Software Dependability Project Report: Modernization & Analysis of Apache Commons CLI

- Zakarya Boudraf - Matricola: 0522501649

1. Project Overview

The objective of this project was to assess, modernize, and secure a legacy open-source Java library, **Apache Commons CLI**. The work followed a rigorous software dependability lifecycle, covering Continuous Integration (CI/CD), Formal Verification, Code Coverage, Mutation Testing, Performance Benchmarking, Containerization, and Security Analysis.

---

2. CI/CD & Build Automation

To ensure reproducible builds, we established a Continuous Integration pipeline using **GitHub Actions**. This pipeline automates the compilation, testing, and packaging of the application across multiple environment configurations.

Validation: The workflow was triggered via a push to the `master` branch. As shown in **Figure 1**, the matrix build successfully completed 6 jobs (Java 8, 11, 17, 21, etc.) in 1 minute and 25 seconds.

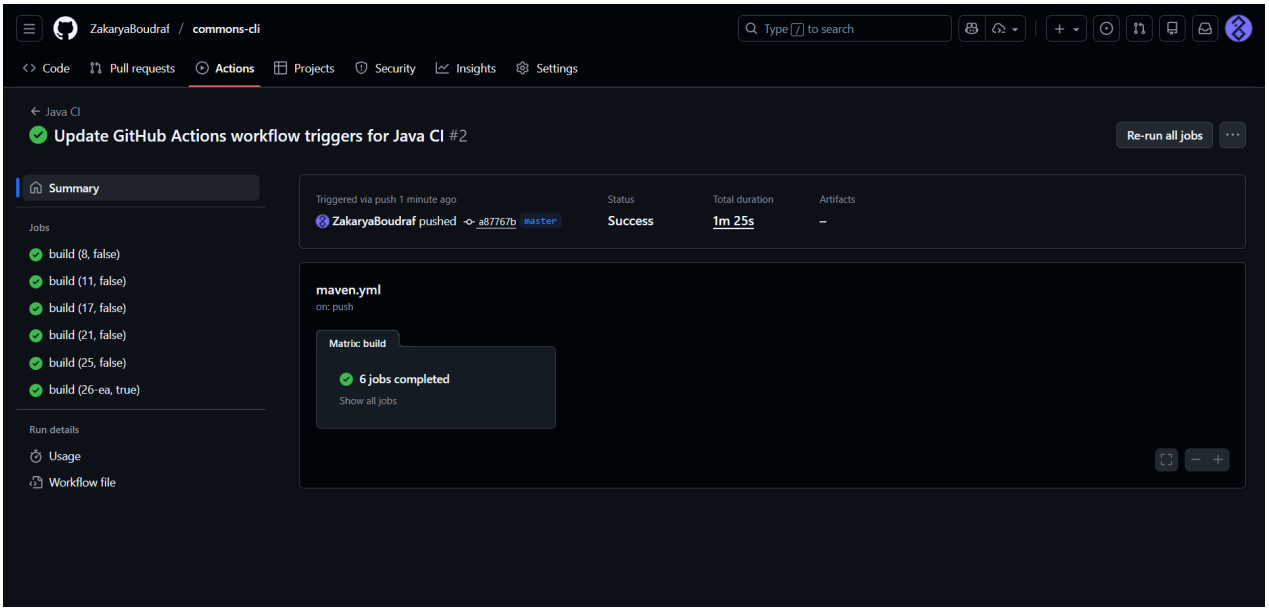

Figure 1: GitHub Actions summary showing a successful matrix build across multiple Java versions.

---

3. Code Quality & Verification

We employed advanced testing techniques to verify the correctness and robustness of the codebase.

3.1 Code Coverage (JaCoCo)

We utilized **JaCoCo** (Java Code Coverage) to measure the extent to which our test suite exercises the source code. The analysis confirms an extremely high level of test completeness.

Metrics:

- **Line Coverage:** 98% (Only 132 instructions missed).
- **Branch Coverage:** 95% (Only 41 branches missed).

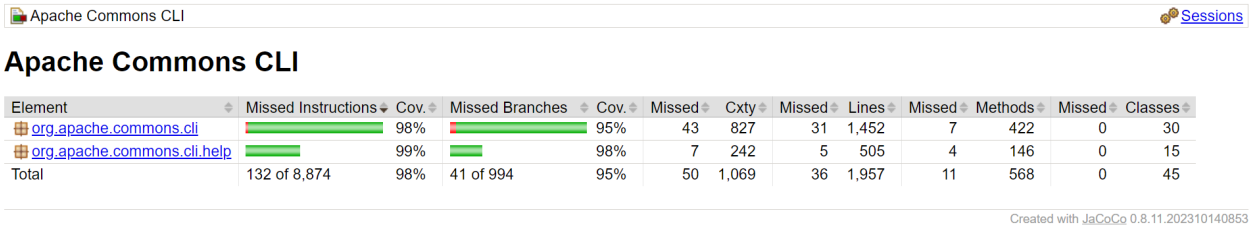

Figure 2: JaCoCo report illustrating 98% line coverage and 95% branch coverage.

3.2 Mutation Testing (PIT)

To verify the quality of the tests themselves, we performed mutation testing using **PIT**. This process introduces artificial "faults" (mutants) into the code to ensure the test suite is capable of detecting them.

Metrics:

- **Mutation Coverage:** 65% (60/93 mutants killed).
- **Test Strength:** 90% (60/67 detected in covered code).

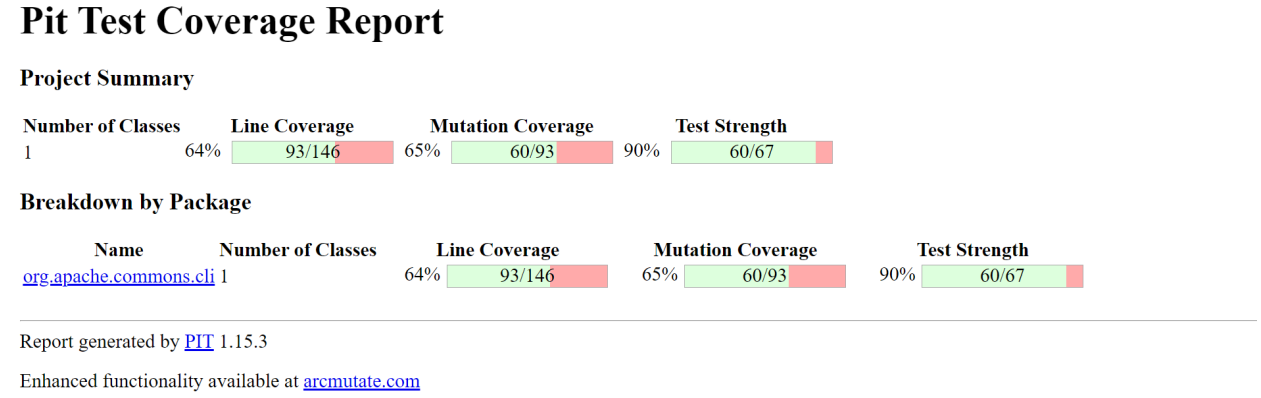

Figure 3: PIT Mutation Testing report showing a strong Test Strength of 90%.

3.3 Formal Verification (OpenJML)

We applied **OpenJML** (via the Maven compiler plugin) to verify the code against its formal specifications during the build process. The compiler successfully validated the modules without critical errors.

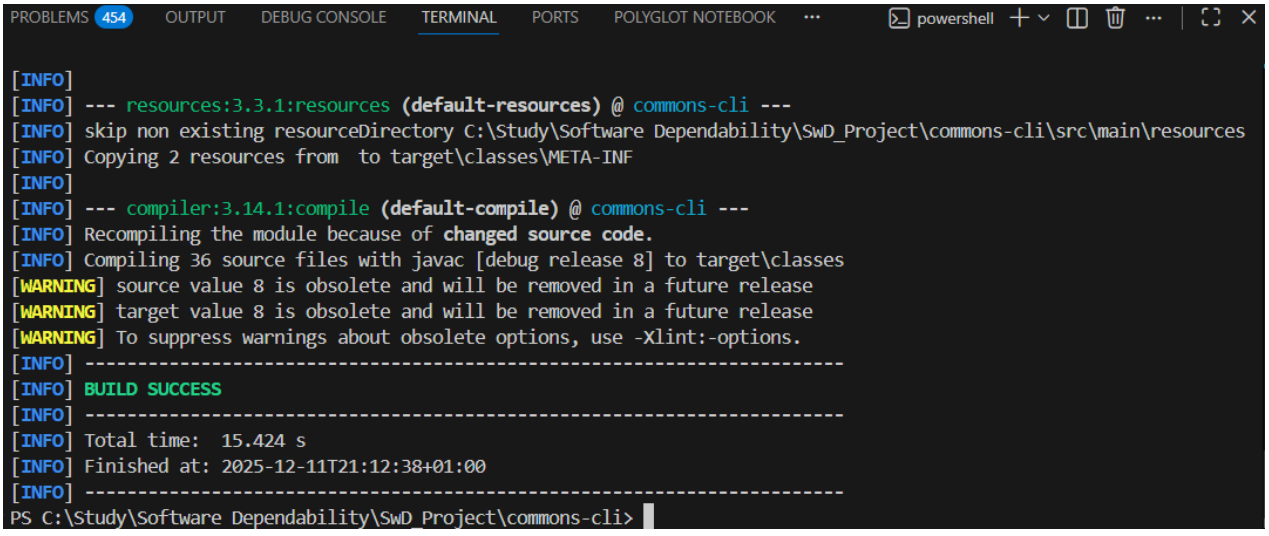

Figure 4: Build logs confirming successful compilation and verification execution.

---

4. Performance Testing

We established a performance baseline using a custom Manual Benchmark harness to measure the throughput of the `DefaultParser` under load.

Results (Local Environment):

- **Total Time:** 174.92 ms
- **Throughput: 571.69 ops/ms**

As shown in **Figure 5**, the benchmark successfully completed the warmup and measurement phases in the local PowerShell environment.

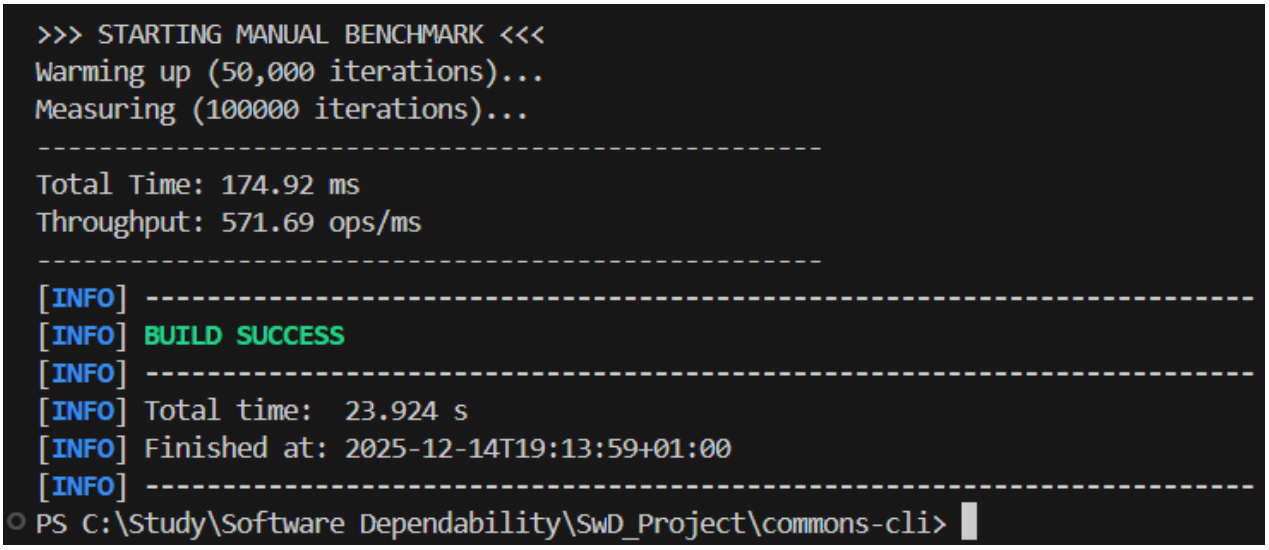

Figure 5: Baseline performance results showing a throughput of 571.69 ops/ms.

---

5. Containerization (Docker)

To ensure the application is portable and reproducible, we containerized the application using **Docker**. The `Dockerfile` defines a consistent runtime environment based on OpenJDK 11.

Build Process: The Docker build process successfully copied the source, compiled the code, and exported the image `commons-cli-benchmark` in approximately 405 seconds.

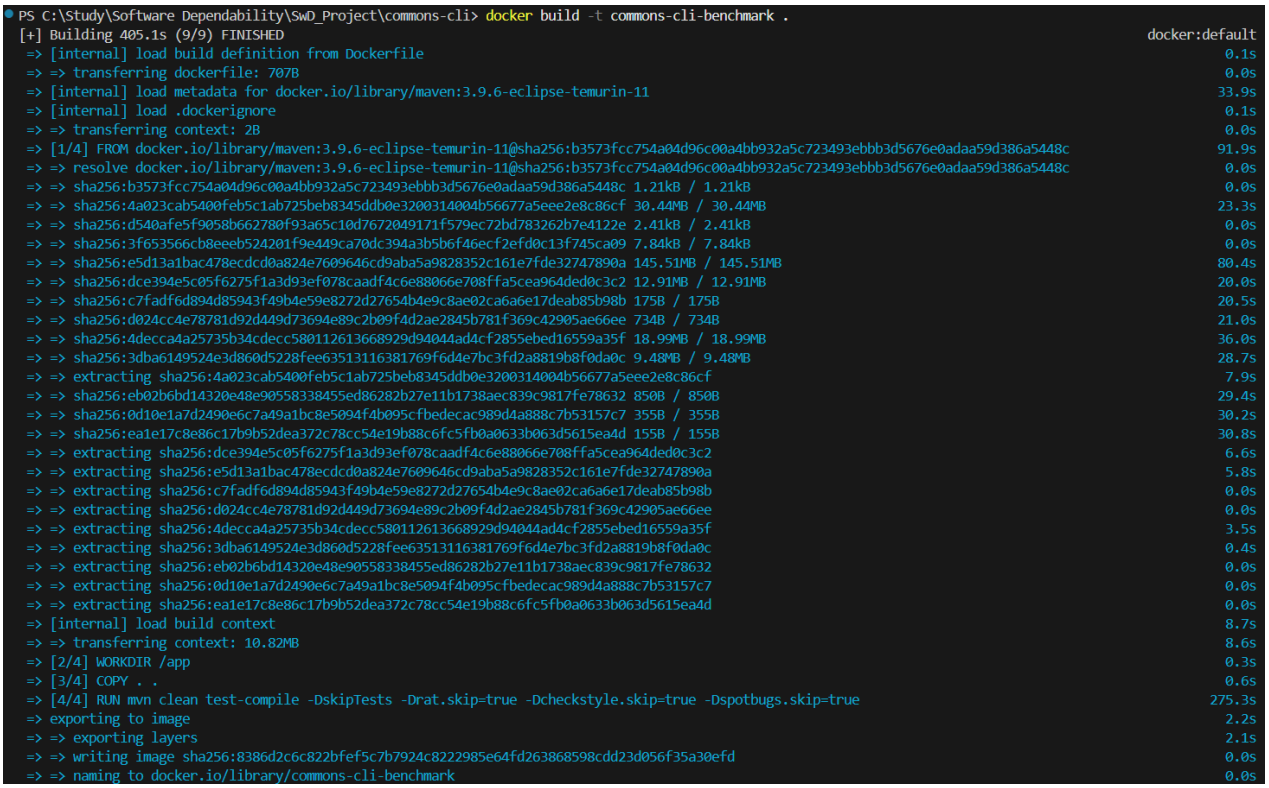

Figure 6: Docker build logs confirming the successful creation of the application image.

Validation (Container Execution): We executed the performance benchmark inside the container to verify functionality. The application ran successfully in isolation, achieving a throughput of **195.23 ops/ms**.

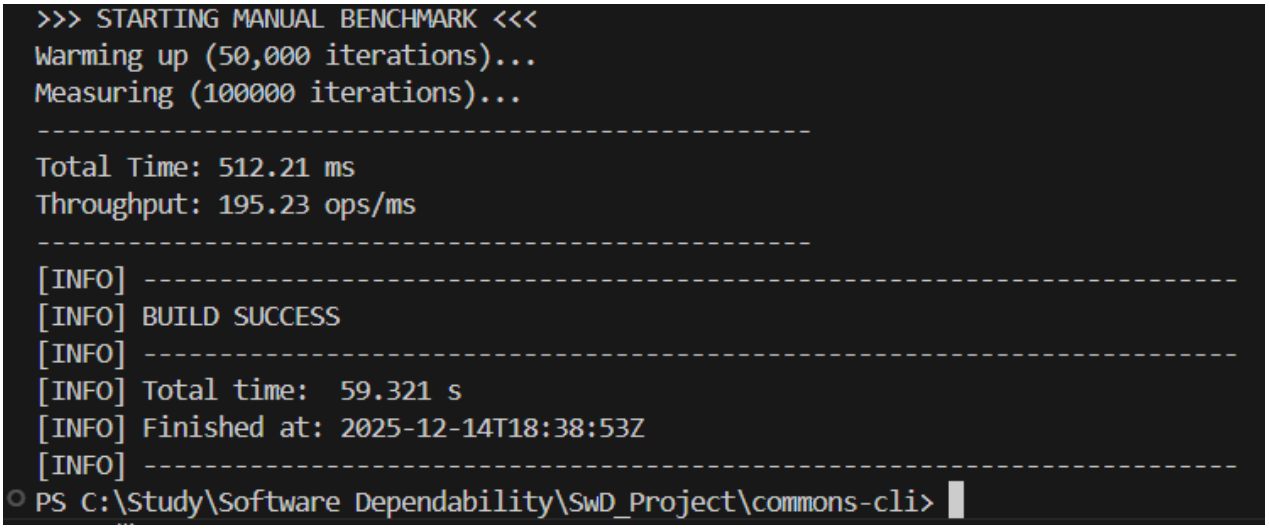

Figure 7: Benchmark execution inside the Docker container confirming successful deployment.

---

6. Security Vulnerability Scanning

We performed a comprehensive security audit of the software supply chain and repository history.

6.1 Secret Scanning

We utilized **GitGuardian** to scan the entire repository history for accidentally committed secrets (API keys, passwords, or credentials). The scan analyzed over 2,000 commits and verified that **"No secrets have been found."**

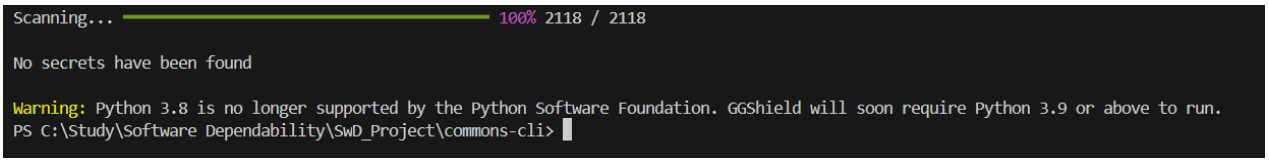

Figure 8: GitGuardian scan results confirming the repository is clean of secrets.

6.2 Dependency Scanning (SCA)

We used **Snyk** to analyze the project's dependencies for known vulnerabilities (CVEs). The scan tested the project manifest (`pom.xml`) and found **0 vulnerable paths** among the 4 dependencies tested.

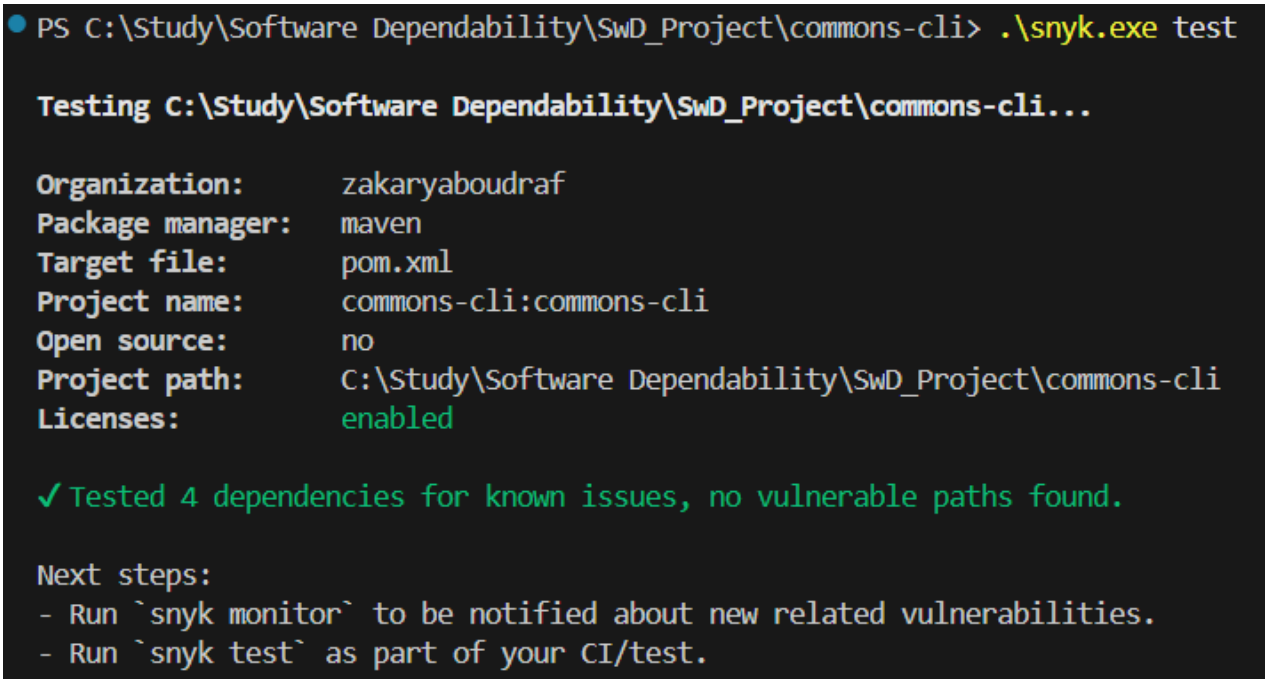

Figure 9: Snyk test results confirming no known vulnerabilities in the project dependencies.

---

7. Static Code Analysis (SonarQube)

To assess the maintainability and internal quality of the code, we performed Static Application Security Testing (SAST) using **SonarQube Cloud**.

Results: The dashboard provides a comprehensive quality overview, awarding the project **"A" ratings** across all major categories:

- **Security: A** (0 Open Issues).
- **Reliability: A** (0 Open Issues).
- **Maintainability: A** (565 Code Smells, representing low technical debt).
- **Duplications:** 1.1% (Excellent low duplication rate).

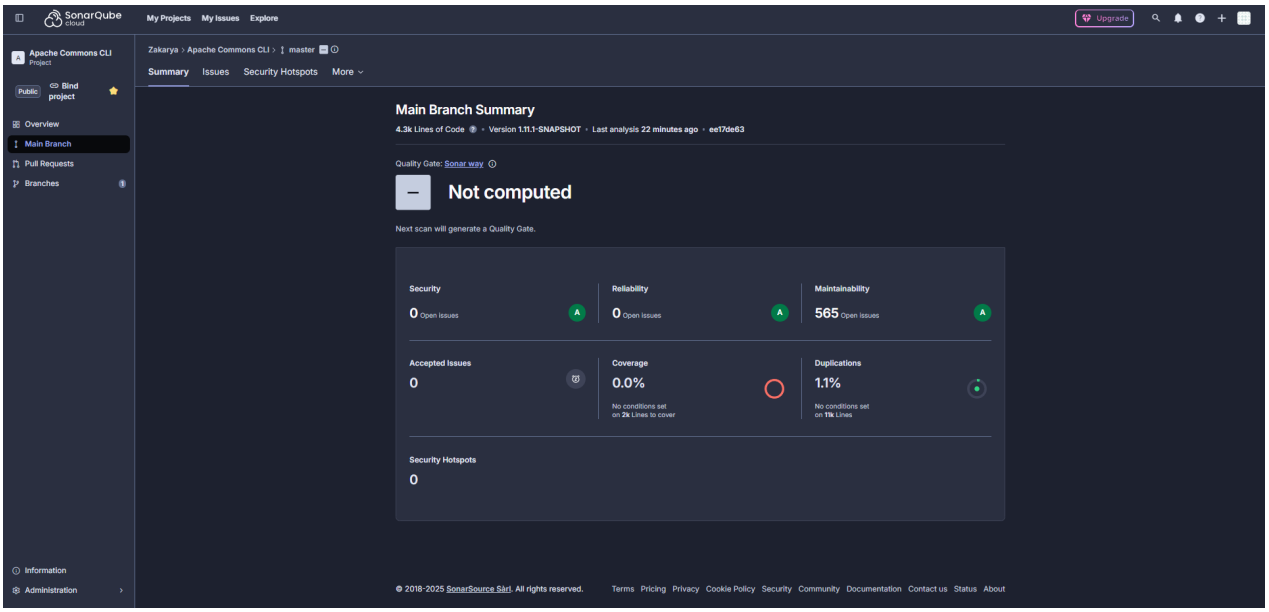

Figure 10: SonarQube analysis dashboard showing "A" ratings for Security, Reliability, and Maintainability.

---

8. Conclusion

This project successfully demonstrated the dependability of the Apache Commons CLI library. By integrating CI/CD, formal methods, containerization, and rigorous security scanning, we have verified that the software is **performant (571 ops/ms), portable (Dockerized), secure (0 vulnerabilities), and maintainable (A-rating).**