universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

# TQS: Quality Assurance manual

*Ana Rita Fernandes da Silva [114220], Ângela Maria Martins de Abreu Ribeiro [109061], Carolina Sofia Pereira da Silva [113475], Hugo Gonçalo Lopes Castro [113889]*
v2025-04-25

## Contents

# 1 Project management

## 1.1 Assigned roles

The team consists of four members, each assigned a specific role while contributing collaboratively as developers. Their responsibilities are as follows:

Hugo Castro oversees the overall coordination and task distribution within the team. He ensures that work is aligned with the project plan, fosters effective collaboration, and takes the initiative to resolve issues as they arise. Hugo is accountable for ensuring that project deliverables are completed on time and meet the required standards.

Rita Silva represents the stakeholders' interests and has a deep understanding of the product vision and application domain. She is responsible for clarifying requirements, defining product features, and validating solution increments. The team turns to Rita for guidance on product expectations and feature prioritization.

Angela Ribeiroleads the quality assurance efforts. She is responsible for promoting QA practices within the team, implementing tools and processes to measure and ensure product quality, and monitoring adherence to agreed QA standards throughout development and deployment.

Carolina Silva oversees the development and production infrastructure. She is responsible for configuring and maintaining the development framework, deployment environments (machines/containers), version control systems, cloud infrastructure, and database operations. Carolina ensures that the team’s technical environment supports continuous integration and delivery effectively.

## 1.2 Backlog grooming and progress monitoring

The team manages the project backlog and sprint planning using Jira Cloud with the Scrum template. The backlog is organized into Epics, User Stories, and Tasks, reflecting the requirements and functionalities described in the product specification report.

The backlog is continuously refined through weekly backlog grooming sessions, led by the Team Leader. During these sessions, the team:

- Reviews existing user stories for clarity and completeness
- Adds new stories or tasks based on evolving understanding of requirements
- Splits or merges stories if needed for better estimation or planning
- Updates priorities and story points estimation
- Ensures acceptance criteria are clearly defined

Each sprint starts with a sprint planning session, where the team selects a subset of backlog items to work on based on team capacity and priorities. Progress is tracked using story points estimation to assess workload, flow diagrams and Jira dashboards summarizing issue status across epics and sprints.

Each user story or task is linked to the corresponding test cases in Xray (integrated into Jira) to ensure traceability from requirements to tests and defects. This allows proactive monitoring of requirement-level test coverage using Xray’s coverage analysis features and reports.

The team uses Jira notifications and alerts to stay informed of issue status changes, overdue tasks, or sprint scope adjustments. During sprint reviews, the progress towards overall project goals is validated by comparing completed stories and test coverage against planned deliverables.

# 2 Code quality management

## 2.1 Team policy for the use of generative AI

Our team is open and supportive of using AI assistants to aid in coding, as long as the developer fully understands and takes ownership of the code produced. AI-generated code should never be accepted blindly: always review, test, and validate the suggestions before integrating them into the project. For newcomers, it is critical to approach AI-generated solutions thoughtfully—verify their correctness, ensure they align with project standards, and understand their functionality. AI tools are highly recommended for writing repetitive code, generating simple tests, or accelerating front-end development for straightforward tasks.

However, exercise caution when using AI to create complex functions or critical components on which other code depends; these may appear correct initially but can introduce subtle issues or long-term maintenance challenges, especially if not fully understood. In testing, if an AI-generated fix resolves a failing test, be wary of superficial solutions or "band-aid" fixes that mask underlying problems rather than solving them. Finally, keep in mind that AI-generated code may lack considerations for security, performance, or clean code practices, even if functionally correct—always apply critical thinking and professional judgment.

## 2.2 Guidelines for contributors

Our team adopts a coding style aligned with the AOS project standards, emphasizing **clarity, consistency, and readability**. Key conventions include:

- **Braces** should be placed on the **same line** as the preceding code, not on a new line. Braces are **required around conditional statements**, except when both the condition and the body fit on a single line (in which case braces are optional).

- **Line length** should not exceed **100 characters**, with exceptions for import statements, long URLs, or command examples in comments to maintain usability.

- **Annotations** must appear **before other modifiers** for the same element. A simple marker annotation (e.g., `@Override`) may be on the same line as the declaration. When using multiple or parameterized annotations, list them **one per line in alphabetical order**.

- Treat **acronyms and abbreviations as words** in names to enhance readability (e.g., `getHttpResponse`, not `getHTTPResponse`).

- Use **`TODO:` comments** (uppercase, followed by a colon) for temporary or incomplete code, to flag areas needing improvement.

- For **test methods**, adopt a naming convention that uses **underscores to separate what's being tested from the specific test case** (e.g., `calculateTotal_withEmptyCart_returnsZero`) to improve clarity of test coverage.

Code Reviewing

Code reviews are a critical step in maintaining high-quality code, ensuring consistency with project standards, and enabling knowledge sharing across the team. All non-trivial changes must undergo review before being merged.

### When to Review Code

- **Mandatory** for all feature additions, bug fixes, and refactors exceeding 10 lines.
- **Recommended** for documentation updates if they introduce or change technical explanations or examples.
- **Optional** for minor comment fixes or trivial formatting adjustments.

### What to Look For

- **Correctness**: Does the code do what it claims? Are edge cases and failure modes handled?
- **Clarity**: Is the logic clear and easy to follow? Are variable and method names descriptive?
- **Consistency**: Does the code adhere to the project's coding standards (as defined above)?
- **Test coverage**: Are there sufficient unit and/or integration tests? Are test cases meaningful?
- **Maintainability**: Is the code modular, DRY (Don't Repeat Yourself), and well-documented?

### Review Best Practices

- **Be constructive**: Focus on improving the code, not criticizing the author.
- **Ask questions**: If something is unclear, request clarification rather than assuming.
- **Use inline comments sparingly**: Favor summarized suggestions over excessive line-by-line nitpicks.
- **Respect context**: Evaluate changes in the broader context of the system, not just in isolation.

### Using AI Tools

- **AI-assisted review tools** may be used to identify style issues, potential bugs, or refactoring opportunities. Examples include static analysis plugins or AI code reviewers.
- AI tools are **not a substitute** for human judgment. They should **augment**, not replace, the review process.
- Reviewers are encouraged to verify any AI-generated feedback before incorporating it into final comments or suggestions.

### Turnaround Expectations

**Within 24 hours**: Aim to respond to review requests promptly.

**Communicate delays**: If review cannot be completed within a reasonable time, inform the requester and suggest alternatives.

## 2.3 Code quality metrics and dashboards

To ensure high code quality throughout the development lifecycle, the team relies on a comprehensive set of automated metrics, static analysis tools, and dashboards integrated into the CI/CD pipeline. These practices provide continuous visibility over code health, support early detection of issues, and enforce quality standards at every stage of delivery.

### Tracked Metrics

The team monitors several key code quality metrics using **SonarCloud**, integrated with **JaCoCo** for code coverage analysis and **GitHub Actions** for automation. The most relevant metrics include:

- **Code coverage**, which is measured using JaCoCo during CI workflows and reported to SonarCloud. A minimum threshold of 80% coverage is enforced on new code.
- **Cyclomatic complexity**, used to identify overly complex logic that may hinder maintainability or testability.
- **Code duplication**, automatically detected by SonarCloud, which flags any instances exceeding 3% duplication in newly introduced code.
- **Technical debt**, estimated based on the time required to address quality issues. This metric guides refactoring priorities and informs decision-making.
- **Bugs, vulnerabilities, and code smells**, which are categorized by SonarCloud based on severity and impact. The team prioritizes resolving critical or blocker-level findings immediately.

### Defined Quality Gates

A strict **quality gate** is configured in SonarCloud to ensure that only high-quality code can be merged into the main branch. The enforced criteria include:

- At least **80% test coverage** on new code.
- **Zero critical bugs or vulnerabilities**.
- **No more than 3% duplication** in newly added code.
- **No blocker or critical code smells**.

These thresholds are designed to maintain the existing quality level and promote incremental improvement, while minimizing the risk of regressions or technical debt accumulation.

#### Dashboards and Visibility

All relevant metrics are visualized through automated dashboards provided by SonarCloud, which are accessible to the development team and integrated directly into the GitHub pull request interface. This setup provides immediate feedback on code quality during the review process.

In addition, **Jira dashboards** are used to combine testing progress data (from Xray) with coverage and static analysis insights, offering a comprehensive view of both functional and technical quality. Pull requests are configured to include automatic reports from SonarCloud, summarizing issues and test coverage. If a pull request does not meet the defined quality gate, it is automatically blocked from being merged.

Weekly review meetings include a segment dedicated to analyzing trends in code quality metrics, discussing areas of concern, and planning technical improvements. This regular review process helps the team stay aligned on quality goals and proactively address potential issues before they escalate.

# 3 Continuous delivery pipeline (CI/CD)

## 3.1 Development workflow

### 3.1.1 Coding workflow

Our development process follows a streamlined and collaborative workflow based on **GitHub Flow**, adapted to include main and develop branches for stability and feature separation. When a developer begins work, they select a **user story** from the product backlog, typically managed by Jira.

The developer creates a **feature branch** from develop, named according to a standard convention (e.g., feature/US123-add-login), and begins implementation locally. This branch maps directly to a specific user story or task.

Once the feature is implemented, the developer pushes the branch to the remote repository and opens a **pull request (PR)** against the develop branch. This PR must be reviewed and approved by at least one teammate before it can be merged. Reviews are conducted directly within GitHub, using the built-in PR review tools.

This workflow ensures that:

- Code changes are isolated and traceable.
- All work undergoes **peer review**.
- Continuous integration (CI) checks validate each change before integration.

For production releases, the develop branch is merged into main, which then triggers a deployment pipeline. This dual-branch strategy provides a balance between agility in development and stability in releases.

### 3.1.2 Code Review and Definition of done

All code changes must go through **pull requests (PRs)** and be reviewed by other team members before being merged. The review process ensures code quality, functional correctness, test coverage, and compliance with the project's coding standards.

We enforce strict review requirements based on the target branch:

- **For pull requests into develop**, **a minimum of 2 approvals** are required.
- **For pull requests into main**, **at least 3 approvals** are mandatory before merging.

This policy ensures that features integrated into the development branch are well-validated, and production-level changes meet the highest quality standards.

To streamline and standardize the process, we use **pull request and issue templates**. These templates help contributors:

- Clearly describe the scope, motivation, and implementation details of a pull request.
- Link the PR to the relevant user stories or issues.
- Follow consistent formatting and documentation practices.
- Ensure key checklist items (such as testing, documentation, and changelog updates) are completed before review.

The team's **Definition of Done** for a user story includes:

➔ All acceptance criteria are fully implemented and tested.

➔ Code is clean, maintainable, and follows the project's conventions.

➔ Appropriate unit, integration, and/or behavior tests are written and pass successfully.

➔ CI workflows (build, test, static analysis, etc.) pass without errors.

➔ Pull request is reviewed and approved (2 for develop, 3 for main).

➔ Code is merged into the appropriate branch (develop for ongoing work, main for releases).

➔ The feature is deployed (if applicable) and functioning in the target environment.

➔ Documentation is updated where relevant (README, API docs, inline comments, etc.).

This structured review and validation process helps maintain a high standard of code quality and team accountability throughout the development lifecycle.

## 3.2 CI/CD pipeline and tools

Our CI/CD pipeline is built using **GitHub Actions**, with several workflows designed for different stages of the development lifecycle.

### 3.2.1 Continuous Integration

The main CI workflows include:

- **build.yml**: Runs on pull requests to main or develop. It performs:
  - Checkout of the repository.
  - JDK 21 setup using Temurin distribution.
  - Caching of Maven and SonarCloud dependencies to improve speed.
  - Build and verification of the project using Maven.
  - Execution of unit tests and generation of code coverage reports via **JaCoCo**.
  - Static analysis with **SonarCloud**, which enforces quality gates and highlights code issues.
- **maven_tests.yml**: Executes on pushes and pull requests. It runs:
  - A clean Maven build and test phase using the Backend/pom.xml.
  - Caching of Maven dependencies for optimization.
- **cucumber_tests.yml**: Specifically runs **Cucumber-based integration tests**, verifying system behavior against user-oriented scenarios.

These workflows ensure that every change is automatically built, tested, and analyzed before integration, significantly reducing the risk of bugs entering the main codebase.

### 3.2.2 Continuous Delivery

The **deploy.yml** workflow automates the release and deployment process. It is triggered by:

- Pushes to main or develop.
- Pull requests targeting these branches.
- Manual invocation via workflow_dispatch.

It is composed of the following stages:

1. **Acceptance Tests**:
   - Sets up Firefox and Docker environment.
   - Runs the application using Docker Compose for testing.

   - Validates functionality through automated browser or integration tests.

2. **Docker Build and Push**:
   - Builds Docker images for the application.
   - Pushes them to **GitHub Container Registry (GHCR)** using docker compose.
3. **Deployment**:
   - Runs on a **self-hosted runner** (e.g., a UA VM).
   - Pulls the latest container images.
   - Deploys the application using docker compose up -d.
   - Cleans up unused containers, images, and builders to maintain a clean environment.

This CD process ensures reliable and consistent deployment using **containerized environments**, enabling fast recovery, version control, and reproducibility across environments.

## 3.3 System observability

To maintain high availability and performance, the system incorporates basic observability mechanisms:

- **Logging**: Application and service logs are accessible via Docker and system logs. They can be integrated with centralized logging platforms (e.g., ELK Stack) if needed.
- **Health Checks**: Docker services expose health endpoints to monitor service availability and responsiveness.
- **Performance Testing**:
   - A dedicated workflow (performance.yml) is configured for **load testing using K6**.
   - While currently disabled, it is ready to be enabled to simulate real-world traffic and analyze system performance under stress.
- **Resource Monitoring and Cleanup**:
   - During deployments, the system automatically prunes unused containers, images, and Docker builders.
   - This helps avoid memory bloat and ensures efficient resource usage.
- **Future Enhancements**:

    - The system is designed to support integration with alerting tools (e.g., Prometheus + Alertmanager) for real-time alerts based on service health or resource thresholds.

In summary, this CI/CD pipeline ensures that code changes are automatically validated, tested, and deployed through clearly defined workflows. It supports fast, reliable delivery while maintaining code quality, reducing manual overhead, and increasing team productivity.

## 3.4 Artifacts repository

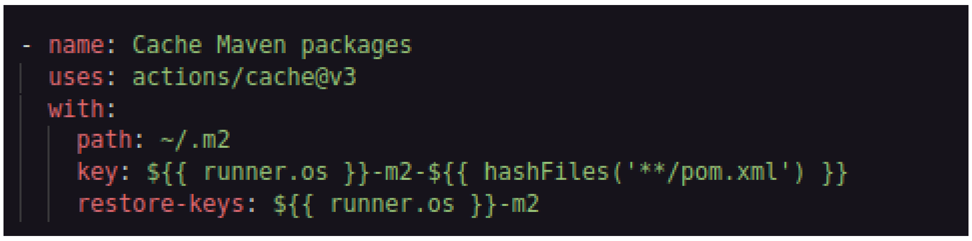
Our project leverages artifact repositories to manage and store build artifacts in a consistent and centralized way. This supports reproducibility, caching efficiency, and reliable deployments.

### Maven Artifacts

For Java dependency management, we use **Apache Maven**. While external dependencies are fetched from public repositories like Maven Central, we implement **local caching** of Maven packages using GitHub Actions' cache mechanism. This significantly speeds up the CI builds and reduces dependency download overhead.

The Maven cache configuration:

- Stores `.m2` directory content based on the hash of the project's `pom.xml` files.
- Ensures dependencies are reused across builds unless the dependency tree changes.
- Avoids redundant downloads during CI runs.



```yaml
- name: Cache Maven packages
  uses: actions/cache@v3
  with:
    path: ~/.m2
    key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
    restore-keys: ${{ runner.os }}-m2
```

This local caching practice enhances CI performance and reduces external dependency on Maven Central during automated workflows.

### Docker Images and GitHub Container Registry

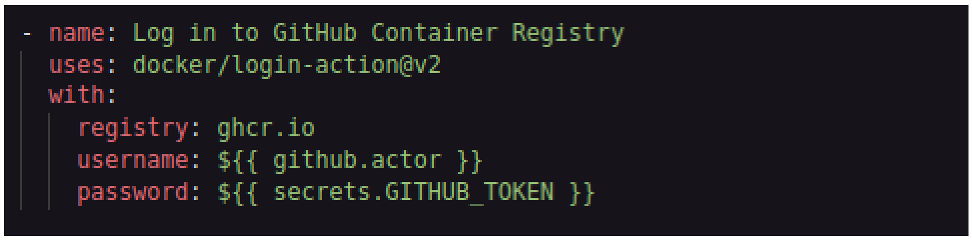
In the Docker Compose setup, we also define a dedicated **maven-build service** using the official maven:3.9-eclipse-temurin-21 image to perform offline or containerized builds. This allows for reproducible Maven packaging in isolated environments, enhancing consistency between local and CI builds.

This containerized Maven builder ensures environment parity and acts as an internal artifact producer within the Compose network.

Our project uses **GitHub Container Registry (GHCR)** as the centralized repository for Docker images. These are:

- Built via CI/CD workflows using docker compose build.

- Tagged and pushed with the latest version or Git commit identifiers.
- Pulled by deployment jobs or used locally for development and testing.



```yaml
- name: Log in to GitHub Container Registry
  uses: docker/login-action@v2
  with:
    registry: ghcr.io
    username: ${{ github.actor }}
    password: ${{ secrets.GITHUB_TOKEN }}
```

Docker images include:

- A **backend** service, built from Dockerfile.backend in the Backend folder.
- A **frontend** service, built from Dockerfile.frontend in the frontend folder.

Both services are pushed to GHCR under the organization or user namespace (ghcr.io/${GITHUB_REPOSITORY_OWNER}/...). This setup supports:

- Version-controlled, containerized builds.
- Lightweight rollbacks and promotions across environments.
- Secure and scoped access control via GitHub tokens.

In summary, by combining **Maven caching**, **containerized local builds**, and **GitHub Container Registry**, our artifact management strategy provides high-performance builds, consistency across environments, and a robust foundation for continuous delivery.

## 4 Software testing

### 4.1 Overall testing strategy

The testing strategy adopted for this project follows a hybrid approach, combining multiple testing methodologies to ensure comprehensive verification of both functional and non-functional requirements. This approach integrates unit testing, behavior-driven development (BDD), integration testing with containerized dependencies, and performance testing, all supported by continuous integration pipelines.

The development team implemented **Test-Driven Development (TDD)** for core business logic, leveraging the JUnit framework in conjunction with Mockito. This practice encouraged early specification of expected behaviors, resulting in modular, well-tested code.

In parallel, **Behavior-Driven Development (BDD)** was adopted to ensure alignment between implemented features and business requirements. Acceptance criteria were defined using the **Gherkin DSL** and automated with the **Cucumber** framework. This enabled clear traceability between user stories and executable test cases, and enhanced communication across team roles.

A combination of the following tools was employed to support the overall testing strategy:

- **JUnit**: Utilized for writing unit and integration tests.
- **Mockito**: Employed to mock dependencies and isolate components during unit testing.
- **Cucumber with Gherkin**: Used to define and automate BDD scenarios.
- **REST-Assured**: Applied for black-box testing of REST APIs.
- **Testcontainers**: Enabled integration tests to run against real containerized instances of external services (e.g., databases), improving realism and reproducibility.
- **JaCoCo**: Integrated to measure and report test coverage metrics.
- **k6**: Used to develop and execute HTTP-based performance tests to validate Service Level Objectives (SLOs).
- **Xray (Jira Test Management Plugin)**: Integrated with Jira to manage test cases, link them to user stories and acceptance criteria, and track test execution and coverage across sprints.

The entire testing process is tightly integrated into the project's **Continuous Integration (CI)** pipeline. All tests are executed automatically on every push and pull request. Pull request merges are blocked if any test fails or if predefined quality thresholds are not met. The integration of **Xray with Jira** supports structured test planning and traceability by linking automated and manual tests to corresponding user stories. This ensures that all acceptance criteria are validated and provides visibility over test coverage within the agile project management environment.

Static code analysis and code coverage are enforced using **SonarQube**, where quality gates are configured to ensure:

- Minimum code coverage is maintained on new code.
- No critical bugs or vulnerabilities are introduced.
- Code duplication and complexity are kept within acceptable limits.

Performance tests created with **k6** are executed in dedicated CI stages to monitor application responsiveness and load handling against defined SLOs. The results of these tests are used to inform optimization efforts and ensure scalability.

Testing outcomes are continuously monitored through dashboards integrated with the CI system. Code quality metrics, test pass rates, and coverage data are reviewed regularly to identify areas requiring improvement. Automated reports from tools like SonarQube provide ongoing visibility into the health of the codebase, supporting informed decision-making throughout the development lifecycle.

### 4.2 Functional testing and ATDD

To ensure consistent validation of user-facing functionality, the team adopted a process centered on **Acceptance Test-Driven Development (ATDD)** supported by **Behavior-Driven Development (BDD)** practices. This process ensures that every implemented feature is backed by clearly defined, traceable, and automated functional tests.

For each user story included in the project backlog, the development team is responsible for:

- Defining Gherkin-based scenarios that describe the expected behavior of the system from a user perspective.
- Implementing corresponding automated step definitions in Java using the Cucumber framework, interacting with the system under test through REST-Assured to validate actual behavior against expected outcomes.

Functional tests are written in a black-box manner, focusing strictly on observable inputs and outputs, without reference to the internal structure of the system.
All Gherkin scenarios are linked to their respective user stories through the integration of Xray with Jira. This allows the team to:

- Maintain traceability between requirements and test coverage.
- Monitor which user stories have associated automated validation.
- Support formal acceptance testing workflows and reporting.

The use of Xray also facilitates assessment of functional test coverage across the project and aids in identifying untested requirements.
The functional test suite is fully integrated into the project's Continuous Integration (CI) pipeline. The pipeline is configured to:

- Execute all Cucumber-based functional tests automatically on every push and pull request.
- Enforce test pass/fail status as a condition for merging into the main development branch, via branch protection rules.
- Provide immediate feedback on test failures to the development team.

This automation ensures that only features that satisfy their functional acceptance criteria, as validated by the associated tests, are allowed to be integrated into the codebase.
To assure adherence to this process, the following practices are enforced:

- Writing and automating Gherkin scenarios is considered part of the definition of done for each user story.
- Peer code reviews are used to verify that the test scenarios are meaningful, complete, and accurately reflect the story's acceptance criteria.

- CI feedback is monitored continuously to detect regressions and maintain a stable, test-verified codebase.

### 4.3 Developer facing tests (unit, integration)

In this project, developer-facing tests form a key component of the quality assurance strategy, focusing on the internal correctness, stability, and reliability of the codebase. These tests are designed and executed from the developer's perspective, using open-box techniques to validate both individual components and their integration within the system.

Unit testing is a mandatory practice for all non-trivial business logic, particularly within service classes, utility methods, and controller logic. Developers are expected to implement unit tests alongside or prior to feature implementation, following a test-driven development (TDD) approach where feasible. Unit tests are written using JUnit in combination with Mockito, allowing for the isolation of components by mocking dependencies. These tests verify functional correctness at the method or class level, ensuring that specific inputs produce the expected outputs under various conditions. Key areas covered include slot booking validation, charging session calculations, and internal payment logic. Code coverage is tracked using JaCoCo and enforced through quality gates configured in SonarQube, ensuring that new code meets predefined test coverage thresholds before it can be merged into the main branch.

In addition to unit tests, integration tests are used to validate the correct interaction between multiple components, particularly those involving the persistence layer or external services. These tests are developed using Spring Boot's testing framework, with the aid of Testcontainers to provide real containerized instances of external dependencies, such as PostgreSQL databases. Integration tests are required whenever code relies on real data access, involves multiple service layers, or is expected to function across component boundaries. These tests help verify the stability and consistency of data flow, configuration, and inter-service communication under realistic conditions.

API testing complements both unit and integration testing by validating the behavior of the system's REST endpoints. Tests focus on ensuring that the application correctly handles HTTP requests, input validation, authentication, and response formatting. For end-to-end HTTP validation, REST-Assured is applied to interact with deployed endpoints in a black-box style. These tests cover both successful and erroneous scenarios, helping to ensure that APIs remain consistent, secure, and reliable across development iterations.

All developer-facing tests—unit, integration, and API—are executed automatically as part of the continuous integration (CI) process. Failures in any of these test categories result in blocked merges and are surfaced immediately to the development team for resolution. This integration ensures that internal system quality is continuously monitored and maintained, contributing to a stable and predictable software delivery lifecycle.

### 4.4 Exploratory testing

In addition to scripted and automated tests, the team adopted a lightweight **exploratory testing** strategy to identify unexpected behaviors, usability issues, and edge cases that may not have been fully covered by functional or developer-facing tests.

Exploratory testing sessions are conducted **periodically during development and prior to iteration reviews**, typically after new user stories are integrated and deployed to a test or staging environment.

These sessions are carried out manually by developers and the QA engineer, simulating real-world user actions such as booking a charging slot, modifying station availability, reviewing usage history, and performing payment actions.

The primary goals of exploratory testing are to:

- Detect **usability issues**, particularly in edge cases not covered by automated scenarios.
- Identify **inconsistencies in system behavior** when features interact (e.g., booking logic after a station status update).
- Validate assumptions and uncover **defects not anticipated** during test case design.

To structure these sessions without fully scripting them, team members used **informal charters** based on user stories, such as:

- "Explore booking behavior when multiple slots overlap."
- "Explore how the system responds when a station becomes unavailable after booking."
- "Explore data consistency in the charging history dashboard."

Findings from exploratory sessions are documented as issues or improvement suggestions in the team's backlog (Jira), and in some cases, led to the creation of additional automated test scenarios to cover uncovered behaviors.

Although exploratory testing was not formalized with dedicated tooling, it served as a complementary practice to increase confidence in system stability and to support a user-focused perspective during development.

## 4.5 Non-function and architecture attributes testing

As part of our quality assurance strategy, the project will include dedicated practices to evaluate key non-functional attributes, particularly system performance, responsiveness, and frontend quality. These aspects are essential to ensure that the application not only functions correctly but also performs reliably under expected usage conditions and delivers a high-quality user experience.

Performance testing will be conducted based on defined Service Level Objectives (SLOs), which establish clear expectations for system response times, throughput, and scalability. To validate these objectives, we will develop HTTP-based load test scripts using the **k6** framework. These tests will simulate realistic usage scenarios—such as concurrent bookings, API calls under load, or large-volume data retrieval—and will be integrated into the CI pipeline to enable automated performance validation during development. Thresholds will be defined to detect regressions or bottlenecks early in the lifecycle.

In cases where the application includes a frontend web interface, we will also apply automated audits using **Google Lighthouse**. This tool will allow us to evaluate critical quality attributes of the user interface, including performance metrics such as first contentful paint and time to interactive, as well as accessibility, adherence to web development best practices, and mobile usability. Lighthouse results will help guide frontend optimizations and ensure the application meets acceptable standards for end-user experience.

By integrating both backend and frontend non-functional testing into our development and delivery processes, we aim to continuously monitor and improve architectural robustness, system efficiency, and usability throughout the project lifecycle.