EVSYNC: Quality Assurance manual

Bernardo Borges [1035923], Gonçalo Monteiro [107758], Miguel Pinto [108481], Filipe Oliveira [114640]

Contents

1 Project management

1.1 Assigned roles

Bernardo Borges (103592) – Team Coordinator + Developer
Gonçalo Monteiro (107758) – Product Owner + Developer
Filipe Oliveira (114640) – DevOps Master + Developer
Miguel Pinto (108481) – QA Engineer + Developer

1.2 Backlog grooming and progress monitoring

As requested by the professor, we have set up a JIRA backlog directly connected to our Git repository. From here, we can create and manage branches, as well as identify commits using special tags. We can also see PRs made in the repository.
To track the progress made, we are utilizing both work item count as well as Story Points. We can also see how the project develops using burndown charts and flux diagrams, all located within Jira itself.
Regarding requirement-level coverage, we are using XRAY as our test management tool. It's also connected to JIRA and we can properly set it up to see whether tests are successfully completed in each commit.

2 Code quality management

2.1 Team policy for the use of generative AI

Our team embraces the use of generative AI tools (e.g., GitHub Copilot, ChatGPT) as productivity boosters, especially during development and documentation. However, we apply a strict policy to ensure responsible and effective usage:
Do's

- Use AI tools to speed up boilerplate code writing, documentation drafting, and generate testing scaffolds.
- Ask AI to help understand libraries or generate initial versions of functions/components.
- Review all AI-generated code manually and ensure it complies with our coding standards and security practices.

Don'ts:

- Don't commit AI-generated code without reviewing and understanding it.
- Don't use AI to generate entire features without team review and validation.
- Don't use AI tools to write tests or documentation for code you didn't fully read or understand.

The final responsibility for any AI-assisted code lies with the developer who commits it. If errors, bugs, or vulnerabilities are introduced due to AI-generated content, the blame is assigned to the developer who pushed the changes, not the tool.

2.2 Guidelines for contributors

Coding style

Our team follows the **AOSP Java Style Guide** for all Java development. This choice ensures consistency with industry standards in Android and backend Java development.

- Indentation: 4 spaces (8 spaces for line unions).
- Braces: Opening braces on the same line.
- Line length: Max 100 characters.
- Imports: Organized and fully qualified when needed; wildcard imports are avoided.

Code reviewing

Code reviews are mandatory before any changes are merged into the main or development branches. These reviews serve as quality gates and promote shared code ownership.

- At least one peer review is required before merging.
- Reviews are done via PRs in GitHub.
- Developers must provide clear PR descriptions, including context and linked JIRA tickets.

AI tools (e.g., GitHub Copilot, ChatGPT) may assist in proposing improvements or identifying issues, but final validation must be manual. Responsibility for reviewed code lies with both the author and reviewer.

2.3 Code quality metrics and dashboards

To maintain high code quality, our team uses a combination of JaCoCo and SonarCloud, both integrated into our CI pipeline.
JaCoCo generates coverage reports on every push or PR. These reports help ensure that new code contributions are adequately tested. Meanwhile, SonarCloud performs static analysis on each commit, checking for maintainability, code smells, bugs, and vulnerabilities.
We've defined quality gates in SonarCloud that must be satisfied before a PR can be merged:

- Overall project coverage must be at least 80%
- No critical or blocker-level issues can be present

In addition, our pipeline follows a fail-fast approach: any test failure—unit, integration, or functional—will cause the pipeline to fail and display the issue directly on the GitHub PR. SonarCloud dashboards are linked to each PR and give a real-time view of:

- Code coverage evolution
- Technical debt and code duplication
- Ongoing trends in code quality

These metrics are used both during reviews and sprint planning to help guide technical decisions and uphold quality standards.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

Our team follows a streamlined development workflow using two primary branches: dev for ongoing development and main for production-ready code. This approach avoids the overhead of multiple feature branches while maintaining a clear separation between in-progress work and stable releases.
When a developer begins work, they select a JIRA ticket from the backlog. Tasks are clearly linked to user stories or bug reports and include acceptance criteria and relevant context. All code changes are committed directly to the dev branch. Each commit is associated with the corresponding JIRA issue through standardized commit messages, which improves traceability and project visibility. Developers push changes frequently to ensure early feedback from CI and code reviewers. PRs are created from dev into main only when a feature, fix, or release milestone is ready. These PRs must include:

- A summary of the changes
- Links to related JIRA issues
- Any relevant screenshots or notes for the reviewer

Before a PR from dev to main is merged, it must pass all automated tests and quality checks (via CI and SonarCloud) and be approved by at least one reviewer. Merging into main is considered a deployment event and triggers the delivery pipeline.
This approach keeps the workflow lightweight while ensuring that the main branch always reflects a stable and deployable state.

Definition of done

A user story or task is only considered “done” when it satisfies all the following criteria, ensuring that the work is complete, tested, reviewed, and ready for deployment:

- All acceptance criteria defined in the JIRA ticket are met.
- Code is pushed and committed to the dev branch.
- Automated tests are implemented and pass successfully.
- The CI pipeline passes with no errors.
- The code has been reviewed and approved.
- No critical or blocker issues are reported by SonarCloud.
- Documentation is updated if applicable.

Only when all of these conditions are satisfied can the task be marked as “done” in JIRA and considered for deployment to main.

3.2 CI/CD pipeline and tools

We use GitHub Actions to manage our CI/CD process, which runs automatically on every push to the dev branch.

Continuous Integration (CI)

The CI pipeline handles:

- Building the project
- Running unit and integration tests
- Analyzing code with SonarCloud
- Generating test coverage reports via JaCoCo

If any test fails or if SonarCloud's quality gates are not met, the pipeline fails, and the issue is shown directly in the PR.

Code Review and Merge Policy

Once the tests are completed and the SonarCloud report metrics are passed, the PR can be reviewed by someone else that isn't the creator of the PR. Once that review is completed, the merge can be continued.

Continuous Deployment (CD)

The CD pipeline is triggered on push to either the dev or main branch and handles deployment based on the target branch:

- dev branch:
- Deploys to the staging environment.
- Initializes the Docker environment.
- Runs smoke tests to validate basic functionality.
- The staging VM should be used for additional manual or automated testing.
- main branch:
- Deploys to the production environment.
- Uses the Docker Compose configuration found in the project root directory for deployment.

Secrets Management

All environment secrets and sensitive configurations are securely managed via GitHub Secrets, ensuring safe and isolated deployments across staging and production.

3.3 System observability

Our system leverages Prometheus to collect and store real-time metrics on application performance, resource usage, and operational health. Prometheus scrapes predefined endpoints to gather key data such as request rates, error counts, and latency measurements.
These metrics are visualized through Grafana dashboards, offering the team a centralized and accessible view of system behavior over time. Grafana's alerting capabilities enable proactive notifications when critical indicators—like error rates or response times—exceed predefined thresholds.
Together, this observability setup empowers the team to monitor system health proactively, quickly detect issues, and maintain operational reliability as the project scales.

4 Software testing

4.1 Overall testing strategy

Our testing strategy combines Test-Driven Development (TDD) and Behavior-Driven Development (BDD) to ensure code quality and alignment with business requirements from the earliest stages of development.
We use Cucumber to define user-facing scenarios, allowing functional tests to be expressed in a clear, human-readable format.
Tests are developed alongside code and run both locally and automatically via the CI pipeline on each PR to the dev branch. This ensures that regressions are caught early, and that each commit maintains system integrity.
We use JaCoCo and SonarCloud to monitor code coverage and enforce minimum thresholds as quality gates for PRs.
Additionally, test results and coverage are integrated into XRAY, allowing visibility into test execution and requirement-level traceability within JIRA.

4.2 Functional testing and ATDD

Functional tests in the project follow an Acceptance Test-Driven Development (ATDD) approach and are written using Cucumber. These tests are expressed in Gherkin and describe the system's expected behavior from the user's perspective, making them easy to understand and review.
Developers are expected to write functional test scenarios before or alongside the development of a feature, using the acceptance criteria defined in the related JIRA issue. These scenarios act both as executable specifications and as validation that the implemented functionality meets stakeholder expectations.
The functional tests are:

- Integrated into the CI pipeline and automatically executed on every push
- Tracked in XRAY, providing visibility into which features are covered and whether they pass

By tying tests directly to JIRA requirements, the team ensures that functional coverage is not only enforced technically but also traceable and reviewable at the requirement level.

4.3 Developer facing tests (unit, integration)

Our development process emphasizes thorough internal validation through both unit and integration tests.

Unit Tests
We use JUnit to write unit tests that verify individual components in isolation. These are created following a TDD approach—developers write tests before or during implementation to ensure correctness from the ground up. Unit tests run locally during development and automatically within the CI pipeline, helping catch regressions early.

Integration Tests
Integration tests are written to validate the interaction between multiple components or external systems. These tests verify the correct behavior of the system as a whole when various parts are combined. They are also included in the CI workflow, providing an added layer of verification during each push.

Both types of tests are tracked using JaCoCo for code coverage, and their results are evaluated by SonarCloud, which enforces quality gates. The testing strategy ensures that both isolated logic and component interactions are validated continuously.

4.4 Exploratory testing

Alongside automated tests, we incorporate **exploratory testing** as part of our quality assurance process. This manual testing approach allows team members—particularly the QA Engineer—to interact freely with the application, exploring different user flows and edge cases without predefined scripts.
Exploratory testing sessions are typically conducted:

- After the implementation of major features
- Before milestone deliveries
- When bugs are reported that require reproduction in varying scenarios

This strategy helps identify usability issues, unexpected behaviors, or gaps not covered by automated tests.

4.5 Non-function and architecture attributes testing

To validate the system's behavior under stress and ensure it meets architectural expectations, we conduct non-functional testing with a focus on performance and operational stability.
We use k6 to perform load and stress testing, simulating real-world traffic patterns and high-concurrency scenarios. These tests help evaluate system responsiveness, throughput, and failure handling under pressure.
Test metrics are exported to Prometheus, and visualized in Grafana, where the team can monitor key indicators such as request rates, response times, and error rates in real-time. This integration allows for effective analysis and comparison of performance across different builds and deployments.