearly distinguishing between application-level issues and those inherited from the execution environment.

Containerization and CI/CD automation ensured reproducibility and consistency of the analysis process. Although no production deployment was performed, quality gates enforced during the pipeline provided continuous verification of dependability requirements.

Overall, the results illustrate how dependability analysis can be effectively conducted on an existing codebase by combining complementary techniques, even when the system was not originally designed with verification in mind.