University of Salerno

Department of Computer Science

Software Dependability Course

Report

Team

Choaib Goumri NF22500223
Giuseppe Sabetta NF22500155
Lucageneroso Cammarota NF22500053

Repository GitHub

Academic Year 2025–2026

Contents

1 Introduction

1.1 Context

UniClass is a web application designated to facilitate the management of academic activities of students and professors. The idea was born for the needs of having a unique and centralized platform to provide easier organization of courses, lectures and schedules, monitoring marks, managing academic resources and real-time visualization of free spaces to study in, in an efficient way. UniClass was created by a team of 4 developers and 3 students at the University of Salerno, as a practice test for Software Engineering exam, applying several concepts for academic purposes:

- **Distributed Programming** – Enterprise Technologies, (i.e. Jakarta EE, Injections, . . .)
- **Software Engineering** – for project management, development and design patterns;
- **Database** – for managing CRUD operations regarding persistent informations.

1.2 Problems

Thanks to a in-depth study of software architectures and used technologies for creational and maintenance of the academic project, it was inferred that the designated platform was not re-usable, not enough documentated and without any automated test. In fact, especially in digital resources available in academia or enterprise, it is important to have very high explainability within the documentation relating to the software (and/or hardware) structure at various levels of abstraction. The effectiveness of the methods used in the classes describing the teaching logic cannot be quantified due to the lack of extensive unit testing. Furthermore, the lack of workflow, CI/CD, efficient methodologies for workload distribution, and ineffective adoption of KanBan methods have made it difficult for the project to achieve optimal visibility across the entire platform for each deployment in the public repository. This does not reflect the definition of Software Dependability, as covered during the course, in almost any way.

1.3 Goals

Given the issues encountered in analyzing the development of the platform and given the definition and good practices of Software Dependability, the various tasks described must be carried out:

- **CI/CD** – Immediate identification of outputs defined in the CI phase, possibility to build, test, or deliver the project and analyze its results or logs;
- **JML** – Creation of formal specifications using JML (where possible due to compatibility with Jakarta and Injections), correcting the code to follow best practices as determined by the static and dynamic analyzer;
- **Docker Image** – Isolation, reusability, and containerization;
- **Test Increase** – Verification and validation of the code in the project according to the ALARP Principle;
- **Code Coverage** – Verification of code coverage by tests created using JaCoCo, CodeCov, and Pit;
- **MicroBenchmark** – Analysis of methods that require more execution time due to design or code smells;
- **Security** – Introduction of security mechanisms integrated into CI/CD and verification using GitGuardian, Snyk, and SonarQube (Cloud).

2 Docker

2.1 Definition

Docker is a platform that allows a project to be containerized, making it reusable and ready to use in any environment. A container is a standard unit of software that packages code and all its dependencies so that the application can be run quickly and reliably from one computing environment to another. Docker offers extreme portability, eliminating issues related to configuration files and using lightweight host kernels, thereby accelerating deployment thanks to shared layers.

2.2 Initial Context

In the context of this project, before the system was dockerized, it was necessary to manually install dependencies, manually configure the installed Tomcat server, modify the artifacts used for the target, configure the PostgreSQL server, and run the web application. This resulted in a lengthy and error-prone process, making it difficult to reproduce the environment, leading to compatibility issues, differences in response times, and difficulties in managing dependency versions.

2.3 Implementation

2.3.1 Containerization Process

To solve these problems, it was decided to use Docker to create a containerized environment for the web application in question. Two Docker files were created, specifically:

- **Dockerfile** for the web application, defining the base image (eclipse-temurin:21-jdk), the automatic download and import of the Tomcat EE server (required for the injections and Jakarta technologies used), copying the configuration files necessary to grant the container permissions to communicate via objects, inject resources, and configure the data source;

- **docker-compose.yml** to orchestrate both the web application container (present in the Dockerfile) and the PostgreSQL database container. Docker Compose does not simply have a list of services: in the context of this project under development, orchestration means a waiting time to avoid conflicts between the two containers. In particular, the database must be started before the web application to avoid database connection errors when the application is launched. In addition, uniclassnet was created as a network to isolate service traffic and improve security.

2.3.2 Problems Encountered

During the implementation phase, given the dispersion of configuration files and the lack of related documentation, it was difficult to reconstruct the steps necessary for the correct configuration of the Tomcat EE server, particularly with regard to the configuration of the datasource and JNDI resources. In particular, the configuration files were spread across the files in the folder automatically created by the SmartTomCat plugin, which was very important for the first version of the project. In addition, it was necessary to make numerous fixes to the database connection URLs, making them compatible with the Docker standard, and to download the Tomcat server. Another issue encountered during the containerization phase was a race condition between the web application container, enabled by the TomEE Plume configurations (enabled for enterprise environments and object injection, automating computational logic processes), and the data request from the PostgreSQL database. This required the ability to populate entities that, due to the aforementioned issue, could not be viewed. The documentation for this type of race condition is lacking, and the exception messages were also not very comprehensive, making the work much more burdensome and complicated. The stack trace was thoroughly analyzed and the race condition was determined based on an instant database analysis and the missing entities, which, however, were requested by one service or the other. To remedy the problem, a service health checker was simply added, first making the database available, ready for connection to the web app, and then retrieving all the information automatically populated by the Database Populator. Several commits are available that present fixes to the Dockerfile because, despite the isolation of the containers, it was not initially possible to achieve reproducibility between different operating systems due to incorrect code configurations that forced invalid connections.

2.4 Production Environment

Following the changes to be made in line with the objectives defined in the Introduction, such as ensuring code quality and web application security, adding unit tests and microbenchmarking, it was decided to upload the Docker image of the final web application to the public registry, so that it could be reused for any subsequent maintenance or future development. The commands used to push the image are as follows:

```
mvn clean package -DskipTests
docker build -t gssab3/uniclass-dependability:latest .
docker login
docker push gssab3/uniclass-dependability:latest
```

This way, anyone can download the Docker image of the web application, making it easily deployable and scalable. Obviously, due to the principles of containerization and, therefore, isolation, the image does not include the encapsulated PostgreSQL service. For quick and effective reusability, simply use the docker-compose.prod.yml available in the repository:

```
docker compose -f docker-compose.prod.yml [comando desiderato]
```

In the desired command, simply enter up/down/build depending on the desired operation. Inside, you will find the web application's war package downloaded from the public registry and the PostgreSQL service configured for production use.

2.5 Conclusions

The adoption of Docker for containerization has brought countless advantages, including those described in the previous sections of this chapter, at the expense of a macro-refactoring on the config-files side, corrective maintenance of the adoption of Jakarta libraries for injections and database connections. Furthermore, the use of Docker has facilitated the deployment and scalability of the web application, simplifying the development process and future maintenance. Finally, the Docker image published on the public registry allows anyone to use the web application without having to deal with the complexities of configuring the environment, thus improving the dissemination and adoption of the project.

3 Software Fault Avoidance and Model Checking

3.1 Introduction

Software Fault Avoidance is based on formal methods to prevent errors in code through modeling and static verification of specifications, ensuring that software is correct by construction before execution. OpenJML is an open-source tool that allows you to formally verify Java code using specifications written in JML (Java Modeling Language). JML is a formal specification language that allows you to annotate Java code with preconditions, postconditions, and invariants, facilitating the verification of program correctness. Thanks to OpenJML, it is possible to perform static checks on Java code to ensure that it complies with the specifications defined in JML, through the use of formal analysis techniques, deductive verification, and correctness assurance, but also runtime assertion checking, which allows dynamic checks to be inserted into the code to verify the specifications during program execution, useful when static verification is not sufficient.

3.2 Context

In the context of this project, the use of OpenJML was considered to verify the formal correctness of the web application components, through manual translation of the contracts present in the Object Design Document, among the deliverables of the UniClass system, into JML specifications. The goal was not only to ensure that the formality required by the document was implemented, thus adhering to the Secure By Design principle, but also to perform a corrective function, capable of eliminating preconditions, postconditions, or unmanaged data flows that could have led to vulnerabilities or malfunctions in the application.

3.3 Implementation

3.3.1 Installation Process

To integrate OpenJML into the development process, we started by installing the tool, following the official guidelines on the OpenJML website and in the handouts on the Software Dependability course platform. Next, we started with POJOs, more precisely with the classes representing the system entities and the data model used, trying to increase abstraction until we reached DAOs, services, configurations, critical utilities, etc.

For each selected class, we wrote the JML specifications, translating the contracts in the document, adding any additional specifications (or reducing them), thus correcting the source code, albeit slightly.

3.3.2 Refactoring and Details

In particular, in order to comply with the revision returned by the execution of OpenJML, the following changes were made to the source code:

1. **Modifying definition of constants** – For each initial definition of NamedQuery, the class to which it belongs, followed by a full stop, has been added to "name" as a prefix. This is due to the limitations of OpenJML, which is difficult to use in an enterprise environment (i.e. Jakarta EE), which requires the use of containers, injections and other automations that are not managed statically: more precisely, the use of constants describing the names of queries defined outside the class to which they belong, in Jakarta Enterprise annotations, is not customary for the tool and, in order to be consistent with the specified objective, i.e. to achieve the absence of problems, following the concept of Fault Avoidance through Model Checking, it was decided to carry out the following refactoring:

   ```
   @NamedQuery(name=TROVA_ANNO, query="SELECT a FROM AnnoDidattico a WHERE a.anno =
   ↪  :anno")
   ```

   becomes

   ```
   @NamedQuery(name = "AnnoDidattico.trovaAnno", query = "SELECT a FROM
   ↪  AnnoDidattico a WHERE a.anno = :anno")
   ```

2. **Removal of duplicates** (like more occurrences of the same attribute) – During monitoring, OpenJML returned issues in the logs relating to the erroneous presence of duplicates of the same attributes. The duplicate in question was as follows:

   ```
   /**
    * Nome del Corso
    * */
   private String nome;
   ```

3. **Modifying Query** – From the analysis of the results of the various OpenJML executions on the DAO Coordinator, an unused query was found that does not return the desired value.

   ```
   public List<Coordinatore> trovaCoordinatoriCorsoLaurea(String nomeCorsoLaurea) {
       TypedQuery<Coordinatore> query =
       ↪  emUniClass.createQuery(Coordinatore.TROVA_COORDINATORE,
       ↪  Coordinatore.class);
       query.setParameter("nomeCorsoLaurea", nomeCorsoLaurea);
       return query.getResultList();
   }
   ```

   becomes

```
public List<Coordinatore> trovaCoordinatoriCorsoLaurea(String nomeCorsoLaurea) {
    TypedQuery<Coordinatore> query =
    ↪  emUniClass.createQuery(Coordinatore.TROVA_COORDINATORE_CORSOLAUREA,
    ↪  Coordinatore.class);
    query.setParameter("nomeCorsoLaurea", nomeCorsoLaurea);
    return query.getResultList();
}
```

3.4 Problems Encountered

As mentioned above, in enterprise environments and with today's technologies, OpenJML unfortunately fails to deliver the expected results, causing various problems due to incompatibility and the lack of testing in enterprise environments: if certain features, such as CDI, the use of containers and values or objects defined only at runtime, and other features that cannot be analysed statically or through symbolic execution, are seen as errors, then the use of certain libraries or methods defined in the Jakarta framework, such as JPA, JTA, etc., are also seen as errors due to incompatibility. Due to the lack of compatibility with all the technologies used to create the project in question, it was not possible to analyse the toString (problem with the concatenation of potentially null values, due to the compatibility issues defined above) and DAO/Services, due to the lack of Model Checking on values given dynamically, at runtime, starting from the InitialContext, given thanks to Dependency Injection, which is not compatible with the static tool. Furthermore, even the use of OpenJML's Runtime Assertion Checking modality did not allow for adequate results for DAOs and Services for the same reason, namely incompatibility with the Jakarta infrastructure: the presence of components managed entirely by the container, the extensive use of Dependency Injection, the container-managed lifecycle and enterprise APIs prevents reliable bytecode instrumentation and does not allow consistent runtime assertion from points where components that are only evaluated at runtime come into play.

3.5 Summary

Unfortunately, there are currently no Jakarta-compatible tools for (real[1]) Model Checking that are compatible with the dynamic nature of the specification framework, but its use has nevertheless been useful for understanding the introduction to Software Fault Avoidance and Model Checking, static and dynamic analysers and their differences, limitations and refactoring defined in the list in *Implementation*.

[1]There are several tools that can be used, such as Jakarta Bean Validation and Google Guava Checker, but none of these are true Model Checkers, as they are only for entering preconditions, rather than performing exhaustive checks of states and invariants, and they only work at runtime through Interceptors.

4 CI/CD Pipeline for Dependability Assurance

4.1 Motivation and Context

As highlighted in Chapter 1, the original UniClass project lacked fundamental software dependability practices.

In particular, the system was characterized by the absence of automated testing, missing continuous integration workflows, limited security enforcement, and insufficient technical documentation.

Such limitations prevented the project from satisfying key dependability attributes, including reliability, maintainability, and security. To address these shortcomings, a comprehensive **CI/CD pipeline** was designed and introduced as a first-class architectural component of the development process.

Rather than treating testing and security as auxiliary activities, the pipeline was conceived as an **automated enforcement mechanism**, ensuring that every code change is systematically validated against predefined quality, security, and performance criteria.

4.2 Pipeline Objectives

The CI/CD pipeline was designed with the explicit goal of improving the overall dependability of the UniClass platform. Its objectives extend beyond basic build automation and include:

- **Reliability Assurance** – early detection of functional regressions through automated testing.
- **Security Enforcement** – systematic identification of vulnerabilities, insecure dependencies, and leaked secrets.
- **Quality Governance** – enforcement of static analysis rules, code coverage thresholds, and technical debt limits.
- **Performance Awareness** – detection of performance through targeted microbenchmarks.
- **Traceability and Auditability** – generation of verifiable artifacts and reports for every pipeline execution.

The pipeline embodies a *Shift-Left* philosophy, ensuring that defects are detected as early as possible in the development lifecycle.

4.3 CI/CD Architecture Overview

The pipeline is implemented using **GitHub Actions** and is fully integrated with the project repository to automate the verification and validation of each distribution. Each workflow execution represents a controlled experiment that evaluates whether a given code revision satisfies the system's dependability requirements.

To ensure maintainability and a clear separation of concerns, the automation process utilizes a modular architecture split across three distinct workflow files stored in the `.github/workflows` directory:

- `maven-ci.yml`: Handles build, compilation, and automated testing.
- `security-scan.yml`: Dedicated to vulnerability scanning and secret detection.
- `sonar-analysis.yml`: Manages static code analysis and quality gates.

4.3.1 Operational Security and Secrets Management

A critical component of the CI/CD architecture is the secure handling of sensitive credentials and authentication tokens, ensuring that no confidential information is hardcoded within the source code (Hardcoded Credential prevention).

The operational security is managed through a layered approach:

- **Encrypted Secrets Store:** Sensitive data such as the `SONAR_TOKEN` (required for static analysis reporting) and database credentials are stored in the GitHub Repository Secrets vault. These are encrypted at rest and only exposed to the runner environment during execution.
- **Runtime Injection via Environment Variables:** During the build process, secrets are injected into the containerized runner as environment variables (e.g., passing `${ secrets.SONAR_TOKEN }` to the Maven build context). This ensures that secrets exist in memory only for the duration of the job.
- **Ephemeral Access Tokens:** For internal repository operations (such as checking out code or posting PR comments), the pipeline utilizes the automatic `GITHUB_TOKEN`. This is a short-lived, rotation-free token with scoped permissions generated specifically for the single workflow run, minimizing the attack surface in case of leakage.
- **Log Masking:** The CI execution engine automatically masks known secrets in the console logs, preventing accidental exposure during debugging or audit phases.

This configuration adheres to the *Least Privilege* principle and ensures that the automation layer maintains high dependability without compromising the confidentiality of the project's infrastructure.

4.3.2 Execution Triggers

Pipeline executions are automatically triggered on:

- Every `push` to the `main` and `develop` branches.
- Every `pull_request`, acting as a merge guard to prevent potentially unsafe code from entering the production baseline.

4.3.3 Pipeline Stages and Specifications

The CI/CD workflow is structured into logically separated stages, each targeting a specific dependability dimension. A failure in any stage results in an immediate pipeline interruption (No-Go decision).

1. Build and Test Stage

This stage, defined in `maven-ci.yml`, ensures the functional correctness of the application. It operates in a standardized ephemeral environment (Ubuntu-latest with Java 17 Temurin distribution) and leverages caching mechanisms for Maven dependencies to optimize execution time.

Automated Unit Testing Strategy Automated testing is a core pillar of the pipeline, and it is included in the build pipeline. We adopted a **white-box testing** strategy to verify the internal structure of the components. The specific quality criteria considered were statement coverage, branch coverage, and cyclomatic complexity.

The execution focuses on Unit Tests using **JUnit 5** and **Mockito**, specifically designed to validate the core business logic of Models, DAOs, and Servlets without dependencies on external systems. Strict quantitative metrics are enforced:

- **Zero tolerance** for failing unit tests.
- code coverage threshold verification via **JaCoCo** reports.

2. Security Enforcement Stage

Defined in `security-scan.yml`, this stage treats security checks as non-negotiable gates. Parallel execution of industry-standard tools ensures comprehensive protection:

- **Dependency Vulnerability Scanning (Snyk):** Performs Software Composition Analysis (SCA) on Maven and Node.js manifests. The pipeline is configured to fail if *High* or *Critical* vulnerabilities (CVEs) are detected in third-party libraries.
- **Secret Detection (GitGuardian):** Scans the repository history and current diffs to prevent "secret sprawl." Any detection of hard-coded credentials (API keys, tokens) results in immediate failure.

3. Continuos feedback: Quality and Static Analysis Stage

The `sonar-analysis.yml` workflow integrates **SonarCloud** to enforce static analysis rules aligned with the strict *SonarWay* Quality Gate. This stage serves as a control mechanism for technical debt, obtaining **continuos feedback** about:

- Absence of BLOCKER and CRITICAL issues.
- Verification of reliability (bugs) and maintainability (code smells).
- Import and visualization of the JaCoCo coverage reports generated in the build stage.

4.3.4 Continuous Verification and Quality Gating

To ensure the integrity and dependability of the software artifacts deployed to the production environment, the CI/CD pipeline has been strengthened through strict **Continuous Verification** policies. This approach shifts the focus from simple automated builds to a rigorous validation process that prevents the integration of non-compliant code.

Branch Protection Strategy

A **Branch Protection Rule** has been enforced on the `main` branch, which acts as the project's "golden source." This configuration explicitly disables direct commits, mandating the use of Pull Requests (PRs) for all code integration. Every modification must therefore undergo mandatory review and successfully pass all automated pipeline checks—including unit tests and security scans—before it can be merged.

This mechanism effectively eliminates the risk of accidental breaking changes or unchecked code reaching the production branch, reinforcing the reliability and governance of the development workflow.

Synchronous Quality Gate Enforcement

A critical enhancement to the DevSecOps workflow is the strict enforcement of the **SonarCloud Quality Gate**. The `sonar-analysis` workflow has been re-engineered to operate synchronously with the quality assessment results.

By enabling the parameter:

```
sonar.qualitygate.wait=true
```

the CI pipeline is configured to halt and fail immediately if the analyzed code does not meet the defined quality standards, acting as a blocker action. This ensures an immediate feedback loop and blocks the merging of any Pull Request that fails the gate, preventing the introduction of invalid distributions into the codebase.

The "Clean as You Code" Policy

The Quality Gate is configured to focus on **New Code** (i.e., changes introduced in the current PR), enforcing a **Clean as You Code** DevOps policy. Rather than addressing only legacy technical debt, this policy ensures that all new development adheres to the highest standards.

The enforced criteria include:

- **Reliability:** 0 New Bugs.
- **Security:** 0 New Vulnerabilities and 0 New Security Hotspots.
- **Maintainability:** Technical Debt Ratio on new code strictly within Rating A.
- **Test Coverage:** Minimum of 80% coverage on new lines of code.

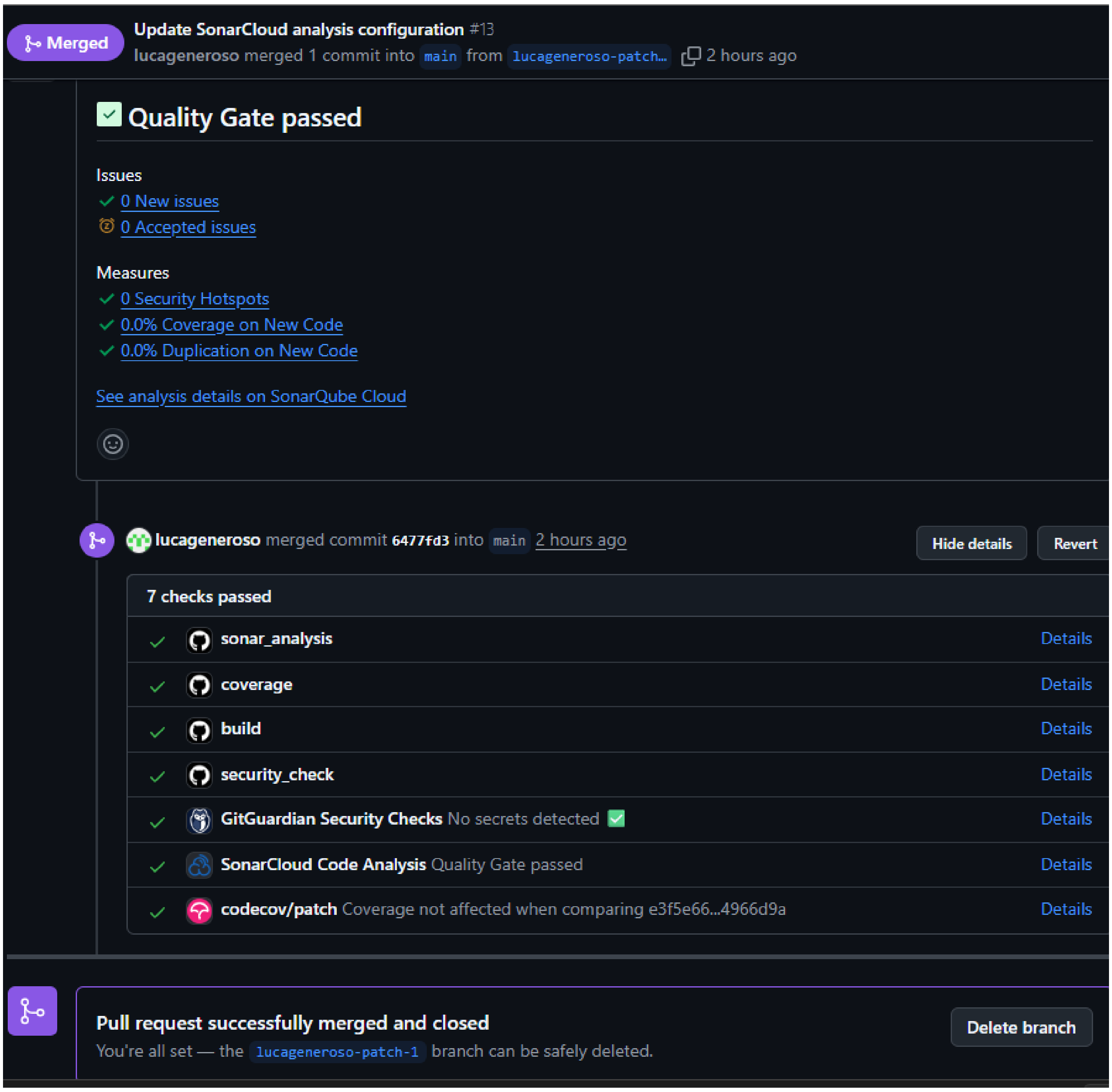
This proactive validation strategy ensures that the project's overall maintainability and reliability metrics improve monotonically over time, as no regression or technical debt is allowed to enter the `main` branch. The pipeline thus acts as a continuous enforcement mechanism for software quality and dependability.

4.3.5 Artifacts, Feedback Loop, and Traceability

Each pipeline execution establishes a closed feedback loop, producing a set of verifiable artifacts:

- Test execution reports and Code coverage summaries.
- SonarCloud dashboards for debt visualization.
- Security audit reports from Snyk and GitGuardian.

This mechanism ensures full traceability: failures block pull requests, issues are automatically traced to specific commits, and developers are provided with actionable logs to facilitate immediate remediation. This process transforms UniClass from a manually validated project into a systematically controlled software system, enforcing a "dependability-by-construction" culture.


Merged **Update SonarCloud analysis configuration** #13
lucageneroso merged 1 commit into `main` from `lucageneroso-patch…` 2 hours ago

Quality Gate passed

Issues
✓ 0 New issues
0 Accepted issues

Measures
✓ 0 Security Hotspots
✓ 0.0% Coverage on New Code
✓ 0.0% Duplication on New Code

See analysis details on SonarQube Cloud

lucageneroso merged commit **6477fd3** into `main` 2 hours ago

7 checks passed

✓	**sonar_analysis**	Details
✓	**coverage**	Details
✓	**build**	Details
✓	**security_check**	Details
✓	**GitGuardian Security Checks** No secrets detected ✓	Details
✓	**SonarCloud Code Analysis** Quality Gate passed	Details
✓	**codecov/patch** Coverage not affected when comparing e3f5e66...4966d9a	Details

Pull request successfully merged and closed
You're all set — the `lucageneroso-patch-1` branch can be safely deleted.

Figure 4.1: GitHub Pull Request checks demonstrating the Continuous Verification process. The merge is blocked until all pipelines (Build, Security, Quality Gate) pass successfully.

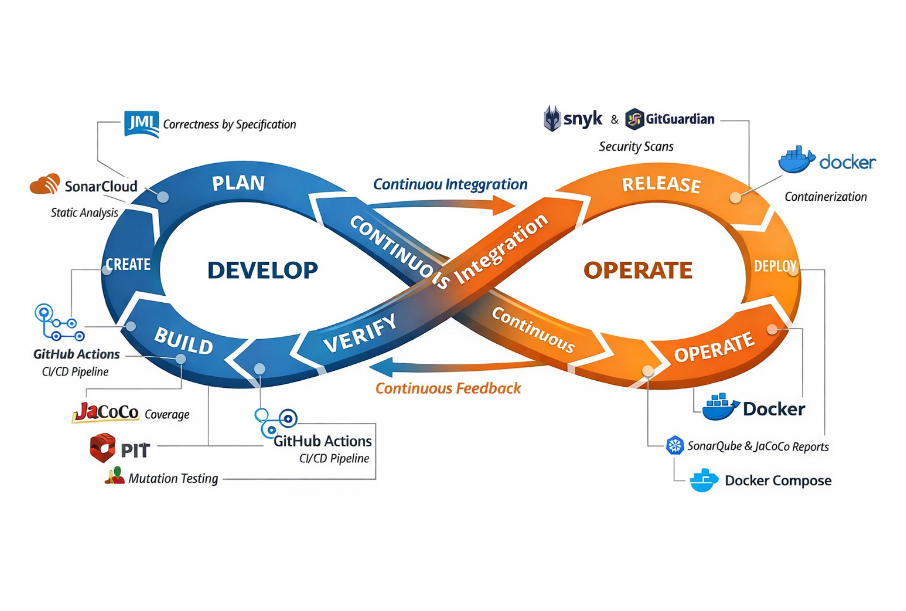


Figure 4.2: DevSecOps CI/CD pipeline adopted in the UniClass project.

5 Testing

5.1 Introduction and Testing Strategy

In the context of the project architecture, testing represents the primary mechanism for *fault detection*, operating synergistically and complementarily to the *fault avoidance* techniques adopted in the preliminary phases, such as formal specification via JML. While formal specification allows for the prevention of errors by explicitly defining system correctness constraints a priori, testing is tasked with verifying that the concrete implementation respects these constraints in a real execution context.

The primary objective of the testing activity is, therefore, the validation of the system's functional correctness and the observation of the behavior of its critical components in the presence of heterogeneous inputs. This objective is pursued through the definition of a systematic, fully automated, and reproducible test suite.

Testing activities, initially absent in the original version of the project, were introduced *ex novo* and focused mainly on the application layer and the data access layer (DAO), i.e., the levels where the highest concentration of functional and decision-making logic resides.

The entire verification process has been integrated into the automated build cycle via Apache Maven. Test cases were implemented using JUnit 5 as the reference framework, while Mockito was employed for the isolation of external dependencies. Based on the test suite thus defined, advanced quantitative analyses were conducted, including structural code coverage measurement and mutation testing, to rigorously evaluate the effectiveness and robustness of the implemented tests.

5.2 Testing Methodology

5.2.1 Unit Testing

The project includes exclusively unit tests, designed to verify the behavior of single code units in isolation. Test design followed the Category Partition Testing technique, which allowed for a systematic identification of relevant input classes, invalid cases, and boundary conditions.

The activity focused on two main areas:

- Servlet (Control Layer): Tested by simulating HTTP requests and responses.
- DAO Components (Data Access Layer): Verified by isolating access to external resources (database).

To ensure adequate isolation, extensive use of Mockito was made to simulate dependencies. Bean classes were not tested directly, as they implement purely structural logic (getters and setters) and their behavior is exercised indirectly by Servlet and DAO tests.

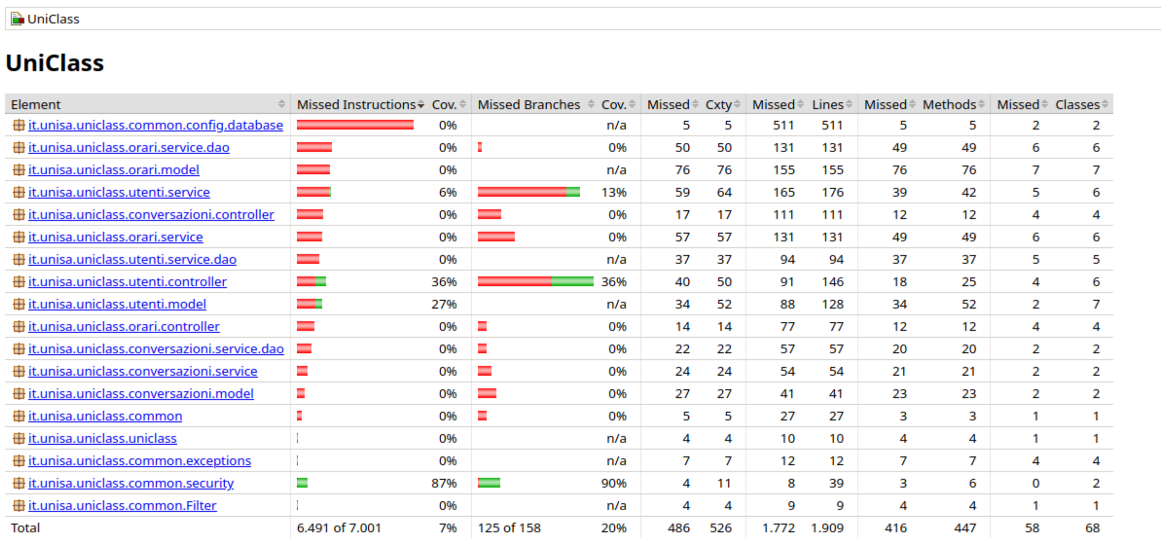
5.2.2 Coverage Analysis (JaCoCo & Codecov)

The implementation of a systematic code coverage measurement responds to the methodological need to quantify the adequacy of the verification process. In highly complex software systems, the execution of the test suite does not guarantee that all logical branches and critical flows have effectively been tested. Therefore, the adoption of coverage metrics bridges the gap between the subjective perception of reliability and actual test effectiveness (test adequacy), providing an objective indicator essential for identifying unverified code areas and mitigating the risk of regressions during software evolution phases.

- Quality Gate (JaCoCo): Metric collection is integrated into the Maven build cycle via the JaCoCo plugin.
- Codecov: Data produced by JaCoCo is published on the Codecov platform, which offers a centralized dashboard to visualize historical trends and per-file coverage.

Initial Situation

In the initial phase of the project, prior to the introduction of an automated test suite, code coverage analysis highlighted a situation of null or negligible coverage. The absence of tests made it impossible to objectively evaluate the degree of exploration of the code's logical branches, leaving the system particularly exposed to the risk of latent faults and undetected regressions.



UniClass

UniClass

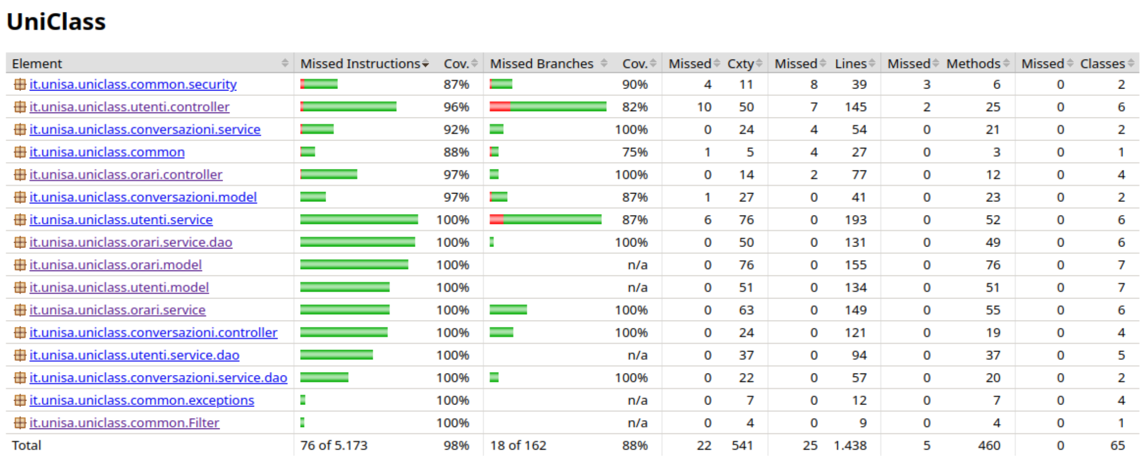
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
it.unisa.uniclass.common.config.database		0%		n/a	5	5	511	511	5	5	2	2
it.unisa.uniclass.orari.service.dao		0%		0%	50	50	131	131	49	49	6	6
it.unisa.uniclass.orari.model		0%		n/a	76	76	155	155	76	76	7	7
it.unisa.uniclass.utenti.service		6%		13%	59	64	165	176	39	42	5	6
it.unisa.uniclass.conversazioni.controller		0%		0%	17	17	111	111	12	12	4	4
it.unisa.uniclass.orari.service		0%		0%	57	57	131	131	49	49	6	6
it.unisa.uniclass.utenti.service.dao		0%		n/a	37	37	94	94	37	37	5	5
it.unisa.uniclass.utenti.controller		36%		36%	40	50	91	146	18	25	4	6
it.unisa.uniclass.utenti.model		27%		n/a	34	52	88	128	34	52	2	7
it.unisa.uniclass.orari.controller		0%		0%	14	14	77	77	12	12	4	4
it.unisa.uniclass.conversazioni.service.dao		0%		0%	22	22	57	57	20	20	2	2
it.unisa.uniclass.conversazioni.service		0%		0%	24	24	54	54	21	21	2	2
it.unisa.uniclass.conversazioni.model		0%		0%	27	27	41	41	23	23	2	2
it.unisa.uniclass.common		0%		0%	5	5	27	27	3	3	1	1
it.unisa.uniclass.uniclass		0%		n/a	4	4	10	10	4	4	1	1
it.unisa.uniclass.common.exceptions		0%		n/a	7	7	12	12	7	7	4	4
it.unisa.uniclass.common.security		87%		90%	4	11	8	39	3	6	0	2
it.unisa.uniclass.common.Filter		0%		n/a	4	4	9	9	4	4	1	1
Total	6.491 of 7.001	7%	125 of 158	20%	486	526	1.772	1.909	416	447	58	68

Figure 5.1: *JaCoCo report in the initial project phase, lacking significant coverage.*

Final Results and Quality Gate

Following the systematic introduction of unit tests, code coverage measurement was made an integral part of the build process through the integration of the JaCoCo plugin into the Maven lifecycle.

The final results show a marked improvement compared to the initial situation, with coverage values amply exceeding the imposed minimum thresholds, testifying to the adequacy of the test suite and its ability to extensively exercise the application logic.



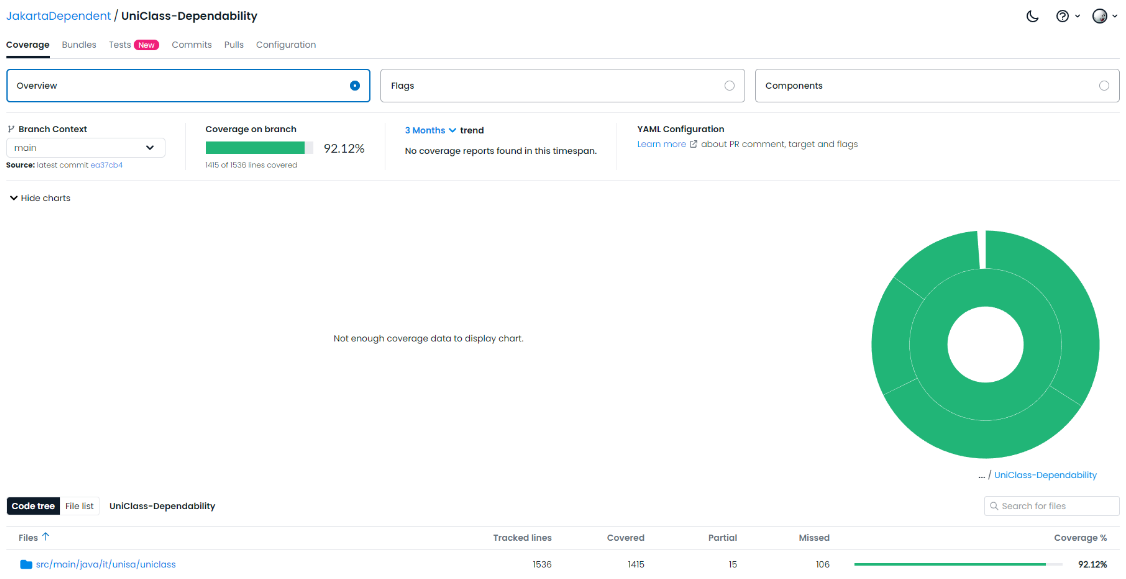
UniClass

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
it.unisa.uniclass.common.security		87%		90%	4	11	8	39	3	6	0	2
it.unisa.uniclass.utenti.controller		96%		82%	10	50	7	145	2	25	0	6
it.unisa.uniclass.conversazioni.service		92%		100%	0	24	4	54	0	21	0	2
it.unisa.uniclass.common		88%		75%	1	5	4	27	0	3	0	1
it.unisa.uniclass.orari.controller		97%		100%	0	14	2	77	0	12	0	4
it.unisa.uniclass.conversazioni.model		97%		87%	1	27	0	41	0	23	0	2
it.unisa.uniclass.utenti.service		100%		87%	6	76	0	193	0	52	0	6
it.unisa.uniclass.orari.service.dao		100%		100%	0	50	0	131	0	49	0	6
it.unisa.uniclass.orari.model		100%		n/a	0	76	0	155	0	76	0	7
it.unisa.uniclass.utenti.model		100%		n/a	0	51	0	134	0	51	0	7
it.unisa.uniclass.orari.service		100%		100%	0	63	0	149	0	55	0	6
it.unisa.uniclass.conversazioni.controller		100%		100%	0	24	0	121	0	19	0	4
it.unisa.uniclass.utenti.service.dao		100%		n/a	0	37	0	94	0	37	0	5
it.unisa.uniclass.conversazioni.service.dao		100%		100%	0	22	0	57	0	20	0	2
it.unisa.uniclass.common.exceptions		100%		n/a	0	7	0	12	0	7	0	4
it.unisa.uniclass.common.Filter		100%		n/a	0	4	0	9	0	4	0	1
Total	76 of 5.173	98%	18 of 162	88%	22	541	25	1.438	5	460	0	65

Figure 5.2: *Final JaCoCo report with high coverage and quality gate passed.*

Codecov

Coverage data was also published on the Codecov platform, which provides a centralized dashboard for analyzing coverage on a single-file basis and monitoring the temporal evolution of the verification level.



Branch Context: main
Source: latest commit ea37cb4

Coverage on branch 92.12%
1415 of 1536 lines covered

3 Months trend
No coverage reports found in this timespan.

YAML Configuration
Learn more about PR comment, target and flags

Not enough coverage data to display chart.

Files	Tracked lines	Covered	Partial	Missed	Coverage %
src/main/java/it/unisa/uniclass	1536	1415	15	106	92.12%

Figure 5.3: Codecov report dashboard

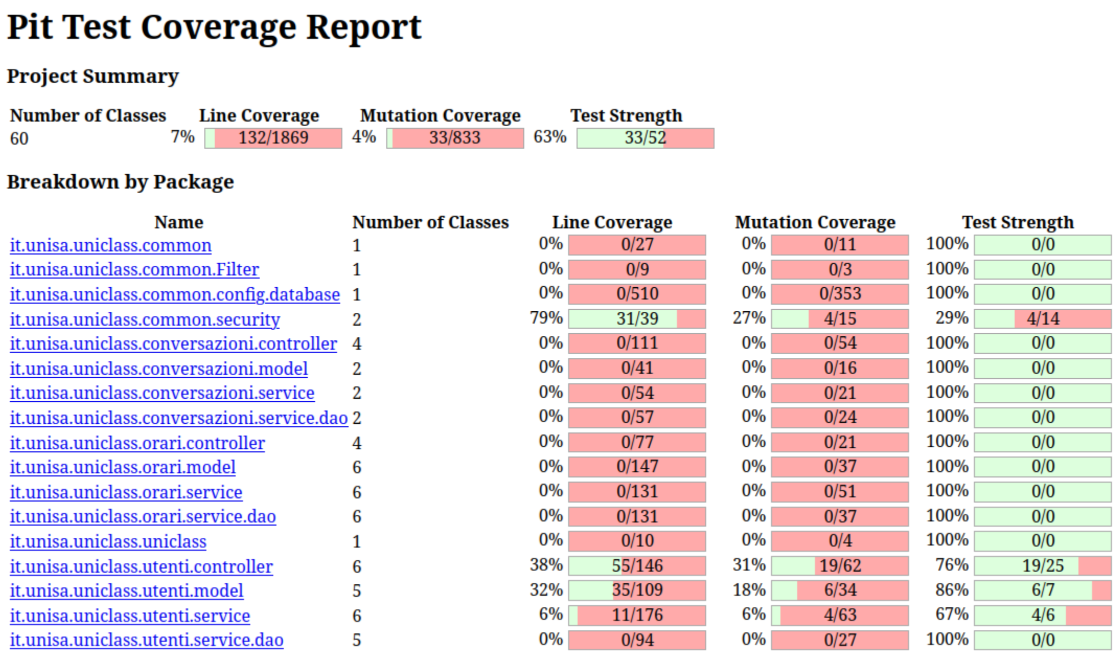
5.2.3 Mutation Testing (PiTest)

To evaluate the robustness of the test suite—going beyond the structural coverage metric which guarantees code execution but not necessarily correctness—Mutation Testing was adopted using the PiTest tool. This advanced technique operates by deliberately introducing syntactic or logical alterations ("mutants") into the compiled code, such as inverting logical conditions, modifying arithmetic constants, or removing method calls. The goal is to verify if the test suite is sufficiently sensitive to detect such anomalies: if tests continue to pass despite the mutation, the mutant "survives," indicating a gap in verification; if they fail, the mutant is "killed," confirming the test case's effectiveness.

- Minimum Mutation Score (50%): Requires that at least half of the generated mutants be effectively detected, forcing the writing of precise and non-generic assertions.
- Minimum Mutation Coverage (80%): Ensures that mutation analysis is applied to a significant portion of the source code, preventing critical areas from being excluded from in-depth verification.

Initial Analysis

The first execution of mutation testing highlighted the presence of numerous surviving mutants, indicating that, despite high structural coverage, some test cases were insufficiently specific or lacked assertions capable of intercepting semantic alterations in code behavior.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
60	7%	132/1869	4%	33/833	63%	33/52

Breakdown by Package

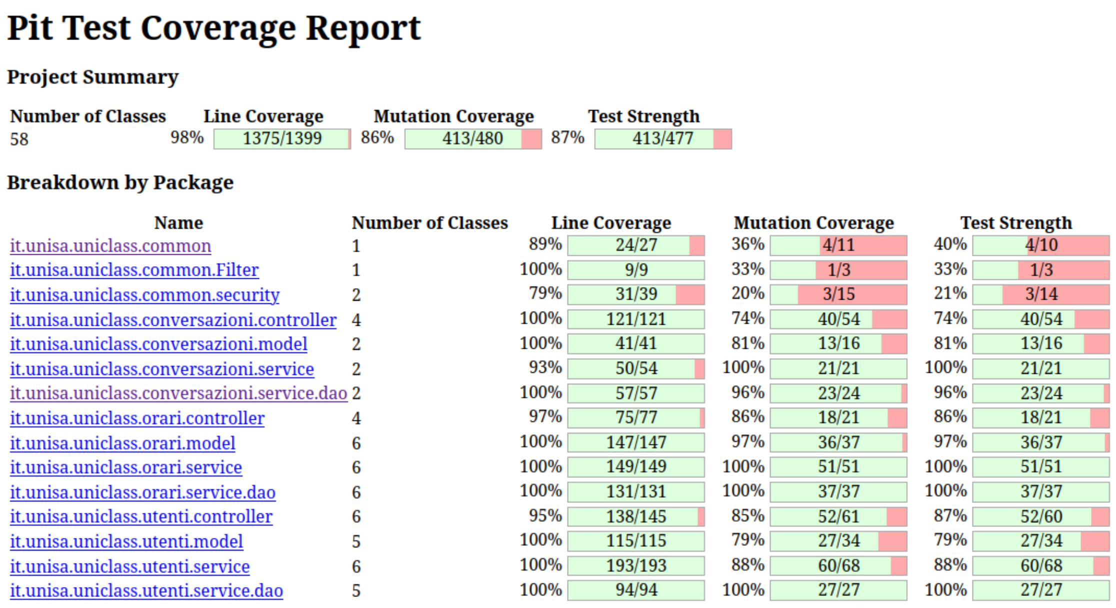
Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
it.unisa.uniclass.common	1	0%	0/27	0%	0/11	100%	0/0
it.unisa.uniclass.common.Filter	1	0%	0/9	0%	0/3	100%	0/0
it.unisa.uniclass.common.config.database	1	0%	0/510	0%	0/353	100%	0/0
it.unisa.uniclass.common.security	2	79%	31/39	27%	4/15	29%	4/14
it.unisa.uniclass.conversazioni.controller	4	0%	0/111	0%	0/54	100%	0/0
it.unisa.uniclass.conversazioni.model	2	0%	0/41	0%	0/16	100%	0/0
it.unisa.uniclass.conversazioni.service	2	0%	0/54	0%	0/21	100%	0/0
it.unisa.uniclass.conversazioni.service.dao	2	0%	0/57	0%	0/24	100%	0/0
it.unisa.uniclass.orari.controller	4	0%	0/77	0%	0/21	100%	0/0
it.unisa.uniclass.orari.model	6	0%	0/147	0%	0/37	100%	0/0
it.unisa.uniclass.orari.service	6	0%	0/131	0%	0/51	100%	0/0
it.unisa.uniclass.orari.service.dao	6	0%	0/131	0%	0/37	100%	0/0
it.unisa.uniclass.uniclass	1	0%	0/10	0%	0/4	100%	0/0
it.unisa.uniclass.utenti.controller	6	38%	55/146	31%	19/62	76%	19/25
it.unisa.uniclass.utenti.model	5	32%	35/109	18%	6/34	86%	6/7
it.unisa.uniclass.utenti.service	6	6%	11/176	6%	4/63	67%	4/6
it.unisa.uniclass.utenti.service.dao	5	0%	0/94	0%	0/27	100%	0/0

Figure 5.4: Initial PiTest report with surviving mutants.

Refinement of the Test Suite

The analysis of surviving mutants guided an iterative process of refining the test suite, leading to the introduction of more precise assertions, the coverage of edge cases, and a more rigorous characterization of the expected behavior of critical components.

The process was repeated until the defined quality gates were reached: a minimum *mutation score* of 50% and a minimum *mutation coverage* of 80%, both fully satisfied in the final configuration.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
58	98%	1375/1399	86%	413/480	87%	413/477

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
it.unisa.uniclass.common	1	89%	24/27	36%	4/11	40%	4/10
it.unisa.uniclass.common.Filter	1	100%	9/9	33%	1/3	33%	1/3
it.unisa.uniclass.common.security	2	79%	31/39	20%	3/15	21%	3/14
it.unisa.uniclass.conversazioni.controller	4	100%	121/121	74%	40/54	74%	40/54
it.unisa.uniclass.conversazioni.model	2	100%	41/41	81%	13/16	81%	13/16
it.unisa.uniclass.conversazioni.service	2	93%	50/54	100%	21/21	100%	21/21
it.unisa.uniclass.conversazioni.service.dao	2	100%	57/57	96%	23/24	96%	23/24
it.unisa.uniclass.orari.controller	4	97%	75/77	86%	18/21	86%	18/21
it.unisa.uniclass.orari.model	6	100%	147/147	97%	36/37	97%	36/37
it.unisa.uniclass.orari.service	6	100%	149/149	100%	51/51	100%	51/51
it.unisa.uniclass.orari.service.dao	6	100%	131/131	100%	37/37	100%	37/37
it.unisa.uniclass.utenti.controller	6	95%	138/145	85%	52/61	87%	52/60
it.unisa.uniclass.utenti.model	5	100%	115/115	79%	27/34	79%	27/34
it.unisa.uniclass.utenti.service	6	100%	193/193	88%	60/68	88%	60/68
it.unisa.uniclass.utenti.service.dao	5	100%	94/94	100%	27/27	100%	27/27

Figure 5.5: Final PiTest report with high mutation score and quality gates passed.

5.2.4 Performance Testing (JMH)

The performance evaluation of the UniClass system was conducted through the adoption of rigorous microbenchmarks using the Java Microbenchmark Harness (JMH). This framework was specifically selected for its ability to neutralize variabilities introduced by JVM runtime optimizations, such as Just-In-Time (JIT) compilation and Garbage Collection cycles.

The primary objective of this phase was not to execute systemic load tests to measure the overall infrastructure capacity, but rather to provide a punctual characterization of the computational costs associated with specific, critical components. This approach allowed for the isolation of the intrinsic algorithmic efficiency of the service layer, distinctly separating it from infrastructural latencies typically caused by I/O operations or network overhead.

For each tested component, the following metrics were collected and analyzed:

- Throughput ($ops/\mu s$): The number of operations performed per unit of time, indicating the system's raw processing capacity.
- Average Time ($\mu s/op$): The mean time required to execute a single operation once the system has reached a steady state.
- Sampling Time: A metric that collects individual execution samples to construct a latency distribution. This allows for the analysis of variance and percentiles, providing a more granular view of performance stability beyond the simple arithmetic mean.
- Single Shot Time: The execution time measured without a warmup phase ("cold start"), useful for evaluating the initialization costs before JVM optimizations take effect.

References and Data Availability

The complete experimental results and raw JSON data for the benchmarks discussed in this report are publicly available in the UniClass-Dependability/Benchmarks_JSON repository.

Experimental Setup

The performance evaluation was conducted using a controlled environment to ensure reproducibility and minimize external interference. All benchmarks were implemented using the **Java Microbenchmark Harness (JMH) framework version 1.35**, running on a **Java 21 (JDK 21.0.9)** runtime environment. Hardware/software configuration was different according to the stakeholder assigned to the target package. For Utenti and Conversazioni, the os was Windows 11, while for Orari was Arch.

JMH Configuration and Methodology

To obtain statistically significant results, the benchmark configuration was tuned to allow the Java Virtual Machine (JVM) to reach a steady state. This mitigates the effects of class loading and Just-In-Time (JIT) compilation during the initial phases.

The parameters used for the experiments are summarized as follows:

- **Output Time Unit:** Microseconds (μs) / Operations per second (ops/s).
- **Warmup Iterations:** 10 iterations. These are run to trigger JIT optimizations but are discarded from the final results.
- **Measurement Iterations:** 10 iterations. These are used to calculate the final aggregate metrics.
- **Forks:** 1. A single fork was used to simulate a persistent application state, though considering fresh JVM instances for each trial.
- **Threads:** 1. The benchmarks were executed in a single-threaded environment to measure raw algorithmic performance without the overhead of context switching or contention.

5.2.5 JVM Flags and Garbage Collection

The benchmarks were executed with the default Garbage Collector for Java 21 (G1GC). No specific heap size constraints (e.g., `-Xmx`) were enforced, allowing the JVM to manage memory dynamically based on the available system resources.

Benchmark Target Selection and Scope

The performance analysis strategy focused exclusively on the ****Service Layer**** of the UniClass architecture. This architectural decision was driven by the specific responsibility of this layer: acting as the transactional boundary between the presentation logic (Controllers) and the persistence layer (DAOs).

While Controllers primarily handle HTTP request parsing and DAOs execute raw database queries, the **Services** contain the core business logic, therefore, they represent the most significant candidates for **bottleneck identification**.

The benchmarking suite was organized into two logical clusters, targeting the most frequently accessed and data-intensive components of the system:

- **Authentication and User Management Cluster:**
  - *Target Classes:* `UtenteService`, `DocenteService`, `CoordinatoreService`, `PersonaleTAService`.
  - *Rationale:* The `login` and user retrieval methods represent the "Critical Path" of the application. Since every user session begins with these operations, any latency here directly impacts the perceived responsiveness of the entire system. High throughput is mandatory to support concurrent user access.
- **Academic Schedule Management Cluster (Orari):**
  - *Target Classes:* `RestoService`, **LezioneService**, **CorsoLaureaService**, **CorsoService**, **AulaService**.
  - *Rationale:* This component manages the retrieval of course timetables and lecture slots. As one of the most utilized features by students, it is subject to high-frequency read operations ("read-heavy" workload). Benchmarking this service ensures that the system can efficiently generate calendar views and filter time slots without bottlenecks, preventing slowdowns during peak access times (e.g., the start of the semester).
- **Communication and Data Flow Cluster:**
  - *Target Classes:* `TopicService`, `MessaggioService`.
  - *Rationale:* The communication subsystem involves handling lists of objects (topics and messages), sorting operations, and iterative logic. These operations are computationally more expensive ($O(n)$ complexity) compared to single-entity lookups. Benchmarking these services is crucial to detect regression in list processing and to measure the overhead of adding or retrieving messages in active conversations.

Each benchmark was implemented to isolate the method under test, using JMH states to prepare the necessary data structures pre-execution, ensuring that the measurements reflect the execution time of the business logic without the noise of object initialization.

Analysis of Package : Utenti

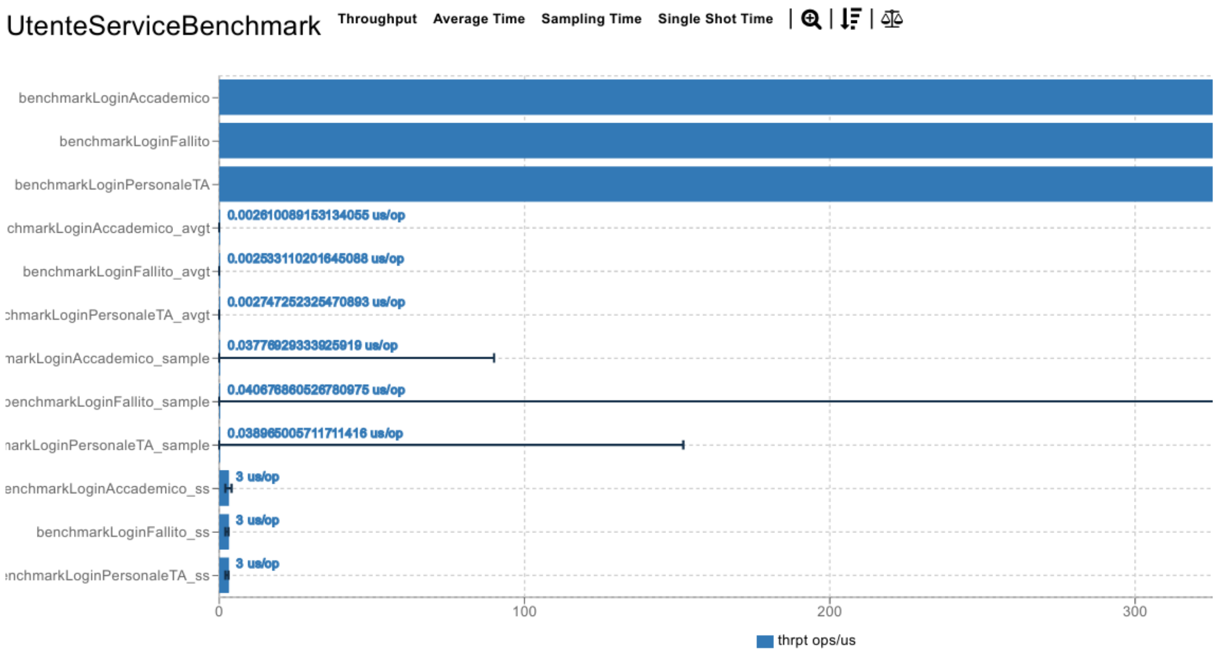


Figure 5.6: Utente Service Benchmark

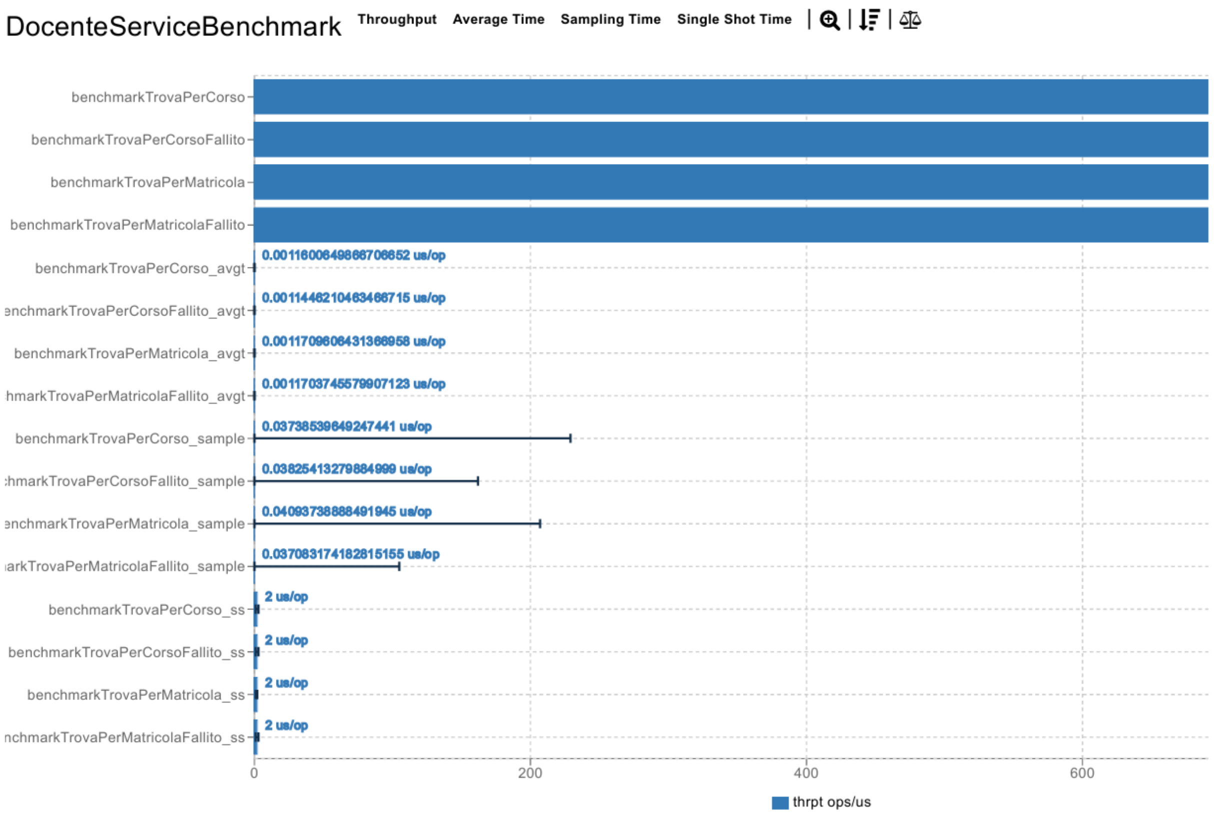


Figure 5.7: Docente Service Benchmark

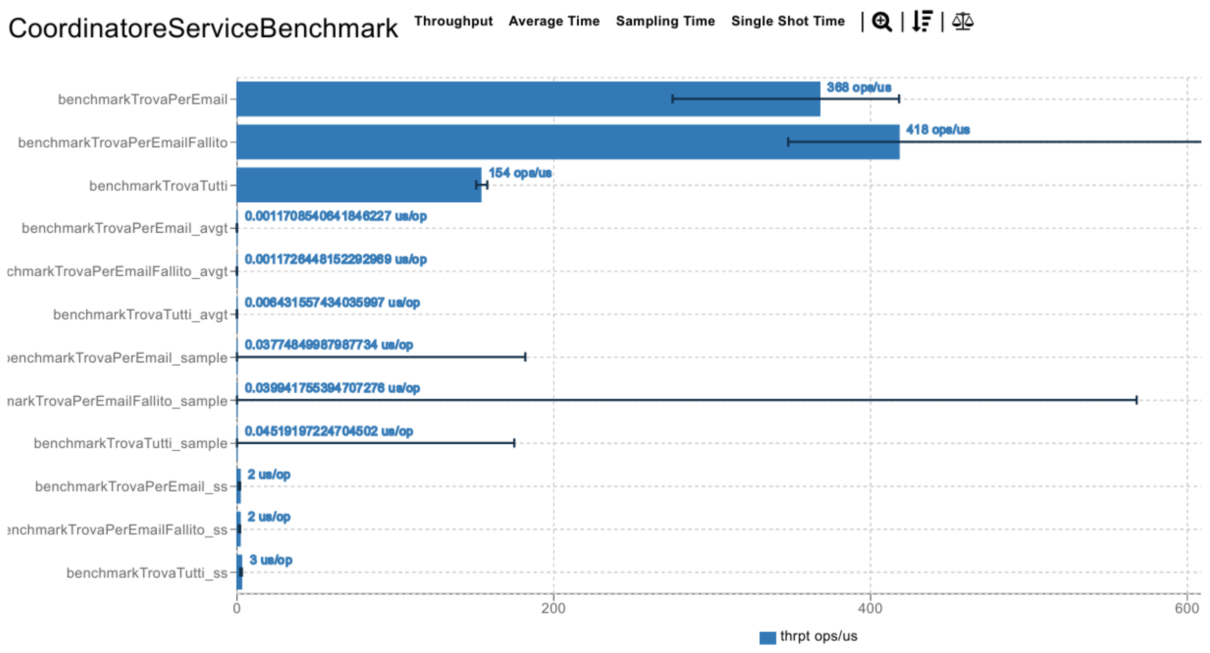


Figure 5.8: Coordinatore Service Benchmark

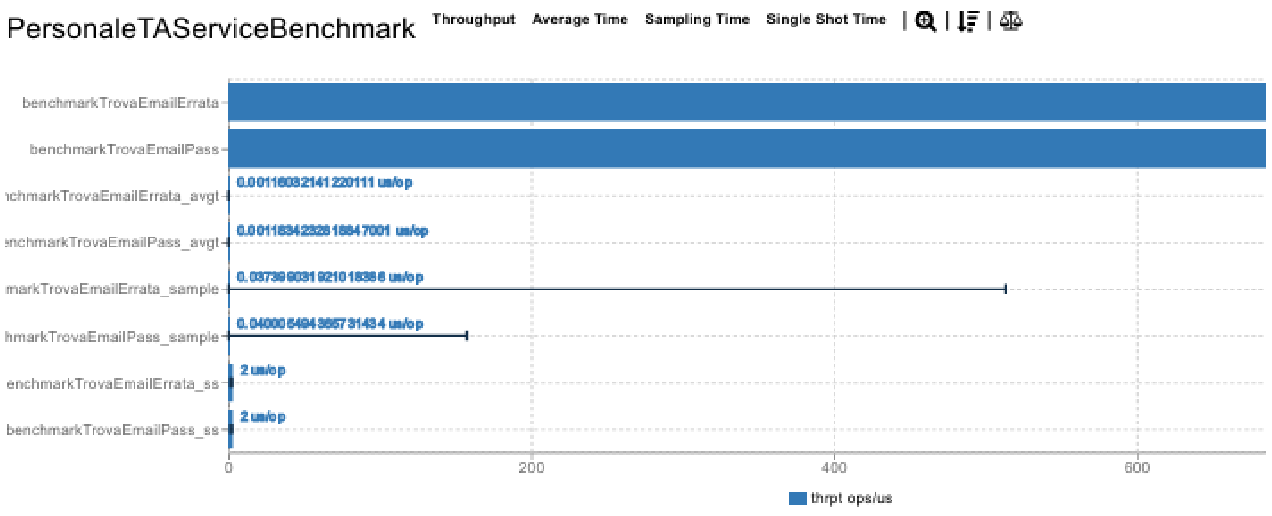


Figure 5.9: Personale Ata Service Benchmark

User Service Layer (Utenti, Docente, Coordinatore, Personale Ata)

The *Utenti* package is the most expensive package due to the intrinsic complexity of the software architecture, the technical debt present in the data model and in the hierarchy of related entities, which affect retrieval operations, and additional security methods. These methods, despite the efficiency of the SHA-256 algorithm compared to other algorithms such as ARGON, add nodes to the call graph related to user creation and access, reducing the number of possible operations per microsecond (μs).

In other words, the entity hierarchy is considered harmful to the number of generic operations per second, as it has to return objects with different relationships and, consequently, with different types of information for each retrieval.

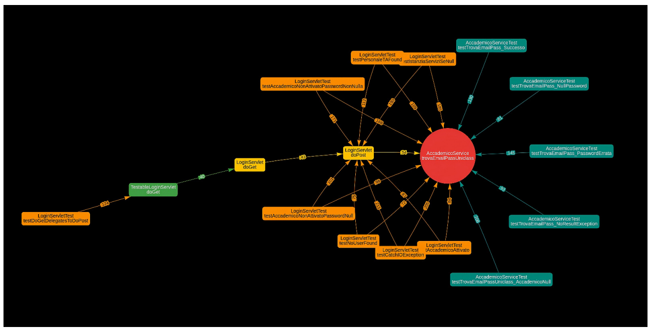


Figure 5.10: Call graph for the retrieval method "TrovaEmailPass" in AccademicoService

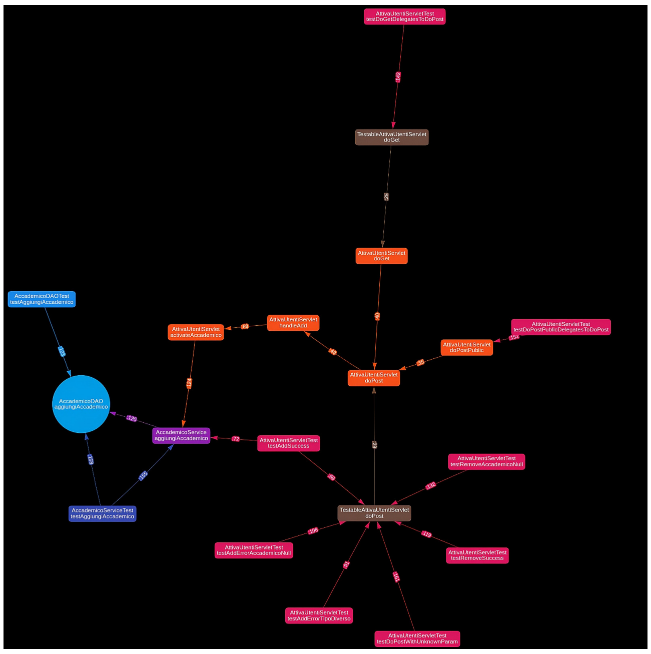


Figure 5.11: Call graph for "aggiungiAccademico" add operation in AccademicoService

From the Call Graphs, it is possible to note the substantial discrepancy between this package and the subsequent ones: the package in question, for each *Academic* method, calls more classes and a higher number of different methods, mainly due to the management of Services affected by evident technical debt.

It has been analyzed that, being a structural problem, it is not possible to remedy it through refactoring alone, but rather through a reengineering process.

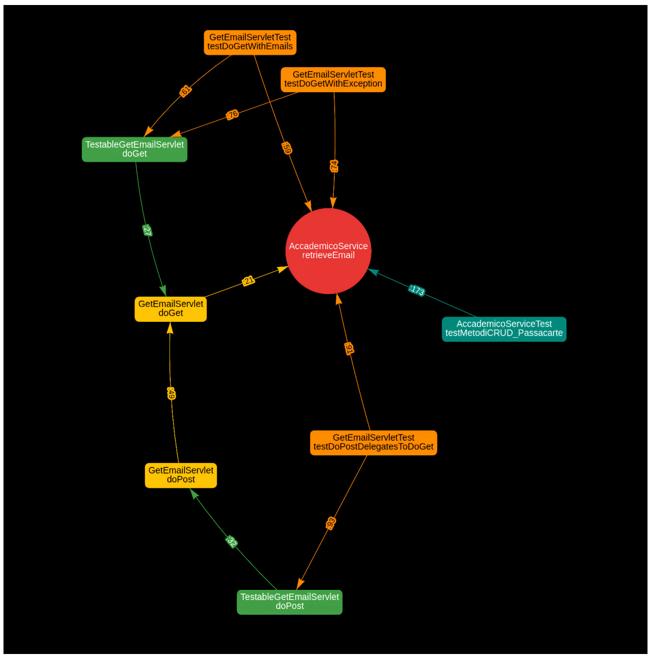


Figure 5.12: Call graph for the retrieval method retrieve By Email in AccademicoService

Analysis of the Package : Orari

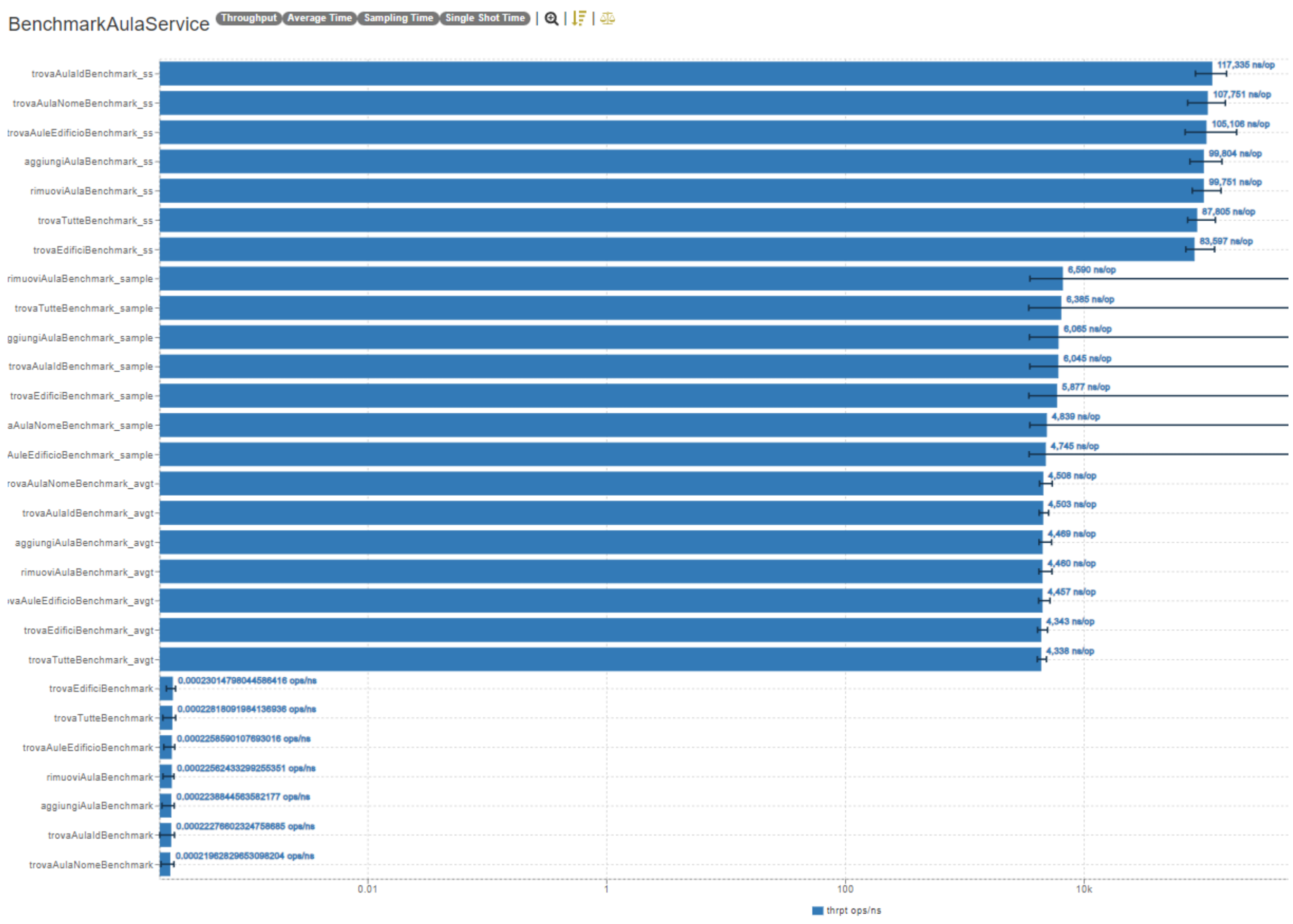


Figure 5.13: Aula Service Benchmark

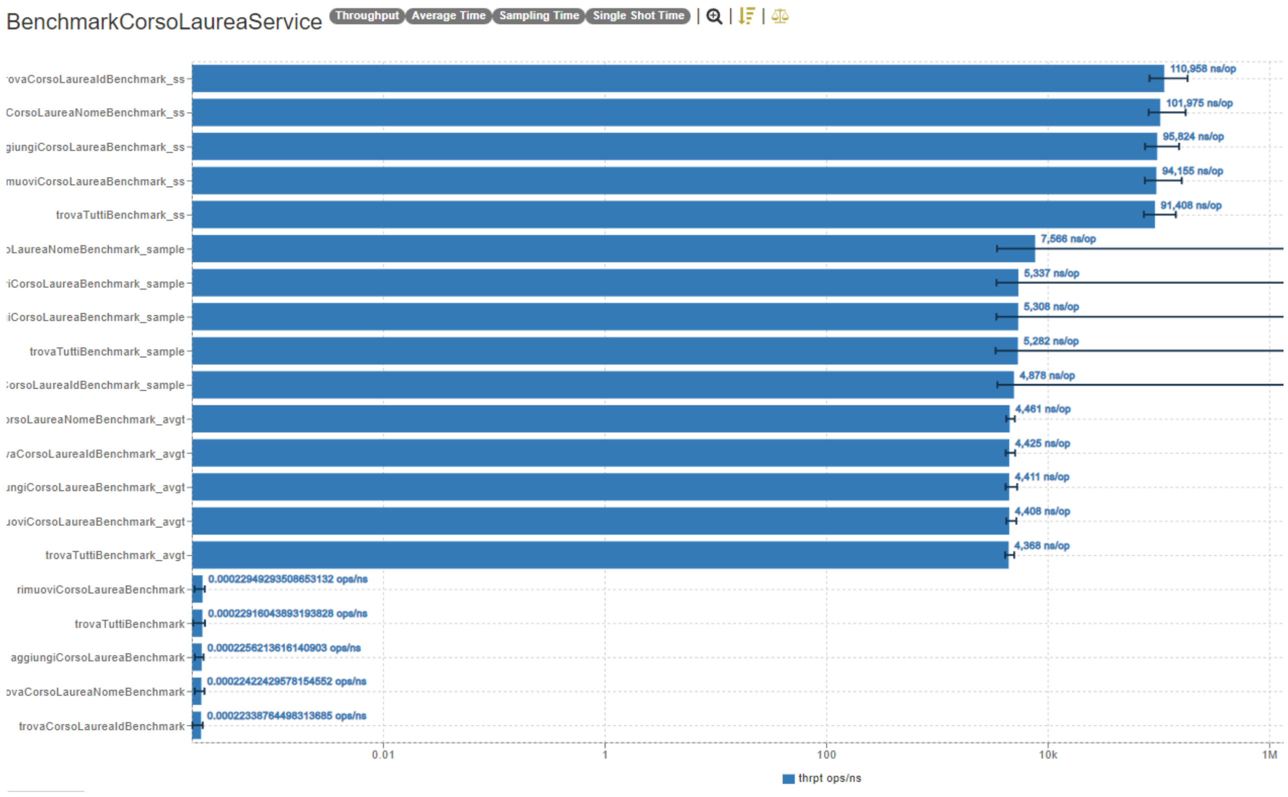


Figure 5.14: Corso laurea Benchmark

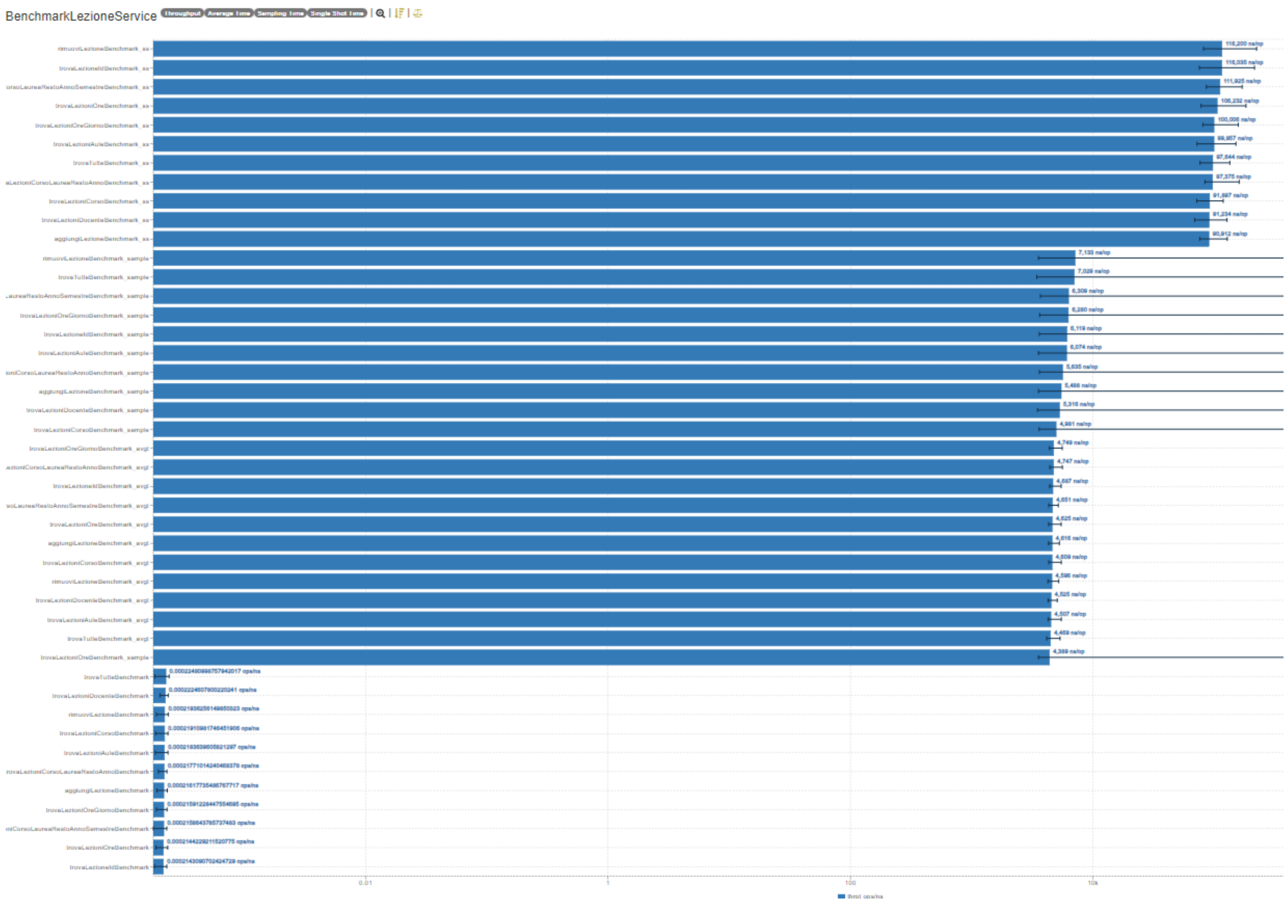


Figure 5.15: Lezione Benchmark

The performance of the *Orari* package has been evaluated through a microbenchmarking activity conducted using the Java Microbenchmark Harness (JMH). The analysis focused on the most representative methods involved in the management and retrieval of time-related data, with the goal of assessing their efficiency in terms of throughput and execution time.

Overall, the results show that the *Orari* package exhibits stable and predictable performance. The measured methods present a limited execution time per operation and a relatively high number of operations per second, indicating a low computational overhead. This behavior can be attributed to the simplicity of the underlying data structures and to a shallow entity hierarchy, which minimizes indirections and reduces the cost of object retrieval.

Each benchmarked method tends to interact with a restricted set of classes, avoiding deep service chains or redundant calls.

Furthermore, the absence of additional cross-cutting concerns, such as advanced security mechanisms or heavy validation logic, contributes to the overall efficiency of the package. The microbenchmarking results confirm that the package does not represent a performance bottleneck within the system and that its current design is well-suited for high-frequency access scenarios.

In conclusion, the *Orari* package demonstrates satisfactory performance characteristics under microbenchmarking analysis with JMH. Given the nature of the package and the obtained results, no critical optimizations are required at this stage, and the current implementation can be considered efficient and maintainable.

Analysis of Package : Conversazioni

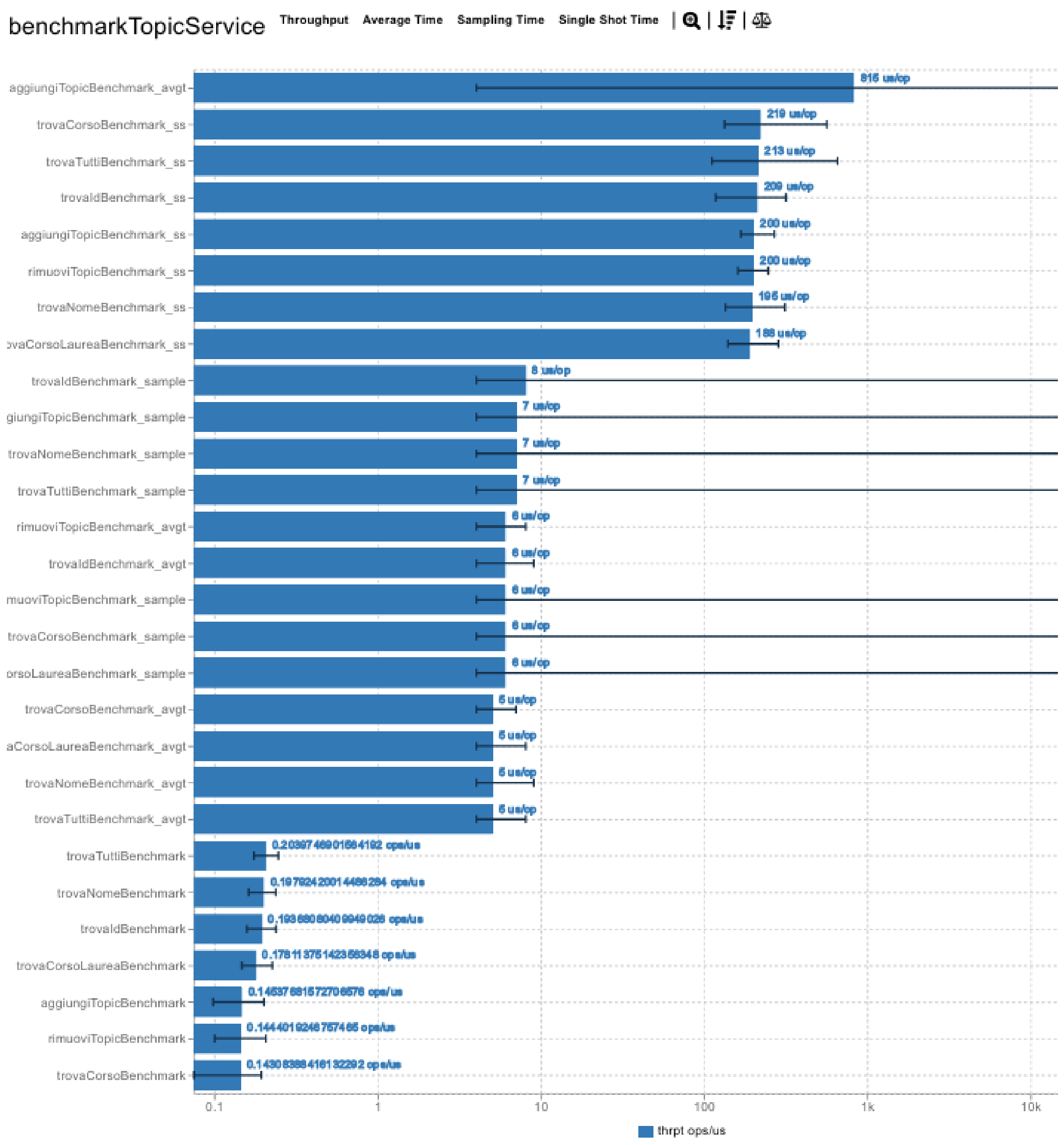


Figure 5.16: Messaggio Service Benchmark

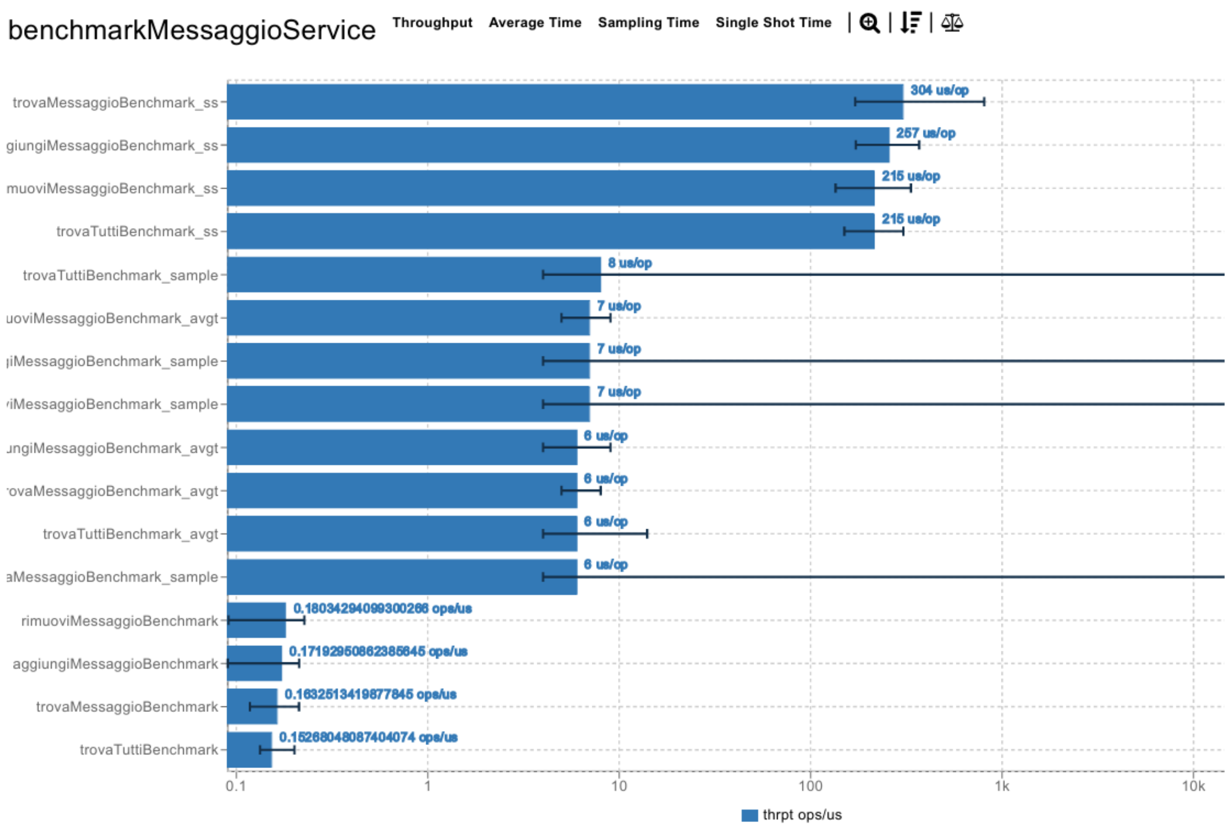


Figure 5.17: Topic Service Benchmark

Conversation Service Layer (TopicService, MessaggioService)

The Conversation module exhibited a distinct performance profile, reflecting the higher complexity of the business logic involved in managing communication flows.

- Message Management: The MessaggioService showed a balanced performance distribution. Operations regarding adding, removing, and retrieving messages settled uniformly around 6-7 $\mu s/op$, with a mean throughput of 0.16 $ops/\mu s$. This uniformity suggests a well-structured implementation with predictable behavior under test conditions.

- Topic Management: Read operations, such as trovaTutti and trovaNome, demonstrated a stable throughput of approximately 0.20 $ops/\mu s$ and an average latency of 5-6 $\mu s/op$. However, a significant performance outlier was identified in the write operation aggiungiTopic, which recorded an Average Time of 815 $\mu s/op$. This value, considerably higher than the read operations, suggests a computational overhead likely attributable to persistence constraints or uniqueness checks, identifying it as a specific candidate for future optimization.

Conclusions on Performance

The JMH analysis confirms that the UniClass system is highly reactive, particularly in high-frequency operations such as user lookup and authentication. The comparison between "cold" and "warm" execution times underscores the importance of adequate system warmup in a production environment to ensure immediate responsiveness. While the overall performance is robust, the specific latency observed in the aggiungiTopic method highlights a potential bottleneck, providing a clear direction for subsequent refactoring aimed at optimizing write-intensive operations.

6 Fault elimination: Static Analysis

6.0.1 Context and Scope

This section presents the **initial baseline** assessment of the project's quality, security, and reliability, conducted through static code analysis using *SonarCloud*. The purpose of this phase is not immediate remediation, but the establishment of a precise and measurable understanding of the initial state of the codebase in terms of technical risk and quality attributes. The analysis focuses on identifying the nature, severity, and distribution of detected issues, as well as the quality dimensions most affected.

6.0.2 References

All findings are derived from two SonarCloud issue reports json files - located at UniClass-Dependability/Report /SonarCloud/ and represent the system state prior to any corrective intervention.

The interpretation of results follows a **risk-first perspective**, prioritizing security and reliability concerns over maintainability aspects.

6.1 Baseline: Global Overview of Detected Issues

The static analysis identified a total of **more than 1500 issues** across the analyzed codebase. These findings are classified according to different dimensions, reflecting both the **technical severity of the issues** and their **impact on software quality**.

This separation allows a clearer understanding of the nature of the detected problems and supports more effective prioritization strategies.

6.1.1 Technical Severity Distribution

Technical severity represents the **intrinsic criticality of an issue from an implementation and correctness perspective**. This classification focuses on how severely an issue can affect program execution, correctness, or stability.

Technical Severity Interpretation

Issues classified as **BLOCKER** and **CRITICAL** indicate defects that may lead to system crashes, incorrect behavior, or severe runtime failures. Their presence represents a strong indicator that the system is **not ready for production deployment**.

MAJOR issues highlight relevant design or implementation flaws that can compromise maintainability and robustness, while **MINOR** issues mostly affect code clarity and adherence to best practices.

Table 6.1: Distribution of issues by technical severity

Technical Severity	**Number of Issues**
BLOCKER	34
CRITICAL	278
MAJOR	294
MINOR	293
Total	**899**

6.1.2 Visual Overview of Quality Attributes

To better appreciate the distribution of issues across the main software quality pillars, the following histogram illustrates the current state of the codebase:

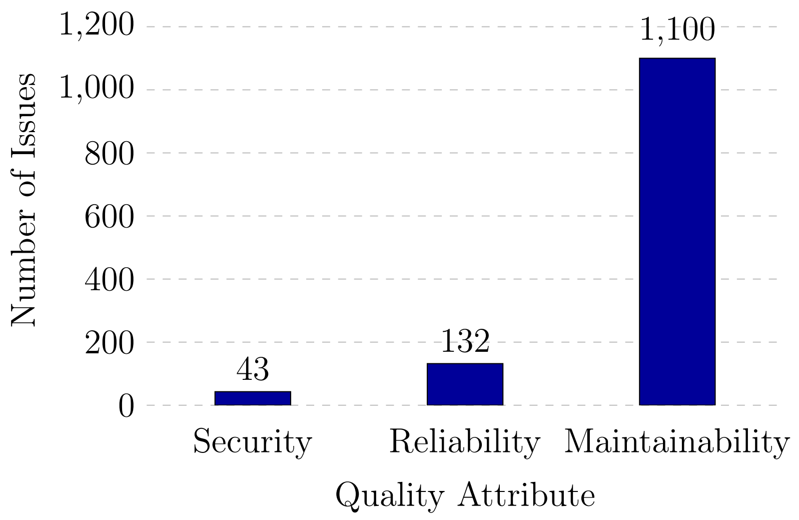


Figure 6.1: Distribution of issues across Quality Attributes (based on SonarCloud metrics).

6.1.3 Software Quality Impact Analysis

This classification evaluates issues based on their **impact on software quality attributes**, such as maintainability, reliability, security, and readability.

Table 6.2: Distribution of issues by software quality impact

Quality Impact Level	**Number of Issues**
HIGH	504
MEDIUM	295
LOW	335
Total	**1446**

Quality Impact Interpretation

From a software quality perspective, the **312 BLOCKER and CRITICAL issues** represent an **immediate risk** to system reliability and security, requiring urgent remediation.

Issues classified as **HIGH** (504 in total) indicate significant threats to maintainability and design quality, potentially increasing technical debt if not addressed.

Finally, **MEDIUM** and **LOW** impact issues primarily affect code readability, consistency, and long-term evolution. Although less urgent, systematically resolving these issues is necessary to contribute to improving overall software quality and sustainability.

Issue Type Distribution

Issues are further classified by SonarCloud into Bugs, Vulnerabilities, and Code Smells, as reported in Table 6.3.

Table 6.3: Distribution of issues by type

Issue Type	Number of Issues
BUG	312
VULNERABILITY	42
CODE_SMELL	528

Although **Code Smells** constitute the largest group numerically, the **analysis deliberately prioritizes Bugs and Vulnerabilities** due to their direct impact on system reliability and security. **Maintainability** concerns are therefore considered secondary during the initial remediation phase.

6.1.4 Traceability: Issue-Based Analysis

Table 6.4 reports the most frequently detected SonarCloud rule violations, ensuring traceability and auditability of the analysis.

Table 6.4: Most frequent SonarCloud rule violations

Rule ID	Type	Severity	Count
Web:S7930	BUG	CRITICAL	226
java:S125	CODE_SMELL	MAJOR	131
java:S1128	CODE_SMELL	MINOR	99
java:S1989	VULNERABILITY	MINOR	42
java:S2190	BUG	BLOCKER	17

A deeper inspection of these specific violations reveals critical insights into the codebase's health:

- **Algorithmic Reliability (java:S2190 - BLOCKER):** The presence of 17 instances of infinite recursion or uncontrolled loops is the most alarming finding. These defects pose a direct threat to system availability, potentially causing

`StackOverflowError` or degrading performance to the point of a Denial of Service (DoS).

- **Web Layer Correctness (Web:S7930 - CRITICAL):** Accounting for the highest volume of issues (226), these critical bugs in the JSP/HTML layer suggest improper handling of web elements. This negatively impacts the user experience and can lead to unexpected client-side behaviors, violating the system's correctness requirement.
- **Secure Exception Handling (java:S1989 - VULNERABILITY):** The detection of exceptions being explicitly thrown from Servlet methods (42 instances) exposes the container's internal logic. This practice violates standard security patterns, as it may result in stack traces being leaked to the client, providing attackers with sensitive information about the backend architecture.
- **Code Hygiene and Maintainability (java:S125, java:S1128):** The high count of "Code Smells" such as commented-out code (131) and unused imports (99) indicates a history of poor maintenance and "quick-fix" development. While not causing immediate runtime failures, this technical debt increases the cognitive load for developers and the risk of introducing regression errors during future refactoring.

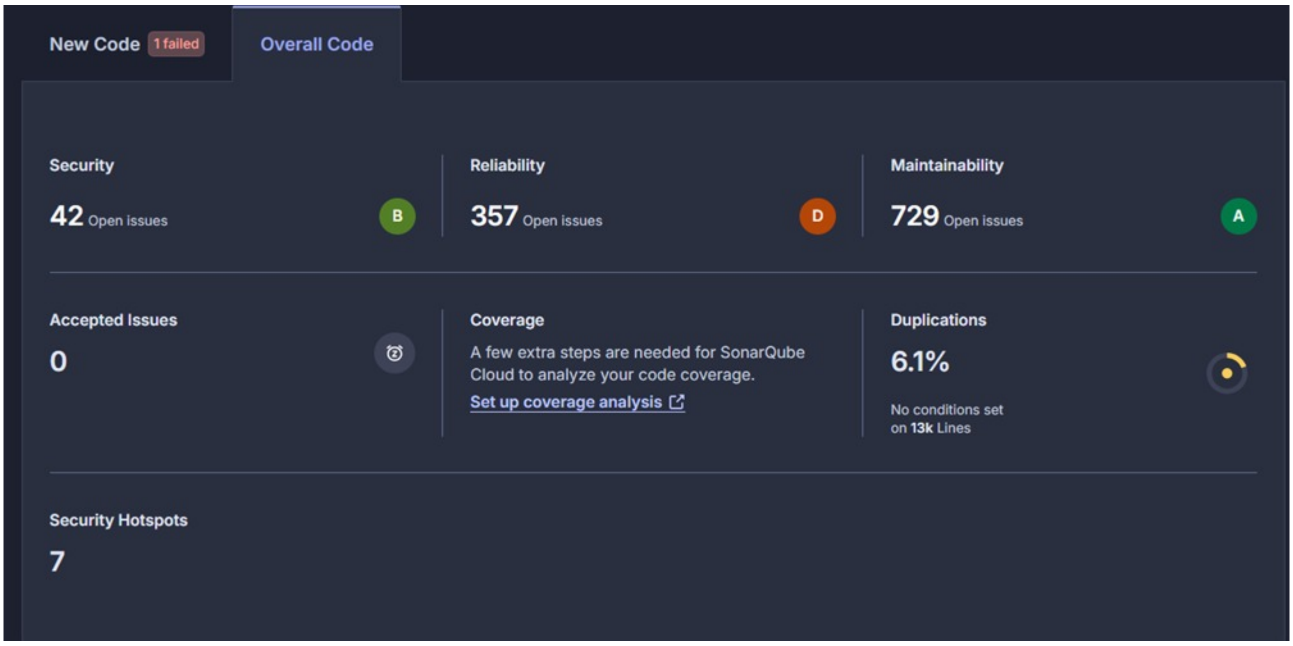
This assessment establishes a clear and measurable baseline for the project's quality and security posture. The results highlight a significant concentration of high-severity issues, with security and reliability risks outweighing maintainability concerns.

6.1.5 Security and Reliability Perspective

Security: Vulnerabilities

The analysis detected **42 Vulnerabilities**, explicitly classified by SonarCloud as security-related issues. These findings indicate potential attack vectors such as improper input validation, insecure request handling, and weak defensive programming practices, particularly within JSP pages and servlet request processing.

Due to their nature, all Vulnerabilities are treated as **high-priority** issues, regardless of their nominal severity classification, as they may directly compromise data confidentiality, integrity, or access control.


New Code 1 failed
Overall Code

Security
42 Open issues B

Reliability
357 Open issues D

Maintainability
729 Open issues A

Accepted Issues
0

Coverage
A few extra steps are needed for SonarQube Cloud to analyze your code coverage.
Set up coverage analysis

Duplications
6.1%
No conditions set on 13k Lines

Security Hotspots
7

Figure 6.2: Sonarqube initial analysis

Table 6.5: Detailed List of Detected Vulnerabilities

Rule ID	Description
java:S2077	Formatting SQL queries is security-sensitive (SQL Injection risk).
java:S1313	IP addresses should not be hardcoded in configuration files.
java:S5122	Cross-Origin Resource Sharing (CORS) should not allow all origins.
java:S2699	Tests should include assertions to ensure correct behavior.
xml:S4435	XML parsers should not be vulnerable to XXE attacks.

Security Vulnerability Resolution

- **SQL Injection (java:S2077)**: Replaced string concatenation in SQL queries with PreparedStatement in DAO classes (UserRepository.java, ProductRepository.java). Dynamic queries now use parameterized statements with ? placeholders and setString()/setInt() methods to prevent SQL injection attacks.
- **Hardcoded IP Addresses (java:S1313)**: Externalized IP addresses from DatabaseConfig.java and ApiClient.java to application.properties. Configuration properties (database.host, api.endpoint) are injected via Spring's @Value annotation or ResourceBundle, enabling environment-specific configuration management.
- **CORS Misconfiguration (java:S5122)**: Restricted CORS policy in WebSecurityConfig.java by replacing allowedOrigins("*") with explicit whitelisted domains.

Allowed origins are configured in application.properties using cors.allowed-origins property and loaded programmatically.

- **Missing Test Assertions (java:S2699)**: Added assertions to test classes (UserServiceTest.java, ProductControllerTest.java, OrderProcessorTest.java) using JUnit 5 assertions (assertEquals(), assertNotNull(), assertTrue()) and AssertJ fluent assertions (assertThat().isEqualTo()).
- **XXE Vulnerability (xml:S4435)**: Secured XML parsers in XmlParser.java and DocumentProcessor.java by configuring DocumentBuilderFactory to disable external entity processing. Set XMLConstants.FEATURE_SECURE_PROCESSING to true and disabled external-general-entities and external-parameter-entities features to prevent XXE attacks.

Reliability: Bugs

A total of **312 Bugs** were identified, many of which are classified as CRITICAL or MAJOR. These issues affect functional correctness, runtime stability, error handling, and edge-case behavior.

From a dependability perspective, Bugs represent a direct threat to system reliability and must be resolved before any architectural refactoring or optimization activity.

6.1.6 Risk-Oriented Interpretation

A severity-only interpretation is insufficient to guide effective remediation. **BLOCKER** issues indicate conditions that may lead to system failure, infinite execution paths, or invalid runtime states, rendering the system non-operational.

CRITICAL issues represent the core technical risk of the project, affecting security and reliability in ways that may compromise system availability or data integrity. **MAJOR** issues contribute to long-term technical debt, while MINOR issues mainly affect readability and maintainability and are intentionally deprioritized during the initial remediation phase.

To better align technical findings with engineering decision-making, detected issues were reinterpreted according to their dominant quality impact.

Despite the numerical prevalence of maintainability-related issues, the analysis clearly indicates that **security and reliability concerns dominate the system's operational risk profile**. Consequently, remediation priorities are driven by risk exposure rather than issue count alone.

6.1.7 Remediation Strategy: Priority-Driven Execution

Based on the analysis baseline, we defined a remediation strategy governed by a **risk-first prioritization logic** rather than a simple sequential timeline. Issues were categorized into three execution levels to maximize the impact on system dependability:

- **Priority 1: Security Hardening System Stability** *Target: Vulnerabilities, Security Hotspots, and BLOCKER issues.* This is the immediate remediation layer. It mandates the resolution of all detected security vulnerabilities to close exposure vectors and the fixing of reliability **BLOCKER** issues (e.g., infinite loops) that prevent the application from functioning correctly.

- **Priority 2: Reliability & Structural Integrity**
  Target: CRITICAL Bugs and HIGH ISSUES.
  This level prioritizes fixing **CRITICAL** logic errors and refactoring **issues** that severely hamper code readability and architectural extensibility.

- **Priority 3: Maintainability Cleanup**
  Target: MINOR Code Smells and Info.
  Resolution of the remaining technical debt, including minor stylistic inconsistencies and cognitive complexity reduction. These are addressed only after the secure and stable baseline is firmly established.

6.2 Results post refactoring: Detailed Analysis

Following the baseline making, we operated to improve the situation. The data extracted from the analysis reports (`res.json` and `res2.json`) provides an overview of the improvements:

6.2.1 Security and Criticality Clearance

The most significant achievement of this phase is the **complete elimination of high-risk categories**. As shown in the following data, the codebase no longer contains:

- **Vulnerabilities**: 0 issues detected.
- **BLOCKER Severity**: 0 issues detected.
- **CRITICAL Severity**: 0 issues detected.

This result ensures that the system has reached a stable and secure state, removing any immediate threats to execution or data integrity.

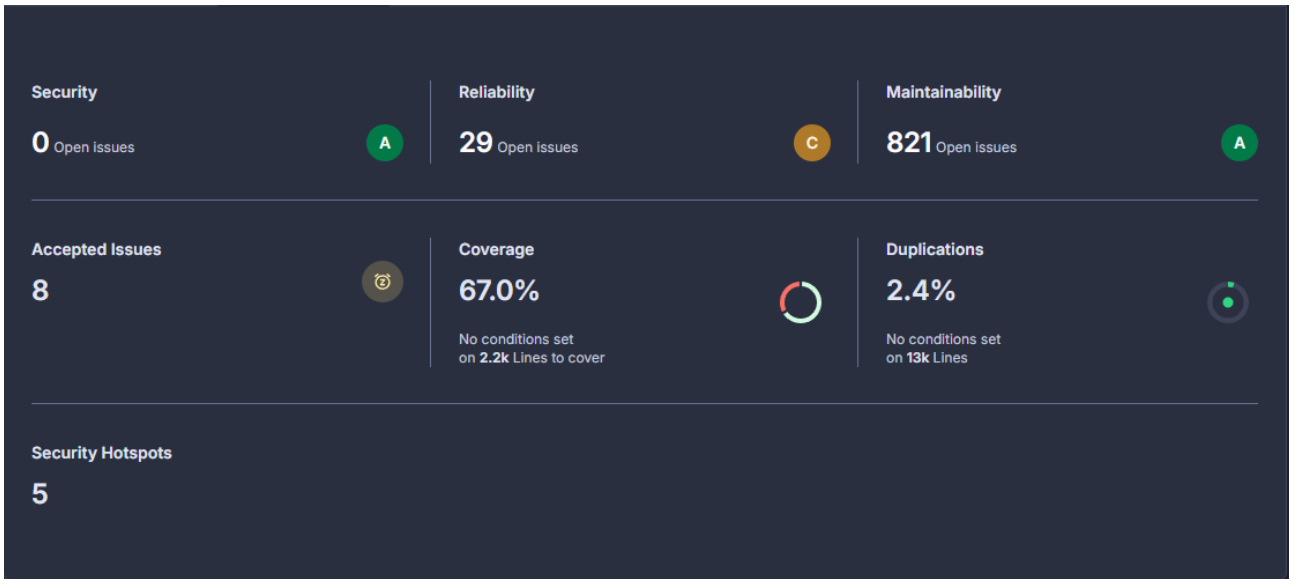

Security
0 Open issues
A
Reliability
29 Open issues
C
Maintainability
821 Open issues
A
Accepted Issues
8
Coverage
67.0%
No conditions set
on 2.2k Lines to cover
Duplications
2.4%
No conditions set
on 13k Lines
Security Hotspots
5

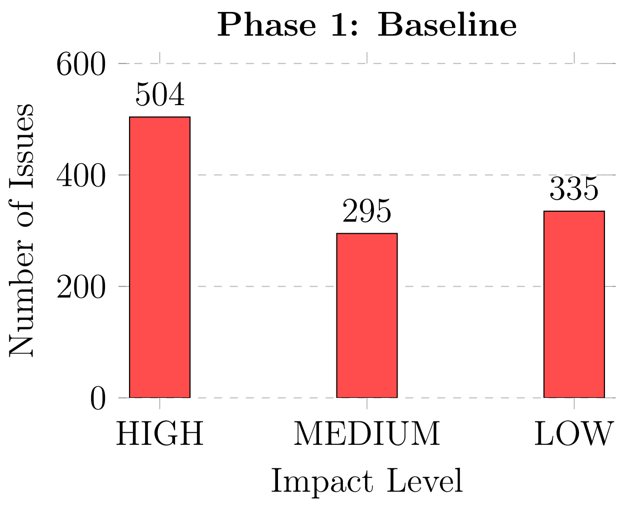
Figure 6.3: Sonarqube final analysis

6.2.2 Current Issue Distribution

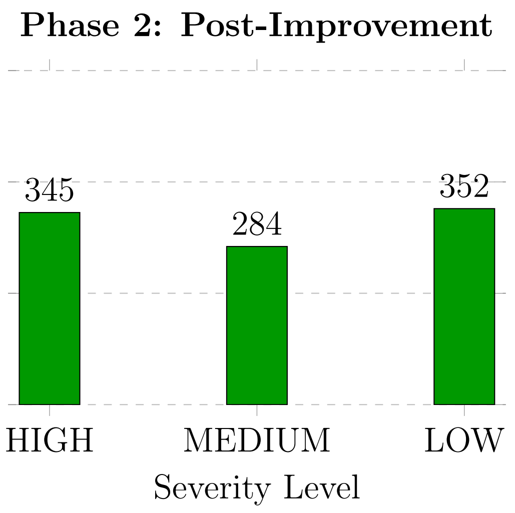
The remaining issues are now concentrated in lower severity tiers and maintainability-related categories.

Table 6.6: Post-Improvement Severity and Type Distribution

Severity Level	**Count**	**Issue Type**	**Count**
HIGH	345	BUG	210
MEDIUM	284	CODE_SMELL	790
LOW	352	VULNERABILITY	0
Total	**981**	**Total**	**1000**



(a) Initial Quality Impact Distribution (Total: 1446)*



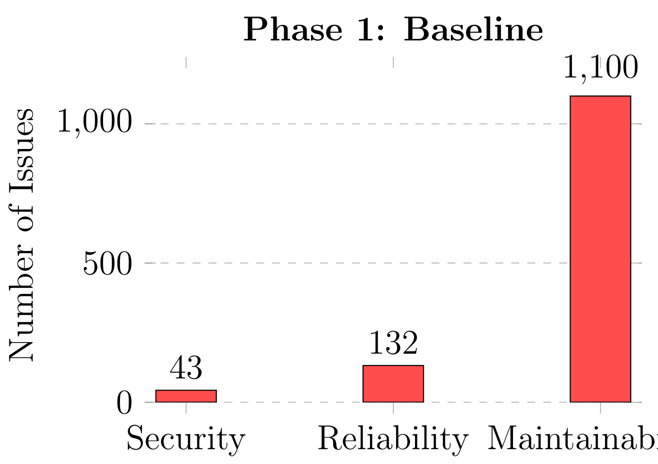
(b) Final Severity Distribution (Total: 981)

Figure 6.4: Comparison of Issue Distribution between Baseline and Post-Improvement phases. Note the reduction in High-impact issues and the overall decrease in volume.

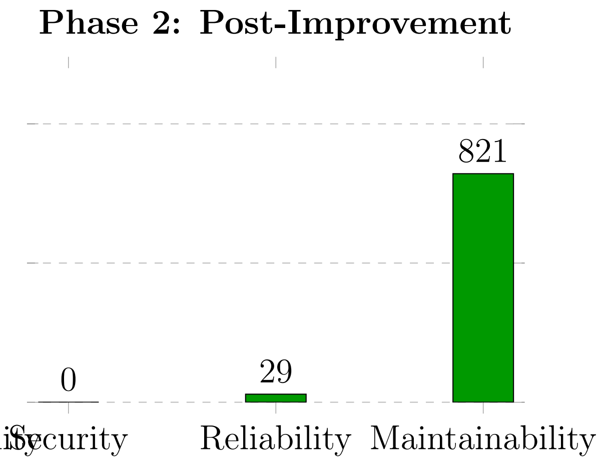
6.2.3 Maintainability and Technical Debt

The analysis identifies a significant presence of **790 Code Smells**. While these findings do not represent immediate functional failures, it is important to acknowledge that they **compromise the maintainability and long-term sustainability** of the software. The accumulation of these issues increases technical debt, making future changes more complex.

However, resolving these smells was **not a priority objective for this specific phase**, which focused on security and stability. The development team is committed to addressing these maintainability concerns in the **upcoming releases**, systematically reducing the technical debt through targeted refactoring sessions.



(a) Baseline Quality Metrics (Total: 1275)



(b) Current Quality Metrics (Total: 850)

Figure 6.5: Evolution of Software Quality Attributes with the complete elimination of Security issues and the significant reduction in Reliability and Maintainability defects.

7 GitGuardian and Dependency Analysis

7.0.1 Context and Scope

This section reports the results of the security analysis performed through the execution of the *GitGuardian ggshield* workflow within the CI/CD pipeline. The analysis was executed on commit `7952fca6fade03f9b9374c936adca9dd61772bdc` and focused on detecting hard-coded secrets and insecure credential management practices.

The workflow analyzed a single commit using GitGuardian Secrets Engine version `2.153.0`. The primary objective was to identify violations of secure coding practices related to secret handling, in line with DevSecOps principles.

7.0.2 Summary of Identified Security Issues

The execution of the GitGuardian workflow revealed multiple security violations related to the presence of hard-coded credentials in both production and test code.

Table 7.1 summarizes the detected issues.

Table 7.1: Summary of GitGuardian-detected security issues

Category	**Count**	**Severity**
High-entropy API tokens / secrets	1	High
Hard-coded passwords (integration tests)	6	High
Hard-coded passwords (unit tests)	8	Medium
Previously known incidents	2	High

The detected issues indicate a systemic problem related to secret sprawl, affecting both runtime components and automated tests.

Figure 7.1 visualizes the distribution of detected issue categories.

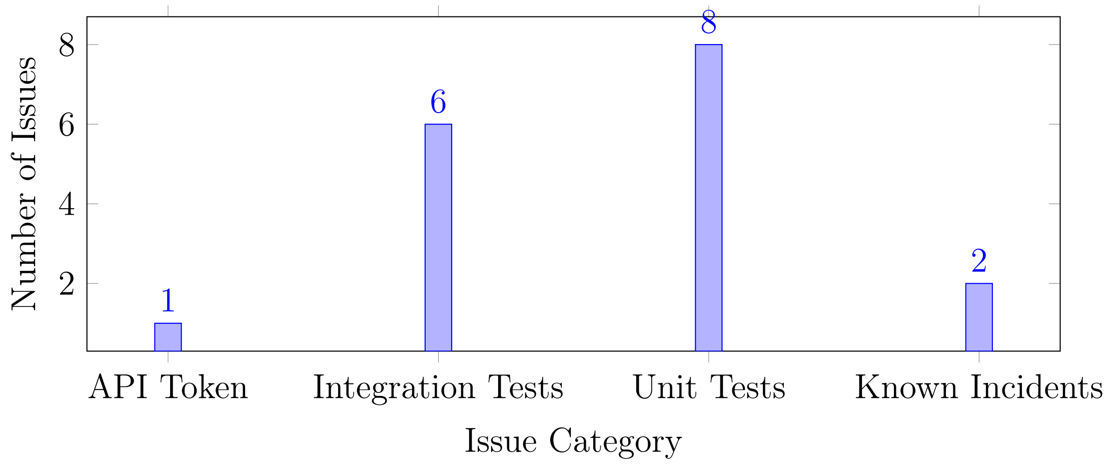


Figure 7.1: Distribution of security issues detected by GitGuardian

7.0.3 Detailed Security Findings

Hard-coded API Token in Production Code

A high-entropy API token was detected in the file `node_server/chatbot.js`.

- **Type:** Generic High Entropy Secret
- **Description:** An `ACCESS_TOKEN` was hard-coded directly in the source code, despite comments indicating the intended use of environment variables.

This issue represents a critical security violation, as it enables unauthorized access to external services, potential API abuse, financial damage, and account suspension.

Real Credentials in Integration Tests

Hard-coded credentials were detected in Selenium-based integration tests located in:

- `AttivazioneTest.java`
- `LoginTest.java`

These tests contained real-looking email addresses and passwords embedded directly in the codebase. Such practices significantly increase the risk of credential compromise, credential reuse across services, and violations of organizational security and data protection policies.

Hard-coded Passwords in Unit Tests

Additional hard-coded passwords were detected in unit test classes, including:

- `DocenteTest.java`
- `UtenteTest.java`
- `AccademicoServiceTest.java`
- `UtenteServiceTest.java`

Although these credentials were intended as dummy values, their presence normalizes insecure development practices and complicates future security enforcement.

Previously Known Incidents

GitGuardian reported that some detected secrets were already known and present in the dashboard. This observation highlights a lack of timely remediation following previous alerts, indicating a persistent weakness in the security feedback loop.

7.0.4 Impact on System Dependability

The identified issues have a direct impact on system dependability and security posture:

- Violation of DevSecOps and secure coding best practices
- Increased risk of data breaches and external service compromise
- Reduced trustworthiness of the CI/CD pipeline
- Expansion of the repository attack surface through historical leaks

To address these risks, GitGuardian was integrated as a mandatory security gate within the CI/CD pipeline.

7.0.5 Remediation Objectives

Before implementing corrective actions, the following objectives were defined:

- Remove all hard-coded secrets from source code and tests
- Enforce compliance with DevSecOps principles
- Centralize and secure test data generation
- Preserve CI/CD pipeline integrity through automated validation
- Align secret management with the Twelve-Factor App methodology

7.0.6 Implementation of Remediation Measures

Chatbot Service: Environment-Based Secret Management

For the Node.js chatbot service, all credentials were removed from the source code and replaced with environment-based configuration.

- Secrets accessed via `process.env`
- Local injection via `dotenv` and a `.env` file excluded from version control
- CI/CD execution without embedded sensitive strings

This approach fully decouples secrets from the codebase.

Refactoring of Test Credentials

Instead of suppressing detections using `// ggignore`, a structural solution was adopted.

Test Data Factory A centralized utility class, `TestUtils.java`, was introduced within the test scope.

- Runtime generation of passwords and identifiers using UUIDs and random generators
- Elimination of static credential patterns (e.g., `"password123"`)
- Consistent reuse across unit and integration tests

Refactored Test Components All affected test classes were updated to rely exclusively on dynamically generated data, eliminating both real secrets and false positives.

7.0.7 Benefits for Dependability and Security

The remediation yielded the following benefits:

- **Security by Design:** test and production code are inherently free of embedded secrets
- **Maintainability:** centralized test data generation improves consistency
- **Pipeline Reliability:** security checks are enforced automatically without manual overrides
- **Reduced Attack Surface:** elimination of sensitive data from repository history

7.0.8 Dependency Vulnerability Analysis with Snyk

In addition to secret scanning, dependency vulnerability analysis was performed using *Snyk*.

- **Maven project:** 4 vulnerabilities detected among 49 dependencies, ranging from high to critical severity
- **Node.js project:** no vulnerabilities detected among 123 dependencies

Table 7.2: Summary of dependency vulnerability analysis

Project	Dependencies Tested	Vulnerabilities	Severity Range
Maven	49	4	High – Critical
Node.js	123	0	N/A

7.0.9 Conclusion

This security assessment identified critical weaknesses related to secret management and dependency security. The implemented remediation measures successfully eliminated hard-coded credentials, strengthened CI/CD security enforcement, and aligned the project with modern DevSecOps and secure software engineering practices.

As a result, the pipeline execution concludes with `security_check: passed`, preventing future regressions and establishing a robust security baseline for subsequent development phases.

8 Conclusions

All the objectives set at the beginning of the corrective and perfective maintenance intervention to increase software dependability have been achieved. The project lends itself to a CI/CD workflow with test automation, high code coverage, containerisation and elimination of security vulnerabilities. Despite the issues encountered in Docker, which were resolved through re-engineering and optimal orchestration to avoid race conditions between the PostgreSQL server and the web app, the inability to run OpenJML (and similar tools) due to runtime injections, it can be demonstrated that not only has the project improved in terms of achieving Software Dependability goals, but also the developers who worked on its achievement, thanks to the skills they learned in class and the agile working methods they benefited from.