

# TQS: Quality Assurance manual

*Eduardo Rosário [119234], Henrique Lopes [119954], Simão Santos [119042],  
Filipe Marques [120303]  
v2025-06-22*

## Contents

|  |          |
|--|----------|
| <b>TQS: Quality Assurance manual</b>                 | <b>1</b> |
| <b>1 Project management</b>                          | <b>1</b> |
| 1.1 Assigned roles                                   | 1        |
| 1.2 Backlog grooming and progress monitoring         | 1        |
| <b>2 Code quality management</b>                     | <b>2</b> |
| 2.1 Team policy for the use of generative AI         | 2        |
| 2.2 Guidelines for contributors                      | 2        |
| 2.3 Code quality metrics and dashboards              | 2        |
| <b>3 Continuous delivery pipeline (CI/CD)</b>        | <b>2</b> |
| 3.1 Development workflow                             | 2        |
| 3.2 CI/CD pipeline and tools                         | 2        |
| 3.3 System observability                             | 3        |
| 3.4 Artifacts repository [Optional]                  | 3        |
| <b>4 Software testing</b>                            | <b>3</b> |
| 4.1 Overall testing strategy                         | 3        |
| 4.2 Functional testing and ATDD                      | 3        |
| 4.3 Developer facing testes (unit, integration)      | 3        |
| 4.4 Exploratory testing                              | 3        |
| 4.5 Non-function and architecture attributes testing | 3        |

# 1 Project management

## 1.1 Assigned roles

The team consists of four members, each with specific responsibilities while maintaining active collaboration in the development of ToolShed:

### **Filipe Marques - Team Coordinator**

Ensure that there is a fair distribution of tasks and that members work according to the plan. Actively promote the best collaboration in the team and take the initiative to address problems that may arise. Ensure that the requested project outcomes are delivered on time

### **Henrique Lopes - Product Owner**

Represents the interests of the stakeholders.

Has a deep understanding of the product and the application domain; the team will turn to the Product Owner to clarify the questions about expected product features.

Should be involved in accepting the solution increments.

### **Eduardo Rosário - QA Engineer**

Responsible, in articulation with other roles, to promote the quality assurance practices and put in practice instruments to measure the quality of the deployment. Monitors that team follows agreed QA practices

### **Simão Santos- DevOps Master**

Responsible for the (development and production) infrastructure and required configurations. Ensures that the development framework works properly. Leads the preparing the deployment machine(s)/containers, git repository, cloud infrastructure, databases operations, etc.

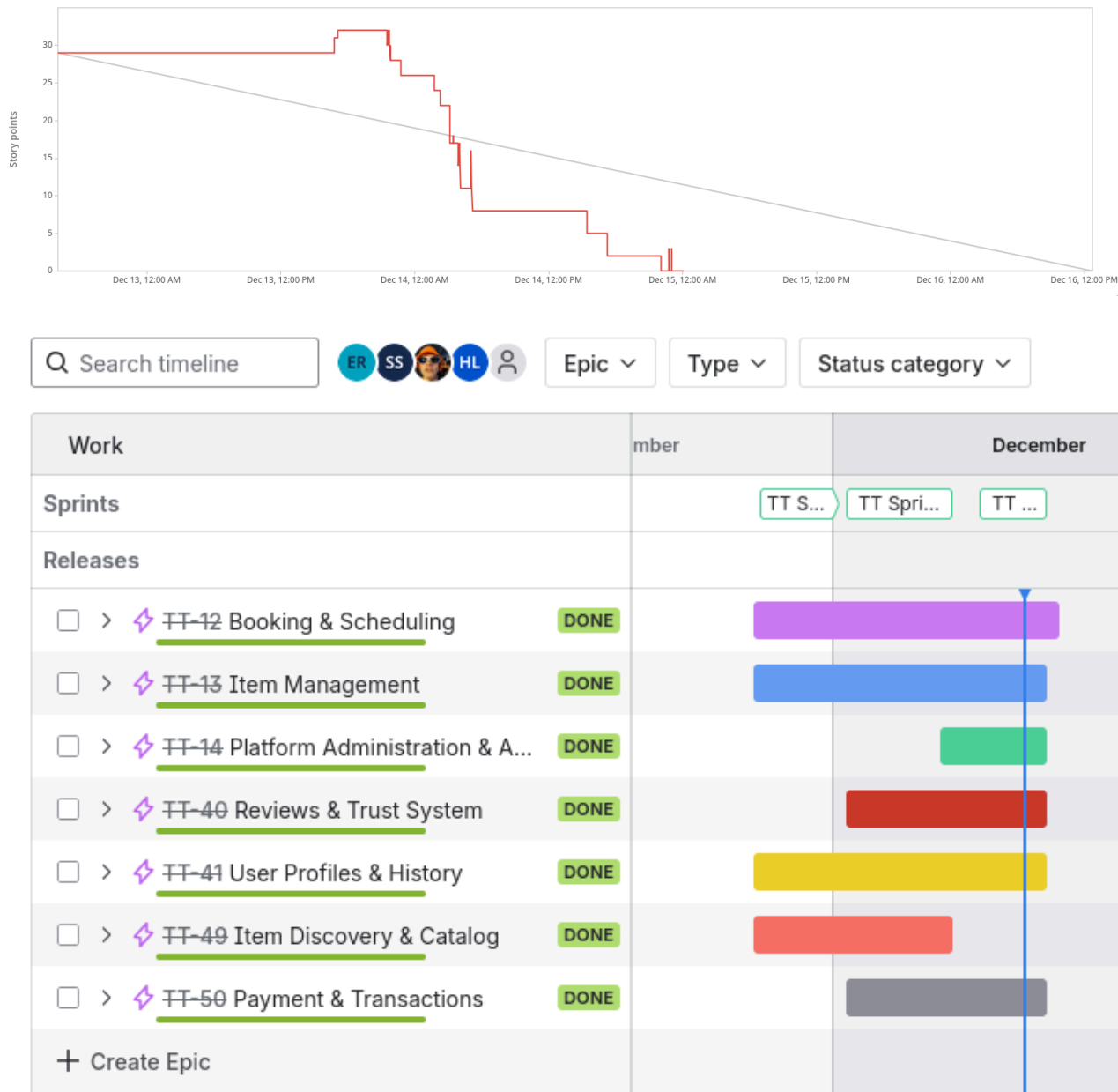
## 1.2 Backlog grooming and progress monitoring

Work is organized in Jira based on User Stories, structured in a backlog, and planned in weekly sprints. In each iteration, stories are reviewed, prioritized, and decomposed into technical tasks to ensure execution readiness. All stories are categorized by functional areas and assigned an owner and effort estimate. Backlog grooming is performed regularly to ensure alignment, requirement clarity, and estimation consistency.

Progress monitoring is conducted via:

- **Story Points:** For effort estimation.
- **Execution States:** To Do, In Progress, In Review, Done.
- **Burndown Charts:** Per sprint analysis.
- **Jira Automation:** Tracking planned vs. completed points.

Additionally, test coverage is continuously monitored via **SonarCloud**. Every pull request triggers an automatic analysis of test coverage, code duplication, security hotspots, and potential vulnerabilities, contributing to continuous improvement and failure prevention.



## 2 Code quality management

### 2.1 Team policy for the use of generative AI

Generative AI was utilized as a support tool throughout the project, specifically for frontend development assistance, debugging support, technical report generation, code review assistance, and test coverage reinforcement. AI usage never replaced team effort; it was applied exclusively

as a support mechanism. It is important to note that AI was not used to generate the application code in its entirety.

## 2.2 Guidelines for contributors

**Coding Style** The project follows a consistent coding style aligned with current best practices to ensure readability, maintainability, and scalability.

- **Backend (Java 21 + Spring Boot 3):** Adopts a layered architecture organized by functionality (Controllers/Boundary, Services, Repositories, and DTOs). Controllers follow the RESTful pattern returning `ResponseEntity`, with OpenAPI documentation. Data validation uses `jakarta.validation`, maintaining separation between JPA entities and DTOs. Tests use JUnit 5, Mockito, and MockMvc (Arrange/Act/Assert pattern).
- **Frontend (TypeScript + React + Vite):** Uses functional components with hooks, organized broadly by domain. Linting is enforced via ESLint, and styling uses Tailwind CSS.

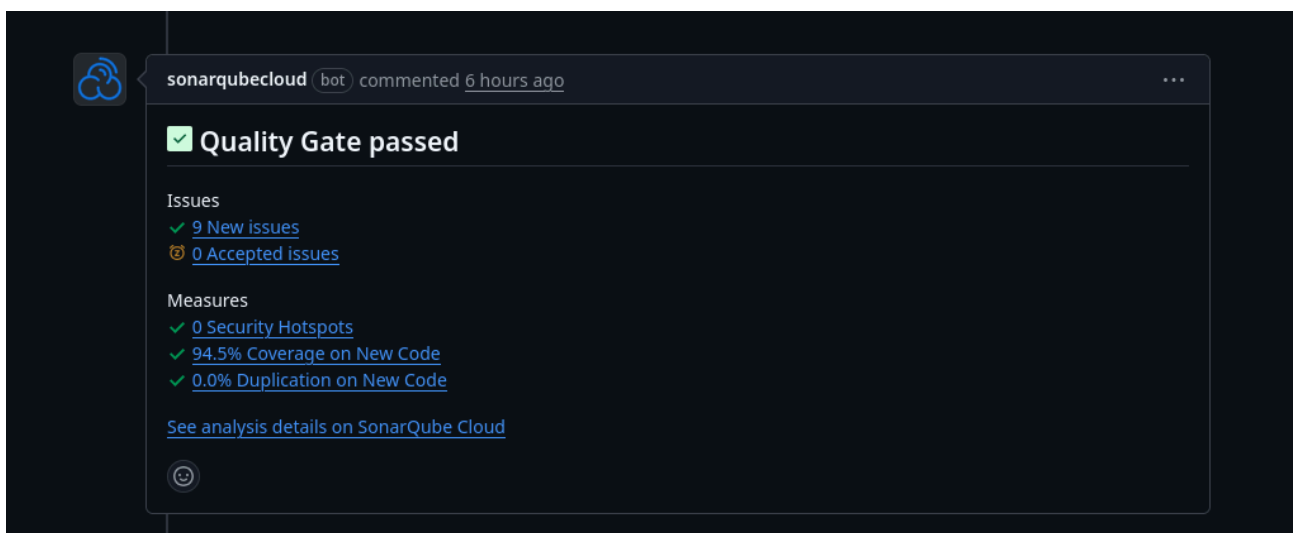
**Code Reviewing** Code is integrated exclusively via Pull Requests (PRs), with mandatory code reviews before merging into the main branch.

Adopted practices include:

- Verification of functional correctness, readability, and compliance with the coding style.
- Validation of adequate tests for developed features.
- **Mandatory manual review** by at least one team member (GitHub Copilot may be used to assist the reviewer).
- Generative AI tools may identify potential issues or optimization suggestions, but the final integration decision remains a human responsibility.

## 2.3 Code quality metrics and dashboards

Static code analysis and continuous quality monitoring are ensured via the CI pipeline and **SonarCloud** integration.



Every PR automatically evaluates:

- Test Coverage.
- Code Duplication.
- Security Hotspots and Vulnerabilities.

**Quality Gates:** Strict minimum quality gates dictate the acceptance or rejection of code integration:

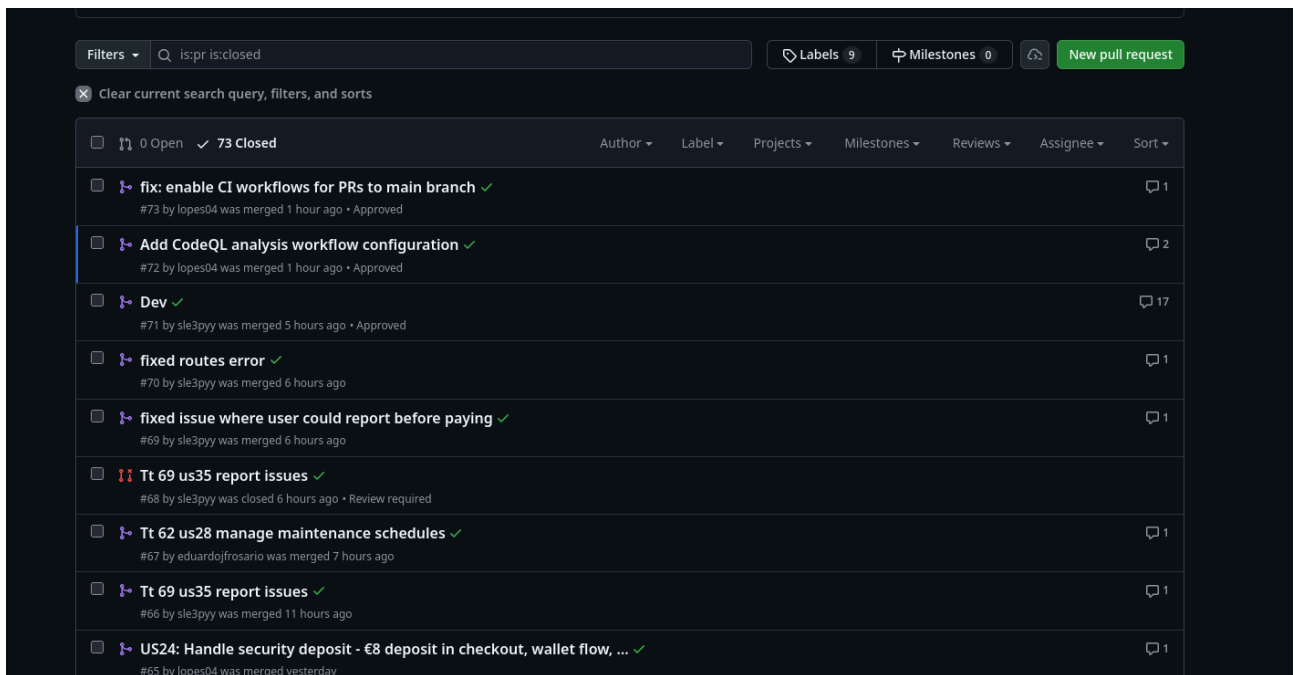
- Minimum Test Coverage (80%).
- 0 Critical Vulnerabilities.
- Limited Code Duplication (<3%).

### 3 Continuous delivery pipeline (CI/CD)

#### 3.1 Development workflow

Work begins in Jira. Each developer selects a "To Do" User Story, moves it to "In Progress," and creates a feature branch. After local implementation and testing, a Pull Request(PR) is created, moving it to "In Review", when a PR is accepted it moves to "Done". The project adopts **GitHub Flow**:

1. One stable main branch (**dev**).
2. One branch per feature/User Story.
3. Integration via Pull Requests.



**Definition of Done (DoD):** A User Story is "Done" only when:

- Functionality meets acceptance criteria.
- Tests are implemented and passing.
- CI Pipeline executes without failure.
- SonarCloud Quality Gates are passed.
- Code is merged into the main branch via approved PR.

## 3.2 CI/CD pipeline and tools

### Continuous Integration (CI)

The project uses **GitHub Actions** (`.github/workflows/build.yml`). It triggers on push and PR events to the dev branch using Ubuntu runners with a PostgreSQL 15 service.

- **Backend:** Automatically checks out code, sets up JDK 21, installs dependencies, and runs full Maven tests. A dedicated job runs SonarCloud analysis (using caching and SONAR\_TOKEN).
- **Tools:** JaCoCo (Coverage), SonarCloud (Quality/Security), JUnit 5/Mockito/MockMvc (Testing).
- **Frontend:** ESLint configured for local validation.

**Continuous Delivery (CD)** *Note: Currently, the project utilizes Dockerfiles and docker-compose.yml for local container orchestration. Automated deployment to a staging environment is part of the roadmap for post-MVP iterations.*

|   |   |         |          |          |                            |
|---|---|---------|----------|----------|----------------------------|
| CD - Deploy to VM<br>CI<br>CI<br>CodeQL Advanced<br>Copilot code review<br>Copilot coding agent<br>SonarCloud | 345 workflow runs   | Event ▾ | Status ▾ | Branch ▾ | Actor ▾                    |
|   | <div>● Merge pull request #72 from TQS-ToolShed/lopes04-patch-1</div> <div>CI #144: Commit e773252 pushed by lopes04</div>            |         |          | main     |                            |
|   |   |         |          |          | Today at 3:10 AM<br>1m 55s |
|   | <div>● Merge pull request #72 from TQS-ToolShed/lopes04-patch-1</div> <div>CodeQL Advanced #3: Commit e773252 pushed by lopes04</div> |         |          | main     |                            |
|   |   |         |          |          | Today at 3:10 AM<br>3m 23s |

3.3 System observability

Currently, observability relies on SonarCloud for code health. Runtime observability (logs/metrics) is managed via container logs during local execution.

4 Continuous testing

4.1 Overall testing strategy

[<https://github.com/TQS-ToolShed/toolshed/tree/dev/backend/src/test>]

The strategy follows a layered approach, ensuring progressive validation from business logic to the API. We combine Unit Tests, Integration (Slice) Tests, and Context Loading tests. Execution is automated in the CI pipeline, acting as a mandatory gateway for code integration.

4.2 Acceptance testing and ATDD

Acceptance tests are defined from a black-box perspective, validating User Story acceptance criteria.

- **Current Status:** Tests are manually validated against requirements.
- **Roadmap:** Implementation of automated BDD (Cucumber) and E2E (Playwright) frameworks is planned for future iterations to formalize this process.

4.3 Developer facing tests (unit, integration)

**Unit Testing:** Follows a white-box approach focusing on business logic (Service Layer) with mocked dependencies.

- **Tools:** JUnit 5, Mockito, AssertJ.
- **Policy:** Mandatory for all business logic; must cover success, edge cases, and error scenarios.

**Integration Testing:** Validates the interaction between layers using Spring Boot Slice Tests:

- **Repositories:** Tested against an H2 in-memory database.
- **Controllers:** Tested via **MockMvc** to validate HTTP responses and JSON structures without full server startup.

#### 4.4 Exploratory testing

The project strategy prioritizes **Automated Regression** over manual scripted testing. Consequently, the team does not conduct dedicated manual exploratory testing phases. Instead, a **Developer-Led Verification** approach is adopted

- **Implementation Validation:** Developers manually verify the happy path and edge cases of a feature locally during development before committing code.
- **Reliance on Automation:** Once code is pushed, the team relies on the extensive suite of Unit and Integration tests in the CI pipeline to catch regressions, rather than manual re-testing.
- **Pull Request Review:** Visual verification is performed ad-hoc during the Code Review process by the reviewer, focusing solely on UI/UX correctness that cannot be easily asserted via code.

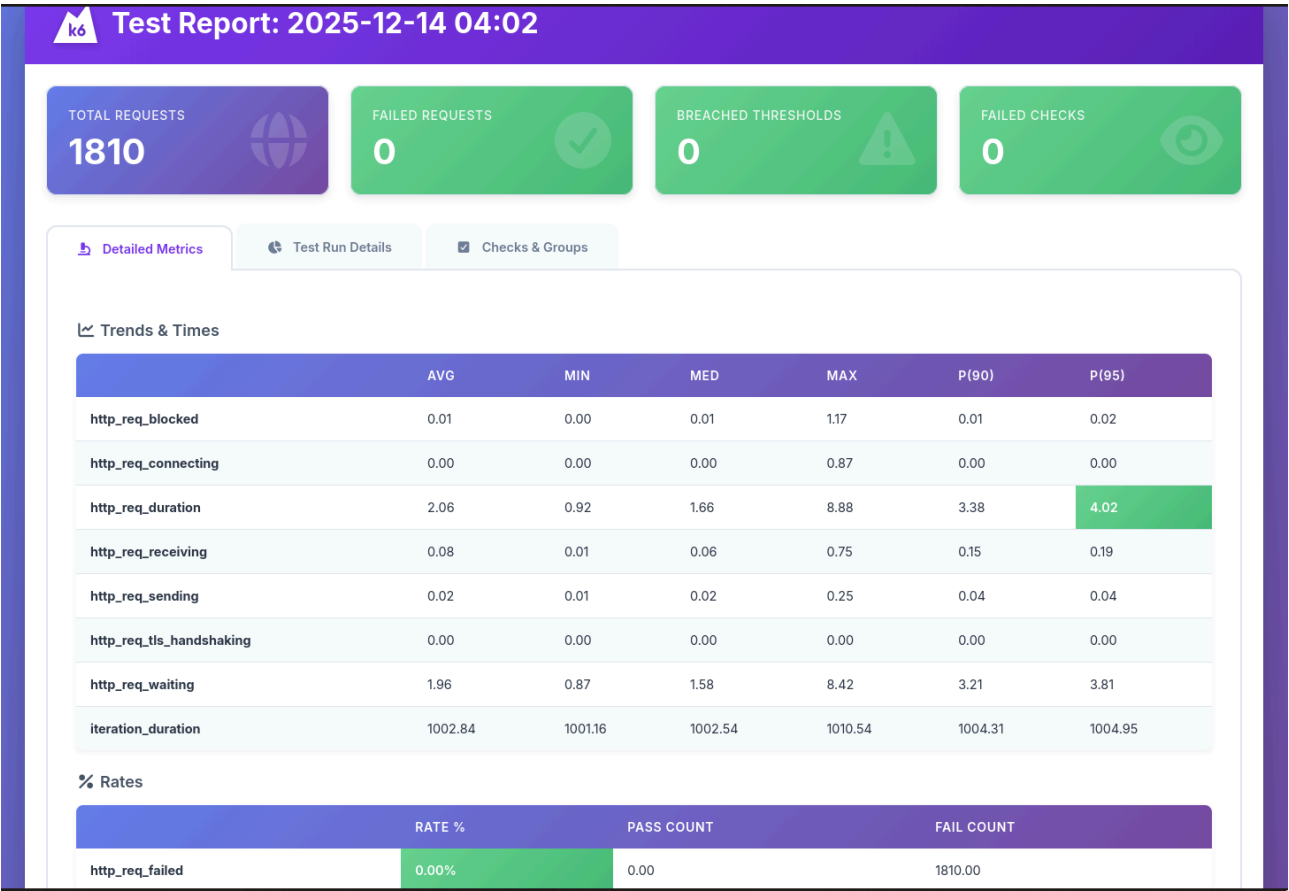
#### 4.5 Non-function and architecture attributes testing

To validate the architecture's reliability, the team performs **Load Testing** using **K6**.

**Policy:**

- Performance tests are executed against critical endpoints (e.g., Login, Item Search) to ensure response times remain under acceptable limits (e.g., < 500ms) under concurrent load.
- **Tool:** K6 scripts allow us to simulate multiple virtual users accessing the API simultaneously.





To ensure a high-quality user experience and compliance with web standards, the team utilizes **Google Lighthouse** to audit the frontend application. This tool allows us to validate non-functional requirements related to performance, accessibility, and best practices.

**Policy:** Critical pages (e.g., Home Page, Item Details, Login) are subject to Lighthouse audits before major releases. The team aims for a "Green" score (>90) in the Accessibility and Best Practices categories.

**Metrics Monitored:**

- **Performance:** Analysis of Core Web Vitals, specifically *Largest Contentful Paint (LCP)* and *Cumulative Layout Shift (CLS)*, to ensure the UI loads quickly and remains stable.
- **Accessibility:** Validation of ARIA labels, color contrast ratios, and keyboard navigation support to ensure the platform is usable by all users.
- **SEO & Best Practices:** Checks for HTTPS usage, correct metadata, and secure JavaScript practices.

