

# QAOps-Based Quality Automation & Performance Management

QAOps Implementation & Results

**VIRNECT QA Team**

**Sungtae Kim**

# I. Project Overview

## Purpose

- Establish a test automation pipeline and standardize code quality measurement to systematically ensure overall project quality.
- Enhance collaboration efficiency between development and QA teams by integrating Agile and CI/CD processes.
- Centralize multi-stack environments (Frontend, Backend, Unity C#) into a single SonarQube dashboard for unified quality visualization.
- Strengthen release stability by implementing a continuous regression testing framework and fostering a team-wide quality culture.
- Conduct Postman-based API functional testing to validate backend API behavior and data integrity.
- Perform JMeter-based performance testing and OWASP ZAP-based security testing to proactively verify not only functional quality but also non-functional quality aspects such as performance and security.

## Goals

1. Build a Static Analysis Environment for Automated Test Code
  - Integrate SonarQube into the Jenkins pipeline to identify code smells, bugs, and security vulnerabilities at an early stage.
2. Secure and Visualize Test Coverage
  - Use tools such as Jest, JaCoCo, and Istanbul to measure Statement, Branch, and Line coverage with automated reporting.
3. Enforce CI/CD Quality Gates
  - Apply merge-blocking policies on pull requests when quality gate conditions are not met to mitigate risks in advance.
4. Strengthen Development-QA Collaboration
  - Share test and quality reports in real time through Jenkins, Slack, and GitHub to facilitate Agile team collaboration.
5. Reduce Code Duplication and Promote Modular Refactoring
  - Lower duplication rates and introduce common modules to reduce maintenance costs and improve development productivity.
6. Implement Unified Quality Management Across Multi-Stack Projects
  - Standardize overall project quality metrics by managing Java, TypeScript, and C# codebases on a single platform.

# I. Project Overview

## Goals

### 7. Establish a Postman-Based API Testing Framework

- Validate request/response behavior, status codes, and data consistency through unit-level and scenario-based API tests.
- Improve regression stability by automating repeated test executions and maintaining structured test reports.

### 8. Establish a JMeter-Based Performance Testing Framework

- Execute load and concurrency tests on login flows, core APIs, and critical user scenarios to verify TPS, response time, and system stability.
- Manage performance test results as quantitative metrics to define clear release criteria.

### 9. Conduct OWASP ZAP-Based Security Testing

- Perform vulnerability scans on web and API endpoints to proactively identify security risks based on the OWASP Top 10.
- Document security test results to systematically track remediation actions and maintain security test history.

# I. Project Overview

## Schedule

No.	Schedule	Category	Key Tasks	Responsible Person		Remarks
1	Monday, April 14, 2025	Configurati on & Integration	<ul style="list-style-type: none"> <li>Configure advanced quality gates and rules per tech stack (Java, TS, C#).</li> <li>Implement branch coverage thresholds in CI.</li> <li>Automate Slack and GitHub PR notifications for quality gate results.</li> </ul>	VIRNECT Co., Ltd.   QA Team	Kim Sung-tae, Senior Engineer	
2	Tuesday, May 13, 2025	Test Coverage Expansion	<ul style="list-style-type: none"> <li>Expand unit test coverage with Jest, JUnit, and Istanbul.</li> <li>Identify and address coverage blind spots.</li> <li>Create coverage heatmap dashboards in SonarQube.</li> </ul>	VIRNECT Co., Ltd.   QA Team	Kim Sung-tae, Senior Engineer	Target coverage ≥80% (backend)
3	Monday, June 16, 2025	Automation & Optimizatio n	<ul style="list-style-type: none"> <li>Optimize Jenkins pipeline parallelization for faster builds.</li> <li>Integrate caching mechanisms for test runs.</li> <li>Refactor repetitive modules to reduce duplication.</li> </ul>	VIRNECT Co., Ltd.   QA Team	Kim Sung-tae, Senior Engineer	
4	Monday, July 14, 2025	Security & Vulnerabilit y	<ul style="list-style-type: none"> <li>Enable SonarQube security hotspot analysis.</li> <li>Conduct vulnerability scanning and remediation.</li> <li>Establish monthly security quality gate reviews.</li> </ul>	VIRNECT Co., Ltd.   QA Team	Kim Sung-tae, Senior Engineer	
5	Monday, August 11, 2025	Stabilization & Reporting	<ul style="list-style-type: none"> <li>Finalize dashboards for CI/CD &amp; QA metrics.</li> <li>Automate quarterly test coverage &amp; quality trend reports.</li> <li>Conduct knowledge-sharing sessions across teams.</li> </ul>	VIRNECT Co., Ltd.   QA Team	Kim Sung-tae, Senior Engineer	Quarterly summary plann ed

# I. Project Overview

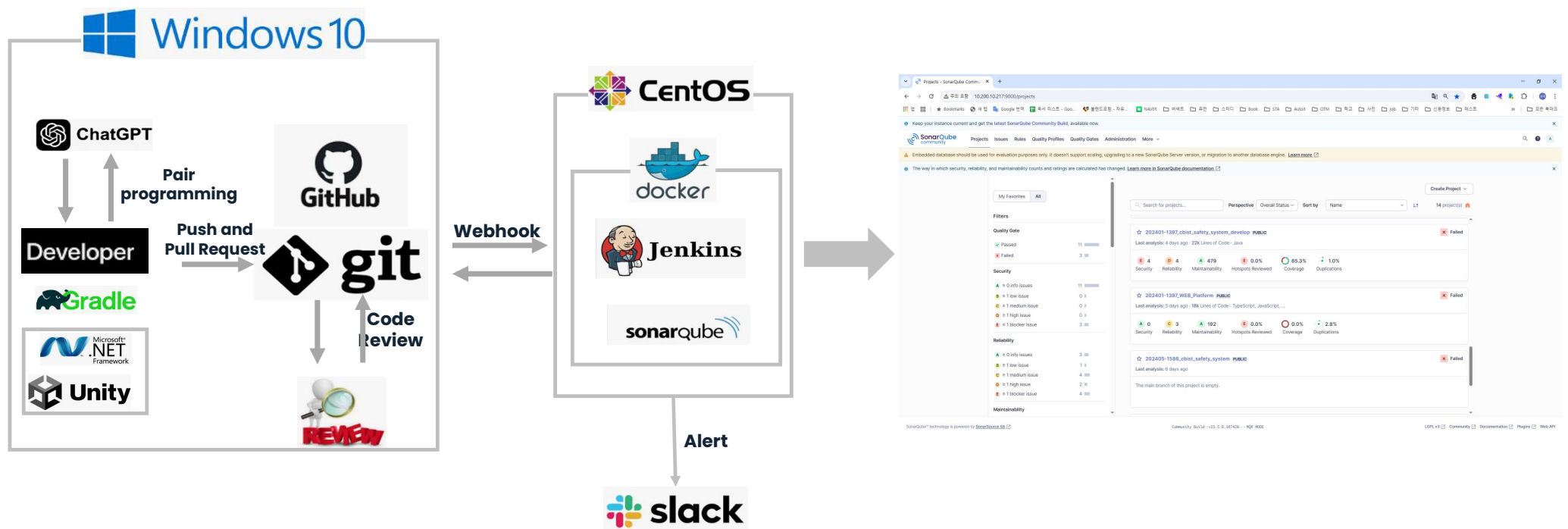
## Schedule

No.	Schedule	Category	Key Tasks	Responsible Person		Remarks
6	August 26, 2025 (Tuesday)	Performance Test Planning	<ul style="list-style-type: none"> <li>Design JMeter-based performance test scenarios.</li> <li>Establish load and concurrency test plans for key APIs and critical user flows.</li> <li>Define performance metrics such as TPS, response time, and error rate.</li> </ul>	VIRNECT Co., Ltd.   QA Team	Kim Sung-tae, Senior Engineer	
7	August 29, 2025 (Friday)	Performance Testing (Internal)	<ul style="list-style-type: none"> <li>Execute internal performance tests using JMeter.</li> <li>Analyze test results and identify system bottlenecks.</li> <li>Document performance test results and derive improvement actions.</li> </ul>	VIRNECT Co., Ltd.   QA Team	Kim Sung-tae, Senior Engineer	
8	December 1, 2025 (Monday)	Final Performance Testing	<ul style="list-style-type: none"> <li>Execute final performance tests on the system with applied improvements.</li> <li>Verify whether target performance metrics (TPS, response time, stability) are satisfied.</li> <li>Prepare the final performance test report and assess release readiness.</li> </ul>	VIRNECT Co., Ltd.   QA Team	Kim Sung-tae, Senior Engineer	
9						
10						



# I. Project Overview

Building an Integrated Quality Management Pipeline (Java + TS + C#)



◀Manage all stacks on a single SonarQube dashboard>

# I. Project Overview

## 1. SonarQube Static Analysis Status – Frontend (TypeScript)

### • Language / Environment

- TypeScript (Web Frontend)

### • Analysis Metrics

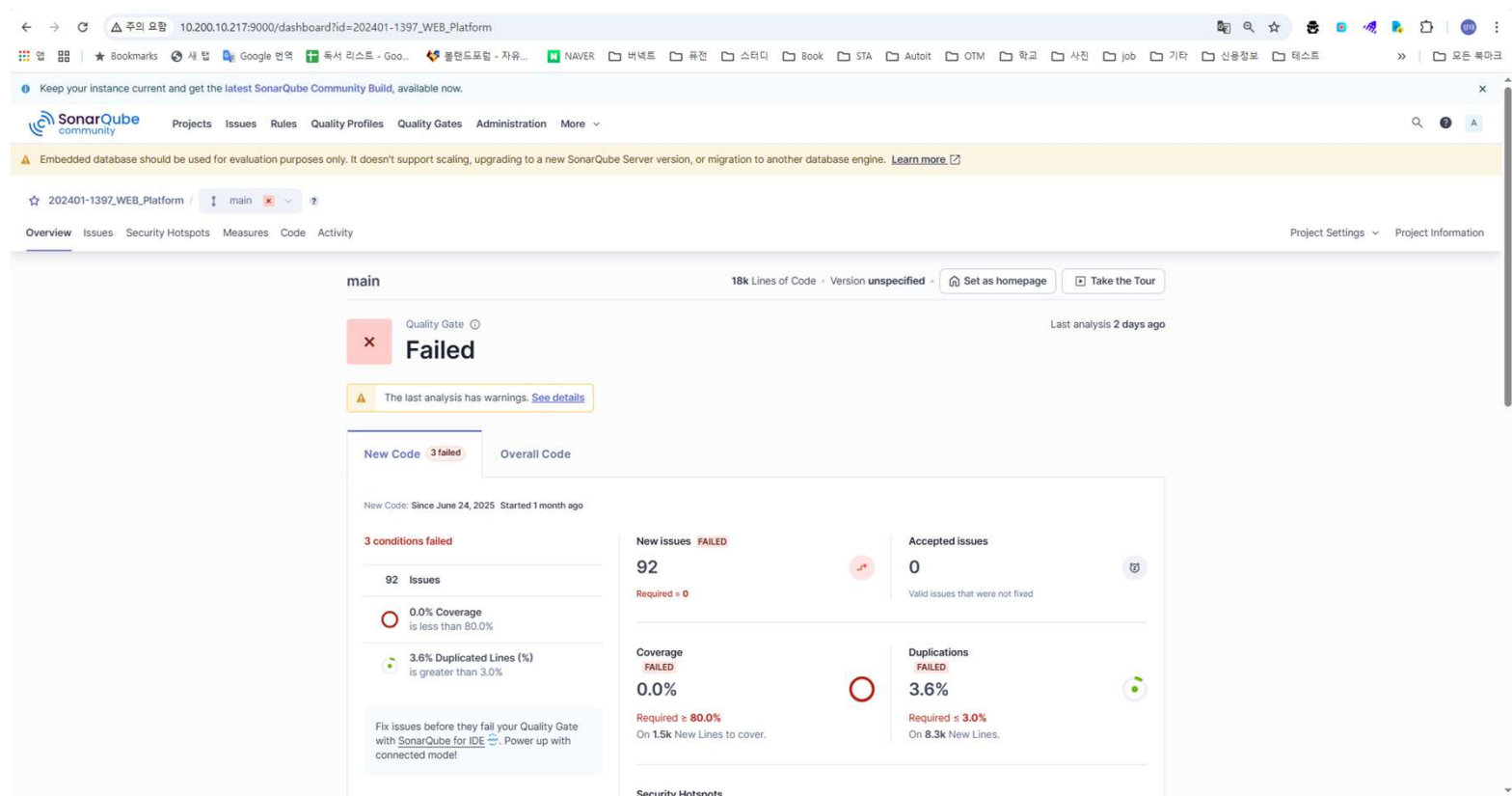
- Coverage: 0.0% (Needs collection/integration)
- New issues: 92
- Duplication: 3.6% (Exceeds threshold of  $\leq 3.0\%$ )
- Quality Gate: X Failed (3 conditions failed)
- Last analysis: 2 days ago

### • Key Points

- Current coverage not collected  
→ Plan to integrate with Jest lcov.
- Code duplication and new issues are the main reasons for Quality Gate failure.

### • Role

- Build SonarQube environment & integrate with CI pipeline.
- Configure language-specific rules and Quality Gate settings.
- Visualize and manage coverage, issues, and code duplication.



# I. Project Overview

## 2. SonarQube Static Analysis Status – Backend (Java)

### • Language / Environment

- Java (Spring Boot)
- Gradle + JUnit + JaCoCo

### • Analysis Metrics

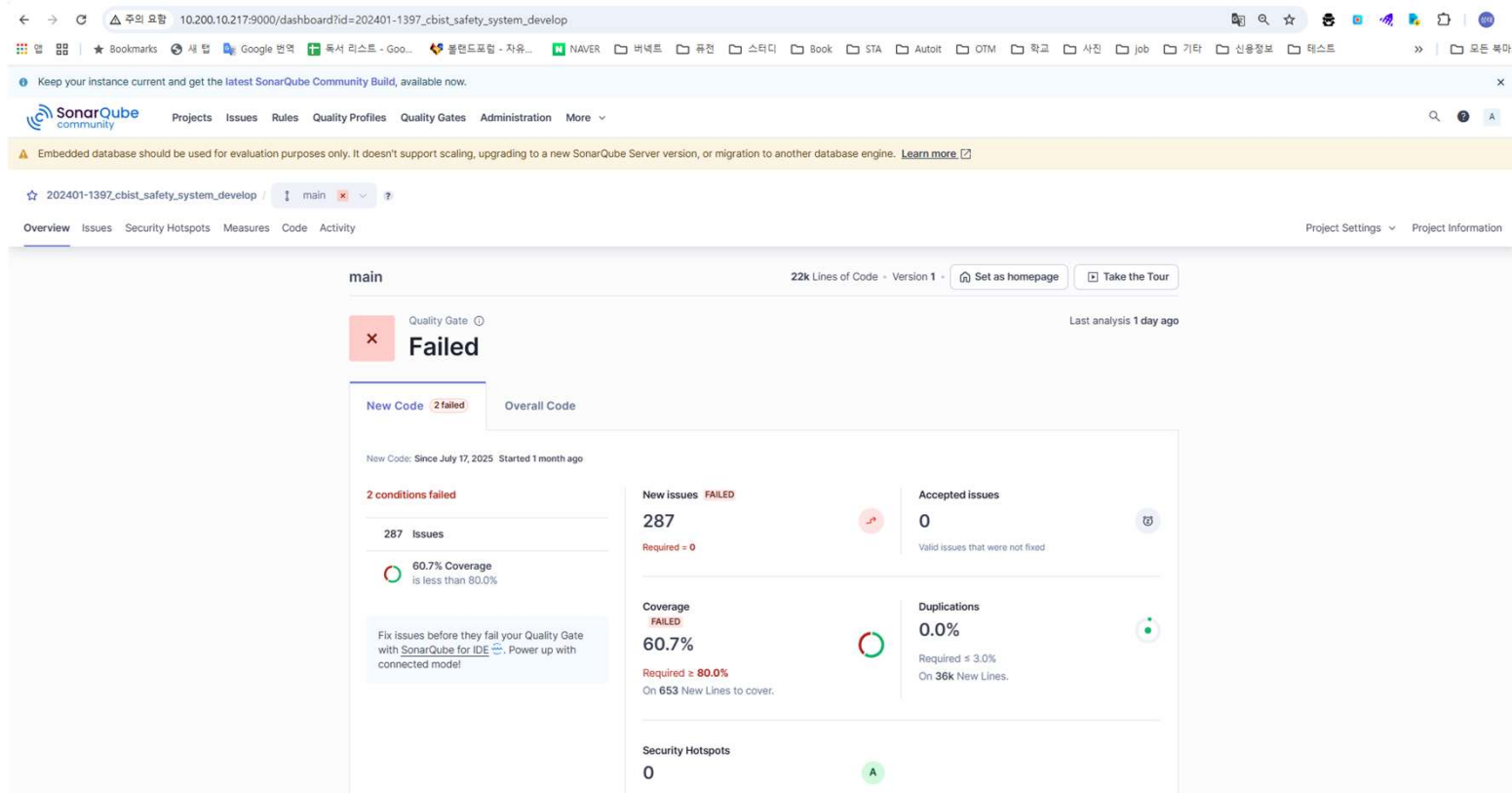
- Coverage: 60.7% (Target > 80%)
- New issues: 287
- Duplication: 0.0%
- Quality Gate: X Failed (2 conditions failed)
- Last analysis: 1 day ago

### • Key Points

- Coverage collection enabled through JaCoCo integration.
- Coverage has been collected but does not meet the target.
- A large number of new issues are the main reason for Quality Gate failure.

### • Role

- Build SonarQube environment & integrate with CI pipeline.
- Configure quality rules and Quality Gate settings per language.
- Visualize and manage coverage, issues, and code duplication.





# I. Project Overview

## 3. SonarQube Static Analysis Status – Unity (C#)

### • Language / Environment

- C# (Unity WebGL)

### • Analysis Metrics

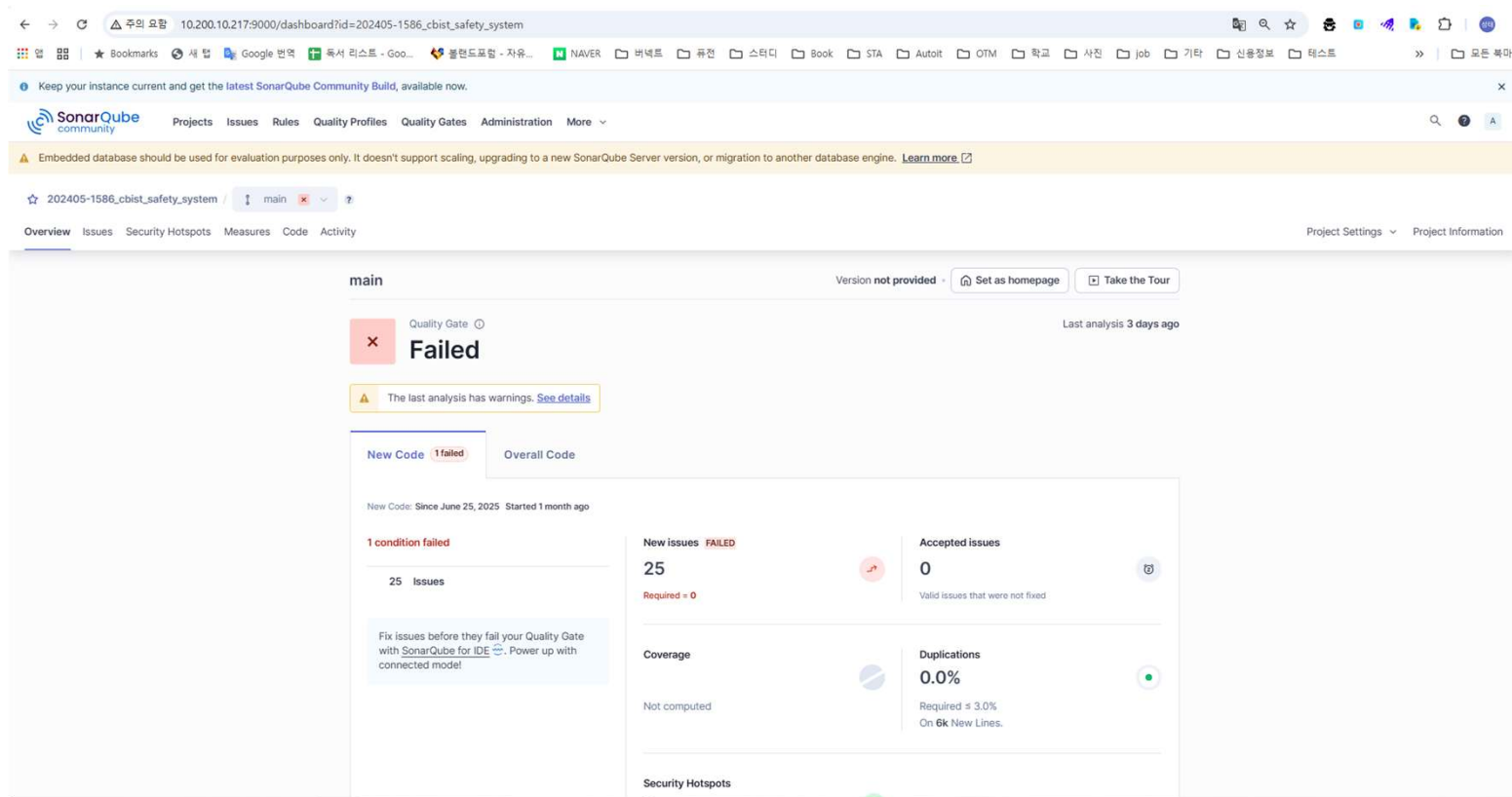
- Coverage: Not computed (Unable to collect)
- New issues: 25
- Duplication: 0.0%
- Quality Gate: X Failed (1 condition failed)
- Last analysis: 3 days ago

### • Key Points

- Unity project also managed under SonarQube.
- Environment makes coverage collection difficult (Test framework not integrated).
- 25 new issues caused Quality Gate failure.

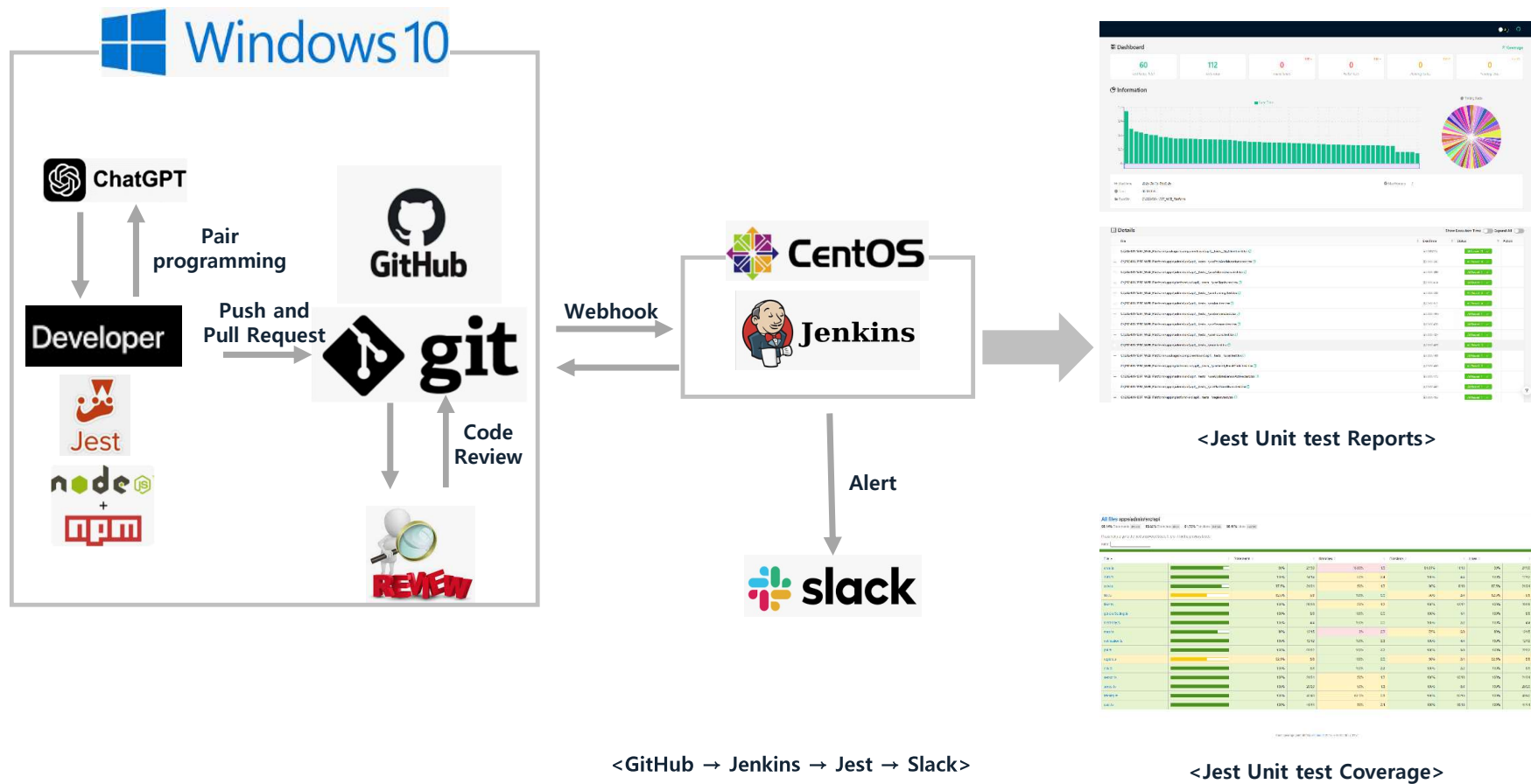
### • Role

- Build SonarQube environment & integrate with CI pipeline.
- Configure quality rules and Quality Gate settings per language.
- Visualize and manage coverage, issues, and code duplication.



# I. Project Overview

## 4. Unit Test Automation Pipeline – Frontend (TypeScript)





# I. Project Overview

## 6. Run the Docker Compose file

### • QA Environment

- Unit tests applied per business domain based on TypeScript/React.

### • Technologies Used

- Measure Statement, Branch, Function, and Line Coverage per feature.

### • Tools Used

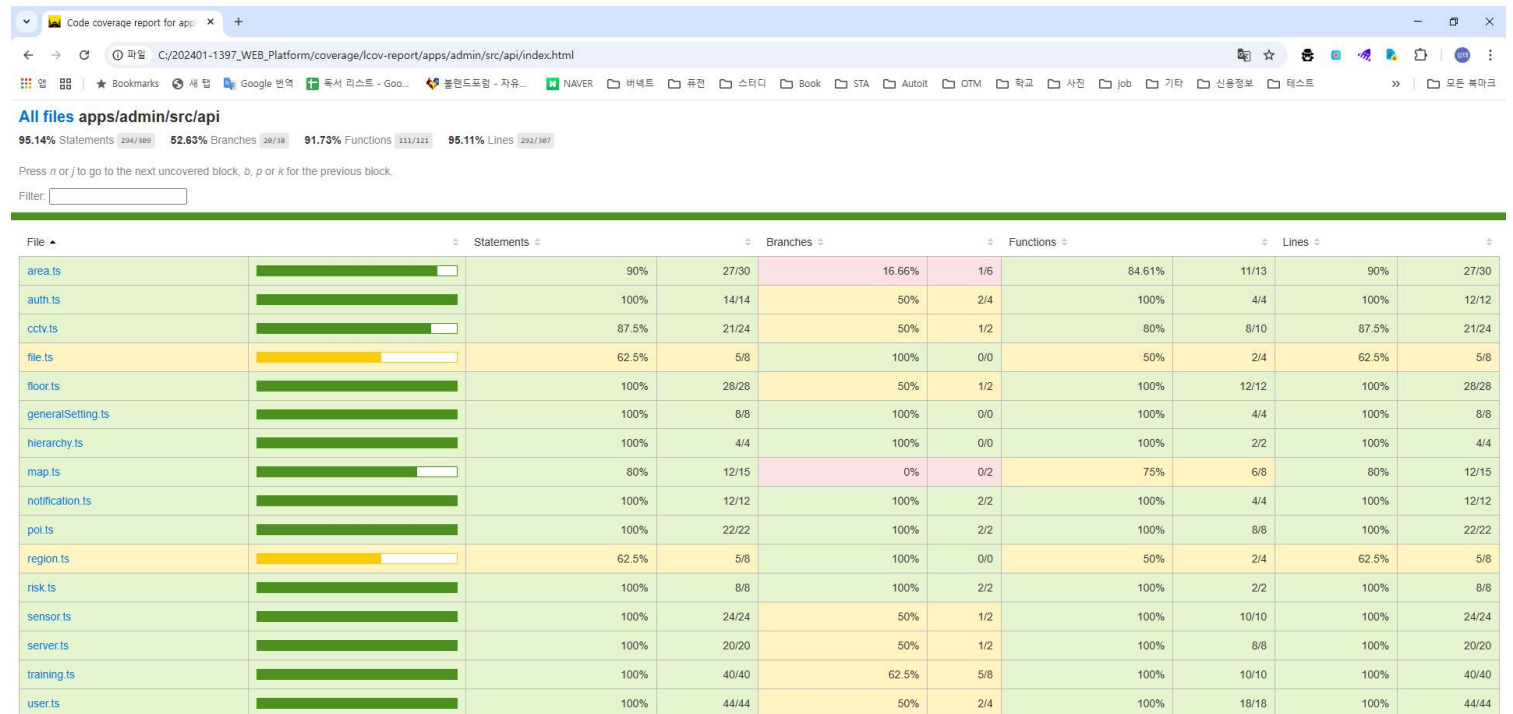
- Jest + Istanbul (Coverage collection)
- HTML Coverage Report (Visualize coverage by file)

### • Achievements

- Achieved 96%+ average Statement coverage overall.
- Achieved 100% Line coverage for most critical business files.
- Identified low Branch coverage (conditional logic) areas.
- Able to visually track per-feature coverage differences for risk management.

### • Role

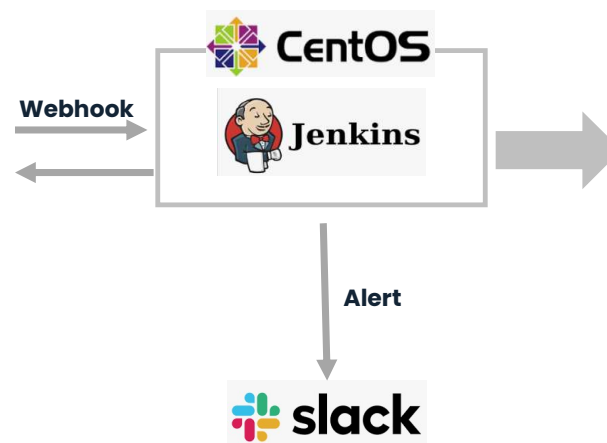
- Standardized frontend test code writing practices.
- Built Jenkins Job & Slack notification automation pipeline.



Code coverage generated by Istanbul at 2025-08-21T05:26:00.239Z

<Jenkins 로그 · Jest HTML Report · Coverage>

The diagram illustrates a development workflow on a Windows 10 system. At the top, the Windows 10 logo is displayed. Below it, a 'Developer' box represents the user. The developer interacts with 'ChatGPT' (represented by its logo) for 'Pair programming'. The developer also performs 'Push and Pull Request' operations using 'git' (represented by its logo) to interact with 'GitHub' (represented by its logo). An 'Extension Pack for Java' (represented by its logo) is shown as a tool used by the developer. The workflow is depicted with arrows indicating the flow of information and code between these components.



## GitHub → Jenkins → JUnit → Slack

[illegible][illegible]

## ◀JUnit test Reports▶

[illegible]

## ◀JUnit test Coverage▶



# I. Project Overview

## 8. Backend (Java) – Test Execution Results & Coverage

### • QA Environment

- Spring Boot-based modular architecture
- Automated test execution in a Jenkins-based CI environment

### • Technologies Used

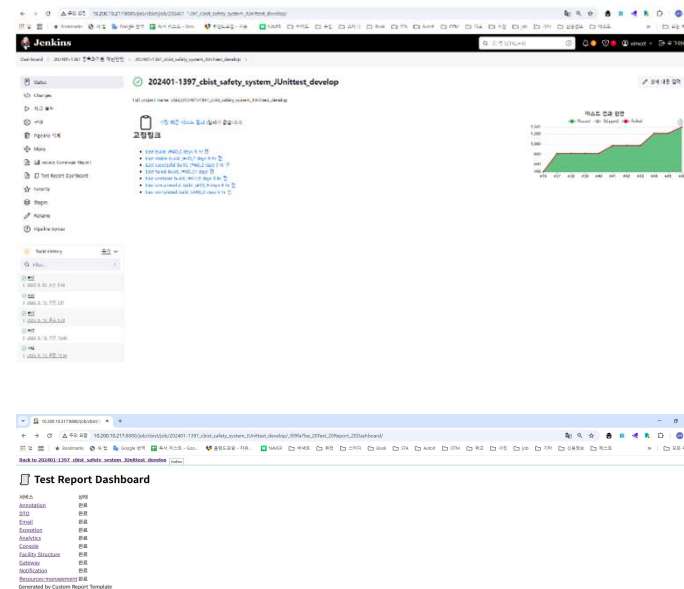
- Java + JUnit-based unit testing
- Verification at Domain / Application / Adapter layer levels

### • Tools Used

- JUnit Report Dashboard
- Jenkins Test Trend (Graph & History)
- Gradle-based automated report generation

### • Achievements

- Recent builds: 1,341 tests executed, 100% success rate
- Established a module-level test execution status dashboard
- Built a foundation for regression test automation and continuous expansion



◀Jenkins Trend · JUnit Report · JaCoCo Coverage▶

# I. Project Overview

## 9. Verify that SonarQube is installed in Docker

### • QA Environment

- Spring Boot-based modular architecture
- Automated test execution in a Jenkins-based CI environment

### • Technologies Used

- Measure Statement, Branch, Line Coverage using JaCoCo
- Detailed coverage tracking per module and detection of uncovered lines

### • Tools Used

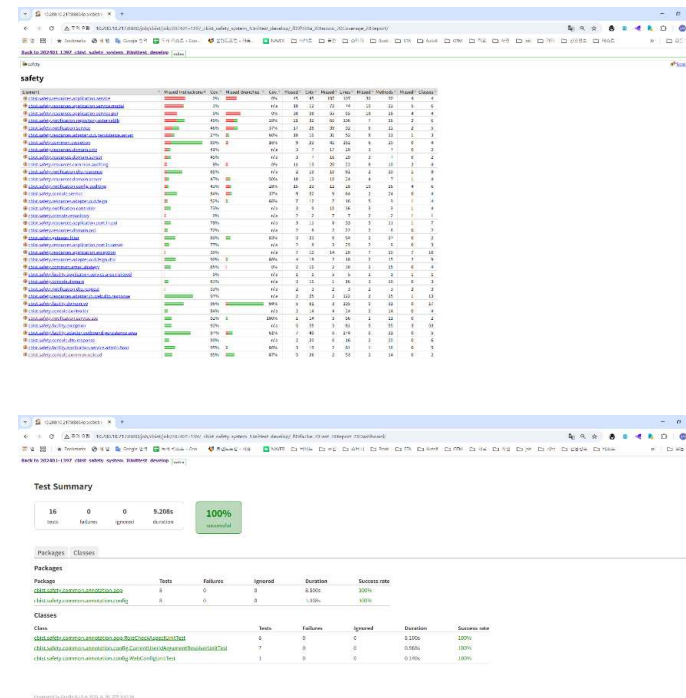
- JaCoCo Coverage Report
- Jenkins Coverage Integrated View
- Gradle automated report generation scripts

### • Achievements

- Achieved 60%+ average Statement coverage overall
- Achieved 90%+ coverage for core business modules
- Identified coverage blind spots for future improvement

### • Role

- Designed and implemented backend service unit test cases
- Configured JaCoCo-based coverage collection and automated reports

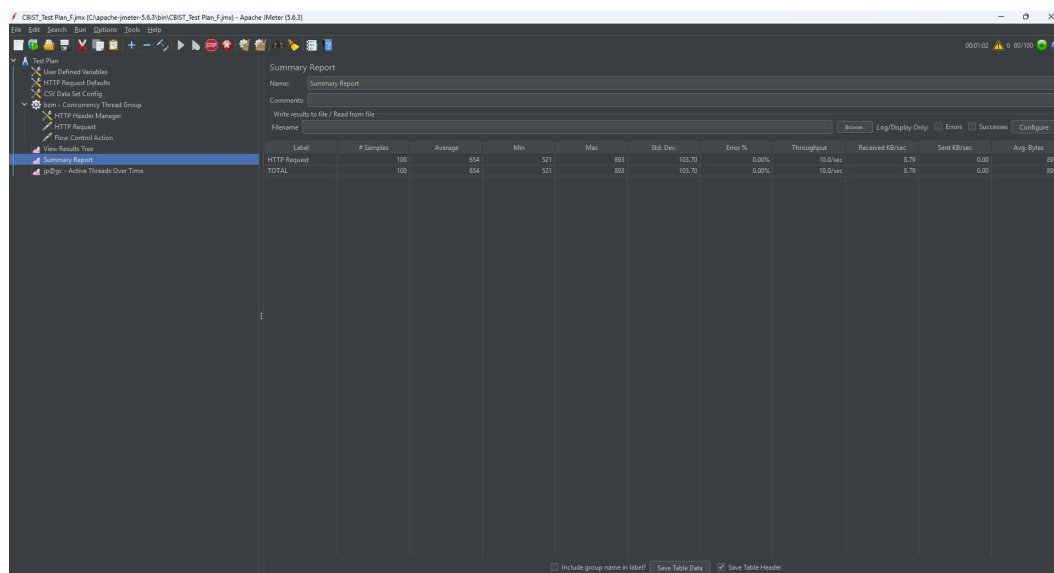


◀ Jenkins Trend · JUnit Report · JaCoCo Coverage ▶

# I. Project Overview

## 10. Performance Testing

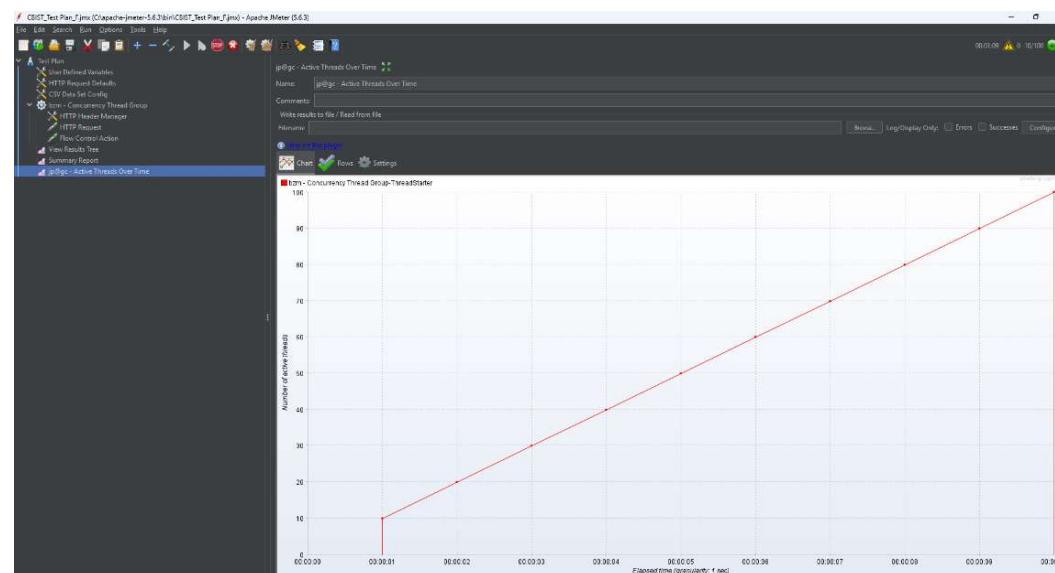
- Summary Report of 100 concurrent users



Summary Report

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	100	654	521	893	103.70	0.00%	10.0/sec	8.79	0.00	899.6
TOTAL	100	654	521	893	103.70	0.00%	10.0/sec	8.79	0.00	899.6

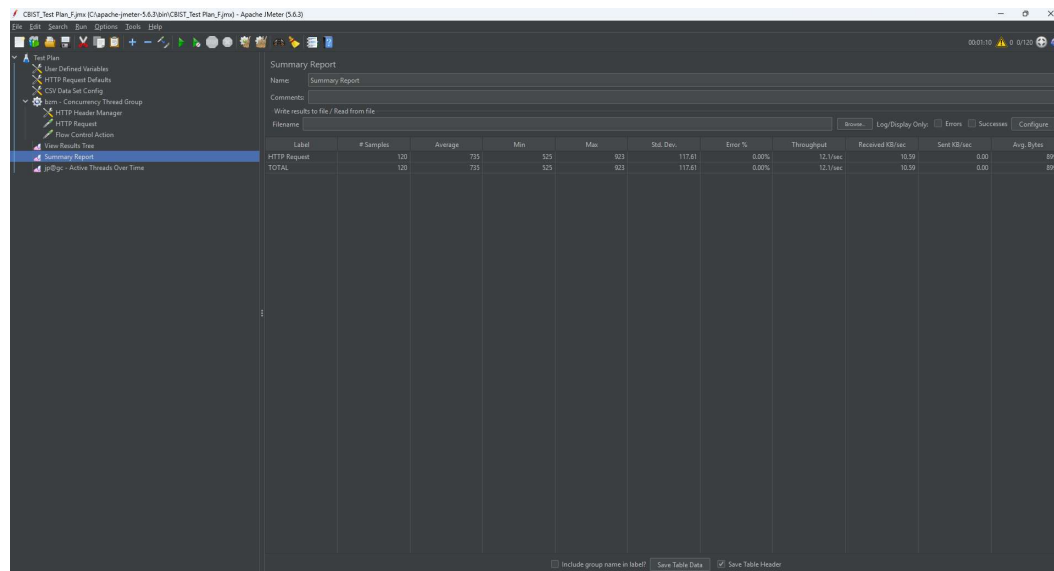
- Graph of 100 concurrent users



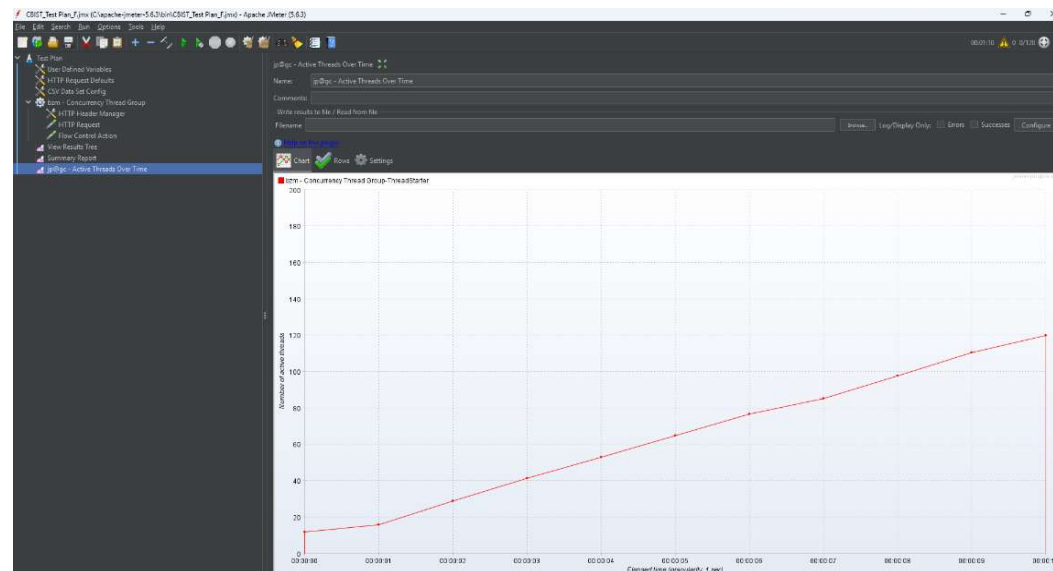
# I. Project Overview

## 10. Performance Testing

- Summary Report of 120 concurrent users



- Graph of 120 concurrent users



# I. Project Overview

## 10. Performance Testing (JMeter)

### • Test Environment

- JMeter-based performance testing environment
- Backend API load and concurrency testing
- Step load pattern with gradual user increase

### • Test Scenarios

#### 100 Virtual Users

- Increased by 10 users step-by-step and sustained

#### 120 Virtual Users

- Increased by 12 users step-by-step and sustained

### • Key Results

#### 100 Users

- Avg Response Time: 654 ms
- Error Rate: 0%, Success Rate: 100%

#### 120 Users

- Avg Response Time: 735 ms
- Error Rate: 0%, Success Rate: 100%

### • Role

- Designed and executed JMeter performance tests
- Analyzed results and supported release readiness assessment

Metric	100 Users	120 Users	Analysis
Average Response Time	654 ms	735 ms	Approx. <b>+12% increase</b> , acceptable
Maximum Response Time	893 ms	923 ms	No significant difference
Error Rate	0%	0%	Highly stable
Success Rate	100%	100%	Successful
Throughput	10.0 / sec	12.1 / sec	Linear increase

### Key Strengths

Excellent response time (under 1 second)

Zero errors → high service reliability

Linear scaling with increased load → good scalability

## Summary Statement

✓ The login API remains stable and operates reliably **up to 120 concurrent users without performance degradation.**



## 2. Project Results

- Successfully deployed and configured SonarQube using Docker and Docker Compose on Linux, establishing a stable environment for continuous code quality analysis.
- Integrated SonarQube with Jenkins CI/CD pipelines, enabling automated static analysis and quality gate enforcement for frontend and backend services.
- Performed static analysis of automated test code using SonarScanner CLI, identifying code smells, bugs, and security vulnerabilities early in the development cycle.
- Built centralized SonarQube dashboards to visualize key quality metrics such as code duplication, coverage trends, and security hotspots, with real-time notifications via Slack.
- Designed and executed JMeter-based performance tests to validate backend API stability under concurrent user load.
- Conducted step-load performance scenarios with 100 and 120 virtual users, confirming that average response times remained under 1 second with 0% error rate and 100% success rate.
- Observed linear throughput growth as concurrency increased, demonstrating stable scalability and supporting release readiness.
- Achieved measurable improvements in test maintainability, system stability, and regression reliability, significantly reducing technical and performance-related risks.