deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Ana Alexandra Antunes [876543], Bruno Bernardes [876543]
v2025-04-25

Contents

1 Project management

1.1 Assigned roles

José Mendes, Product Owner: Jose owns the vision for the product and speaks for our stakeholders. She writes and prioritizes user stories, makes sure acceptance criteria are clear, and gives the final “go” on each feature before it ships. By keeping a tight feedback loop with the team, she ensures we’re always building the highest-value work first.

Luís Godinho, QA Engineer: Luis is our quality guardian. He designs and runs both manual and automated tests against every new feature, tracks and verifies bug fixes, and maintains our test frameworks and dashboards. By partnering early with Alice on story definitions and with DevOps on CI/CD integration, he helps us catch issues before they reach production.

Tiago Lopes, DevOps Engineer: Tiago automates and safeguards our delivery pipeline. She writes infrastructure-as-code, deploys environments, monitors system health, and handles rollbacks or hotfixes when needed. Working hand-in-hand with Marcus, she ensures our tests run smoothly in each build, and with Alice, she sets up demo environments for stakeholder reviews.

José Marques, Team Leader: Jose keeps the engine running. He organizes sprint planning, daily stand-ups, retrospectives, and steps in to clear blockers or deflect unnecessary distractions. By tracking progress against our milestones and coaching team members on agile best practices, he helps Alice's roadmap become reality, with Marcus and Priya supporting every step of the way.

1.2 Backlog grooming and progress monitoring

In our JIRA setup, we organize work primarily through a structured hierarchy where Epics are used to group related user stories and tasks. Each Epic represents a high-level feature or objective, and it's broken down into smaller, manageable stories and sub-tasks that contribute to the overall goal. This structure helps maintain clarity and traceability from top-level initiatives down to individual development efforts.

To track progress on a regular basis, we rely on story points for estimating the complexity and effort of each story. These estimates provide a quantifiable measure that feeds into velocity tracking and sprint planning. Throughout the sprint, we use burndown charts to monitor progress, visualizing the remaining work versus the time left in the sprint. This helps the team assess whether we are on track to complete the committed work and make timely adjustments if needed.

For quality assurance and test coverage, we use Xray, a test management tool integrated within JIRA. Xray allows us to define, manage, and associate test cases directly with stories and requirements. This ensures that each requirement is covered by appropriate tests and facilitates requirements-level coverage monitoring. The integration enables proactive oversight of testing progress and traceability from requirements to test execution results, helping us maintain a high standard of quality and compliance throughout the development lifecycle.

2 Code quality management

2.1 Team policy for the use of generative AI

Team Position on AI Assistants

Our team embraces AI assistants as a valuable resource for specific, well-defined tasks, but not as a substitute for developer expertise. In both production and test code, we allow AI tools to help generate sample data sets, boilerplate configurations, or to suggest approaches when encountering stubborn errors. However, we do **not** delegate the design or implementation of complete features or entire test suites to AI. Every line of business logic, security check, or critical validation must be authored, reviewed, and understood by a human engineer.

Practical Advice for Newcomers

When you join the team, think of AI as a "smart aide" rather than an author. Use it to scaffold repetitive

tasks, such as crafting mock-data schemas, generating stub implementations for hard-to-mock dependencies, or proposing unit-test edge cases you hadn't considered. If you get stuck on an obscure runtime error or a tricky merge conflict, feel free to query an AI for hints on root causes or potential fixes. But always read, validate, and adapt any AI-generated snippet before merging it into our codebase. Trust but verify.

2.2 Guidelines for contributors

Coding style

Our team strictly adheres to the AOS project's established coding conventions to ensure consistency, readability, and maintainability across the entire codebase. This framework encompasses precise rules for formatting (such as indentation, line length, and brace placement), clear naming conventions for variables, functions, classes, and modules, as well as comprehensive documentation standards that dictate how inline comments, docstrings, and external API references should be written and organized. By following these guidelines, and by using automated linters and style-checkers during development and pull-request reviews, we guarantee that every contribution aligns with our shared quality objectives and remains easy for all team members to understand and extend.

Code reviewing

In our development workflow, **code reviews are mandatory** and must be completed by at least one designated reviewer before any pull request (PR) can be merged. This ensures code quality, encourages team collaboration, and helps maintain shared understanding and ownership of the codebase.

When to Perform Code Reviews

Code reviews are initiated as soon as a developer creates a pull request. Ideally, reviews should be performed **promptly after the PR is submitted**, to avoid blocking progress and reduce context-switching for both the author and reviewer. It's important to keep PRs focused and of a manageable size to facilitate timely and thorough reviews.

To maintain flow within sprints and reduce bottlenecks, reviewers should allocate dedicated time each day for reviewing code. This ensures the review process becomes a regular, predictable part of the development cycle, rather than a last-minute checkpoint.

Code Review Guidelines

Reviewers are expected to go beyond syntax or formatting checks and focus on the following aspects:

- **Correctness**: Does the code do what it claims to do? Are edge cases handled properly?
- **Readability**: Is the code easy to understand? Could variable/method names or structure be improved?

- **Testing**: Are there appropriate tests? Do they cover the relevant scenarios and requirements?

Feedback should be **constructive and actionable**, with a focus on improvement rather than criticism. Where appropriate, suggestions should be backed by examples or references to standards and guidelines.

Review Acceptance

A pull request can only be merged once **at least one reviewer has approved it**. Any critical issues raised during review must be addressed by the author, and changes should be re-reviewed before approval is granted.

To maintain transparency and traceability, all comments and decisions are recorded within the PR discussion thread. This allows the team to understand the context behind decisions and provides a historical record for future reference.

2.3 Code quality metrics and dashboards

Our team enforces rigorous contribution standards to maintain high code quality. Every new feature or change must be backed by tests that cover at least 80 % of the affected code. Before any production code is written, the corresponding tests need to be created and must pass successfully. We also integrate SonarCloud into our pipeline to catch security flaws early, and no pull request may proceed if any issues are flagged. Finally, every merge request requires approval from at least one other team member, ensuring that all changes receive a fresh pair of eyes before they enter the main branch.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

Our team employs the GitFlow model on GitHub to manage our version control. For every feature or user story, we first open a corresponding issue that captures its requirements and acceptance criteria. From each issue, we create a dedicated feature branch off of the relevant epic branch, this keeps related work grouped together. When your feature is ready, you submit a pull request back into that epic branch, triggering our peer-review process to verify code quality and consistent use of our coding standards. Once all feature branches under an epic are merged and the epic branch itself reaches a stable, release-ready state, we open a final pull request into the develop branch. After thorough integration testing, the develop branch is promoted to main via one more pull request, marking a clean, deployable milestone. Throughout this workflow, every change is traceable back to its originating issue, and every pull request undergoes review to maintain our high standards.

Definition of done

Our team's **Definition of Done (DoD)** for a user story represents a shared agreement on the minimum criteria that must be met before a story is considered complete. A user story is only marked as done when the implementation has been fully developed, peer-reviewed through a

pull request and approved by at least one team member, with all feedback addressed. The code must be merged into the appropriate branch without introducing any regressions or build failures.

In addition to the technical delivery, the story must have corresponding **unit tests and, when applicable, integration or UI tests**, ensuring appropriate coverage of the functionality. These tests must pass in the CI pipeline, confirming that the feature behaves as expected in the intended environment. The story must also be validated against the acceptance criteria defined at the time of planning, ensuring that the feature meets the user's needs.

Lastly, the feature must be demonstrable and ready for release or inclusion in a larger increment of value, meaning it is production-ready in both quality and stability. Only when all these conditions are satisfied can the story be officially considered done.

3.2 CI/CD pipeline and tools

In our project, we have established a robust Continuous Integration (CI) and Continuous Delivery (CD) pipeline to ensure the ongoing integration of code increments and smooth deployment of software. Continuous Integration plays a crucial role in our development process by enabling early detection and resolution of issues through frequent merging of code changes into the development branch. Our CI pipeline is fully automated using GitHub Actions.

We have set up two main GitHub Actions workflows, the first focuses on SonarCloud analysis and importing test results into XRay. This workflow is triggered by push events to the main and development branches, as well as by pull requests when they are opened, updated, or reopened.

```yaml
name: SonarQube
on:
  push:
    branches:
      - main
      - dev
  pull_request:
    types: [opened, synchronize, reopened]
jobs:
  build:
    name: Build and analyze
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0
      - name: Set up JDK 21
        uses: actions/setup-java@v4
        with:
          java-version: 21
          distribution: 'temurin'
      - name: Cache SonarQube packages
        uses: actions/cache@v4
        with:
          path: ~/.sonar/cache
          key: ${{ runner.os }}-sonar
          restore-keys: ${{ runner.os }}-sonar
      - name: Cache Maven packages
        uses: actions/cache@v4
        with:
          path: ~/.m2
          key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
          restore-keys: ${{ runner.os }}-m2
      - name: Build and analyze
        env:
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        working-directory: ./backend
        run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=TQS-03_deti-tqs-03
      - name: Import results to Xray
        uses: mikepenz/xray-action@v3
        with:
          username: ${{ secrets.XRAY_CLIENT_ID }}
          password: ${{ secrets.XRAY_CLIENT_SECRET }}
          testFormat: "junit"
          testPaths: "**/surefire-reports/TEST-*.xml"
          testExecKey: "SCRUM-30"
          projectKey: "SCRUM"
```

Img 1. Sonarqube integration workflow

The second workflow, focuses on building the project and running the unit tests. These are triggers from pushing to the main branch as well as upon creating a pull request.

```yaml
name: Java CI with Maven

on:
  push:
    branches: [ "main", "dev" ]

  pull_request:
    types: [opened, synchronize, reopened]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up JDK 21
      uses: actions/setup-java@v3
      with:
        java-version: '21'
        distribution: 'temurin'

    - name: Build and run unit tests with Maven
      working-directory: ./backend
      run: mvn clean test
      continue-on-error: false
```

Img 2. Unit tests workflow

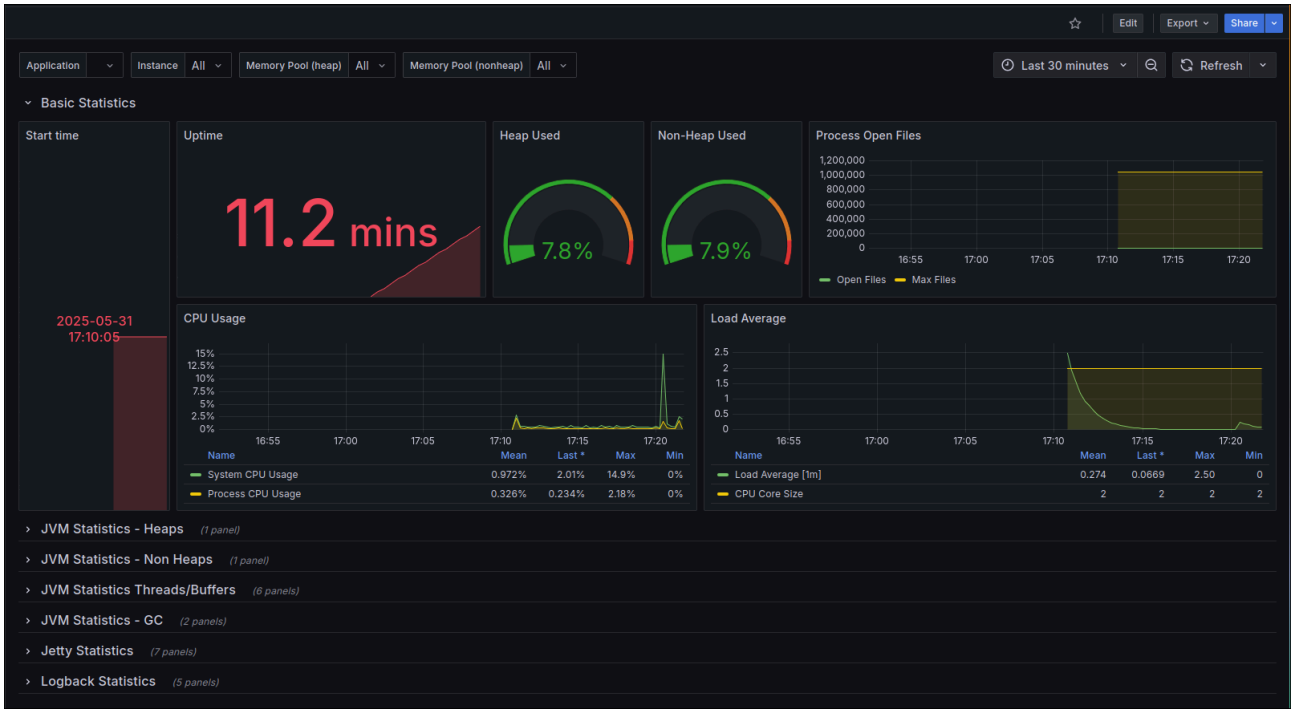
3.3 System observability

To enhance system observability, we implemented a Grafana dashboard backed by Prometheus, which collects and visualizes key metrics from the application. These metrics are exposed through Spring Boot's Actuator endpoints, enabling Prometheus to scrape them at regular intervals. The actuator provides comprehensive operational data, allowing us to monitor the health and performance of the application in real time.

This setup captures a wide range of metrics, including CPU usage, memory consumption, system uptime, load averages, thread statistics, and garbage collection activity. These metrics are stored as time-series data in Prometheus, making it possible to analyze trends, detect anomalies, and understand system behavior over time.

The data is presented in Grafana using custom dashboards that offer both a high-level overview and deep dives into specific areas of interest. This visibility helps teams quickly identify issues, assess the impact of changes, and make informed decisions about scaling and optimization. Additionally, the system supports alerting based on predefined thresholds or patterns, ensuring that potential problems are detected and addressed proactively.

By combining Prometheus, Grafana, and Spring Boot Actuator, we've established a robust observability framework that supports continuous monitoring, performance tuning, and reliable service operation.



Img 3. Grafana Dashboard

4 Software testing

4.1 Overall testing strategy

Our test development strategy followed a pragmatic and layered approach, using best practices from Test-Driven Development (TDD) and Behavior-Driven Development (BDD).

For core business logic and unit-level components, we adopted TDD where applicable, writing tests before implementation to drive design decisions and ensure comprehensive coverage. This practice helped us maintain a high standard of code quality and encouraged modular, testable components.

Complementing TDD, we also incorporated BDD practices—particularly for higher-level features and acceptance criteria—by writing human-readable scenarios that bridge the gap between technical and non-technical stakeholders. This facilitated better communication, shared understanding of requirements, and more focused testing of expected behaviors.

We used JUnit and Mockito for unit and service-level tests, ensuring fast feedback loops during development. The test suite was structured to clearly separate unit, integration, and acceptance tests, allowing for targeted execution depending on the context.

All tests are integrated into the CI/CD pipeline, where they are automatically executed on each commit or merge request. The CI system is configured to fail the build if any test does not pass, ensuring that regressions or broken functionality are immediately caught and addressed. Test results are published as part of the CI run artifacts and are accessible for inspection through the build server's dashboard. Code coverage metrics and trend analysis are also monitored to maintain visibility into the health and evolution of the test suite.

This strategy ensures that tests are not only consistently written and maintained, but that they are also a first-class citizen in our development workflow, directly contributing to the stability, reliability, and quality of the software we deliver.

4.2 Functional testing and ATDD

Functional testing in our project is based on the closed-box method, checking the system from the user's point of view with no regard to how things are implemented internally. These tests check that the application is working as expected in realistic usage situations along the entire stack from UI to back-end services. The tests are programmed to mimic user actions, check expected behavior based on business rules, and provide confidence in the functionality of the system in the face of modifications.

Developers would be expected to add functional tests whenever they add a new user-facing feature. Such tests would check all acceptance criteria defined and be representative of normal user workflow. When fixing bugs, one functional test is to be added to verify that the problem is solved and avoid regressions. When significant refactoring or architectural changes are performed—particularly those that have the potential to affect user behavior—those existing functional tests are to be maintained and new ones added as necessary to maintain coverage.

We employ Selenium, Cucumber, and Testcontainers together to deliver our functional test suite. Cucumber facilitates writing our test scenarios in BDD (Behavior-Driven Development) style, with tests that are simple to understand and readable by both technical and business stakeholders. Selenium is employed to perform realistic browser simulations, allowing user flows to be verified in mock production-like conditions. To enable full-stack integration in tests, we use the Testcontainers library to create isolated versions of our application and its dependencies, providing realistic and stable test environments.

All functional tests are part of our CI pipeline and are run periodically to detect regressions and check system behavior. These tests are essential to ensure software stability and quality, particularly as the system changes. We encourage developers to prioritize functional testing as a primary task and not leave it to later on, and make tests align with business expectations and user journeys.

4.3 Developer facing tests (unit, integration)

Unit testing played a central role in our development strategy, forming the foundation of our Test-Driven Development (TDD) approach. These tests were implemented in Java using the JUnit framework and focused on validating the functionality of individual components and classes in complete isolation from external dependencies. By testing each unit independently, we ensured that the internal logic of every component was thoroughly verified before it was integrated into the broader system.

This methodical separation of concerns allowed us to build confidence in the correctness and stability of each part of the application. Unit tests were designed to be deterministic, fast to execute, and easy to maintain, which made them ideal for supporting frequent iterations during development. The use of mocking frameworks, such as Mockito, further enabled precise control over dependencies, allowing us to simulate various edge cases and error conditions effectively.

The rigorous application of unit tests led to early detection of defects, often during the coding phase itself. This significantly reduced the likelihood of issues surfacing during integration or production, streamlining the overall development cycle and improving software reliability. Moreover, the presence of a comprehensive unit test suite provided a safety net for future changes, enabling developers to refactor code or introduce new features with confidence, knowing that any regressions would be immediately flagged.

Over time, this disciplined testing practice contributed to higher code quality, better maintainability, and a more predictable and efficient development process.

4.4 Non-function and architecture attributes testing

Our performance testing policy is designed to verify that every release satisfies established response time and scalability objectives before it reaches production. To enforce this, we standardized on using k6 as our primary performance testing tool, with all k6 scripts written in JavaScript and maintained in a dedicated repository alongside the application code. This ensures that any change with the potential to impact performance is evaluated immediately.

Each k6 script simulates realistic user behavior, ranging from routine usage spikes to sustained high-load patterns, and is configured with explicit thresholds for response times, error rates, and throughput. If any threshold is exceeded during a test run, the CI pipeline flags the issue and halts further progression until the root cause has been identified and remedied. In this way, performance regressions are caught early, preventing degraded user experiences in downstream environments.

Performance tests are executed against a mirrored staging environment that closely replicates production, encompassing similar network configurations, and data volumes. Prior to merging any new feature branch into the mainline, our CI workflow automatically provisions the test environment, triggers the appropriate k6 scenario, and collects detailed metrics on response latencies, CPU and memory utilization, and throughput. These results are archived in a centralized dashboard, allowing engineering teams to track performance trends over time and compare them against documented service-level agreements.

Our SLAs are maintained in a version-controlled performance requirements reference document. This living document defines acceptable ranges for key metrics, such as 95th percentile response times under a specified user load, and outlines escalation paths should any tests reveal violations. By keeping these requirements transparent and up to date, all stakeholders remain aligned on both current performance expectations and long-term scalability goals.