TQS: Quality Assurance manual

Diogo Costa[112714],Bruno Tavares[113372],Francisco Pinto[113763],André Alves[113962]
v2025-06-05

Contents

TQS: Quality Assurance manual 1
1 Project management 1
1.1 Assigned roles 1
1.2 Backlog grooming and progress monitoring 1
2 Code quality management 2
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
3 Continuous delivery pipeline (CI/CD) 2
3.1 Development workflow 2
3.2 CI/CD pipeline and tools 2
3.3 System observability 3
3.4 Artifacts repository [Optional] 3
4 Software testing 3
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

Francisco Pinto- Team Leader/Developer
Diogo Costa- Product Owner/Developer
Bruno Tavares- QA Engineer/Developer

André Andre- DevOps/Developer

1.2 Backlog grooming and progress monitoring

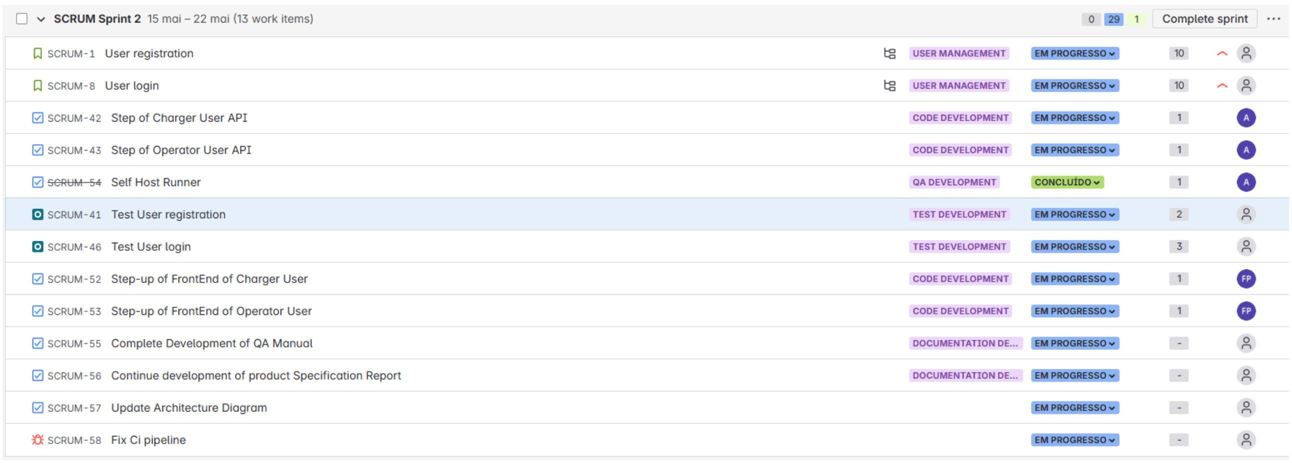
We used Jira as our backlog management tool. Issues were categorized as “User Story,” “Feature,” “Task,” “Problem,” or “Test” to make them easier to understand and manage. Additionally, each issue was assigned to an Epic, which helped group related items under a broader category.

We worked in weekly iterations (sprints). At the start of each sprint, the team manager selected the relevant issues for that sprint and placed them on the board. Each issue was then assigned a difficulty score (1–10) and a priority level. The team manager also divided the workload among the team members, including himself.

To simplify the development of user stories, each one was broken down into smaller sub-issues, making the development process more manageable and granular. Developers created branches in the version control system corresponding to each user story or feature. These branches could then have sub-branches for specific parts of the development process. All branches and sub-branches were tracked using the Jira issue IDs.

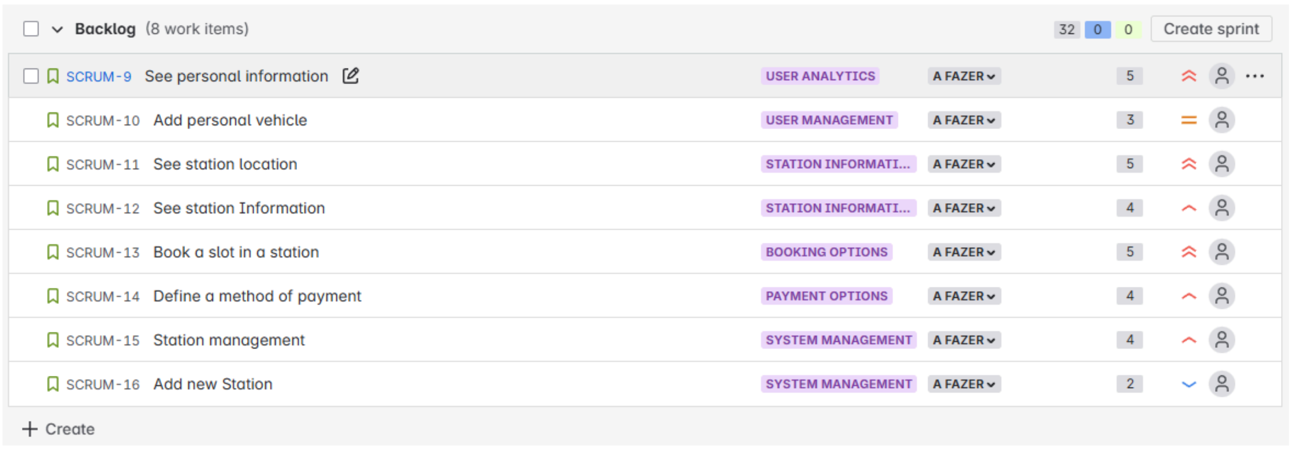
Once a task was completed—whether it involved coding, documentation updates, or the completion of sub-issues related to a user story—the developer marked the corresponding Jira issue as done.

Regarding testing, “Test” issues were always linked to a User Story or Feature. Similar to user stories, test issues could be broken down into multiple sub-issues, each addressing a specific aspect of testing.



SCRUM Sprint 2 15 mai – 22 mai (13 work items) 0 29 1 Complete sprint

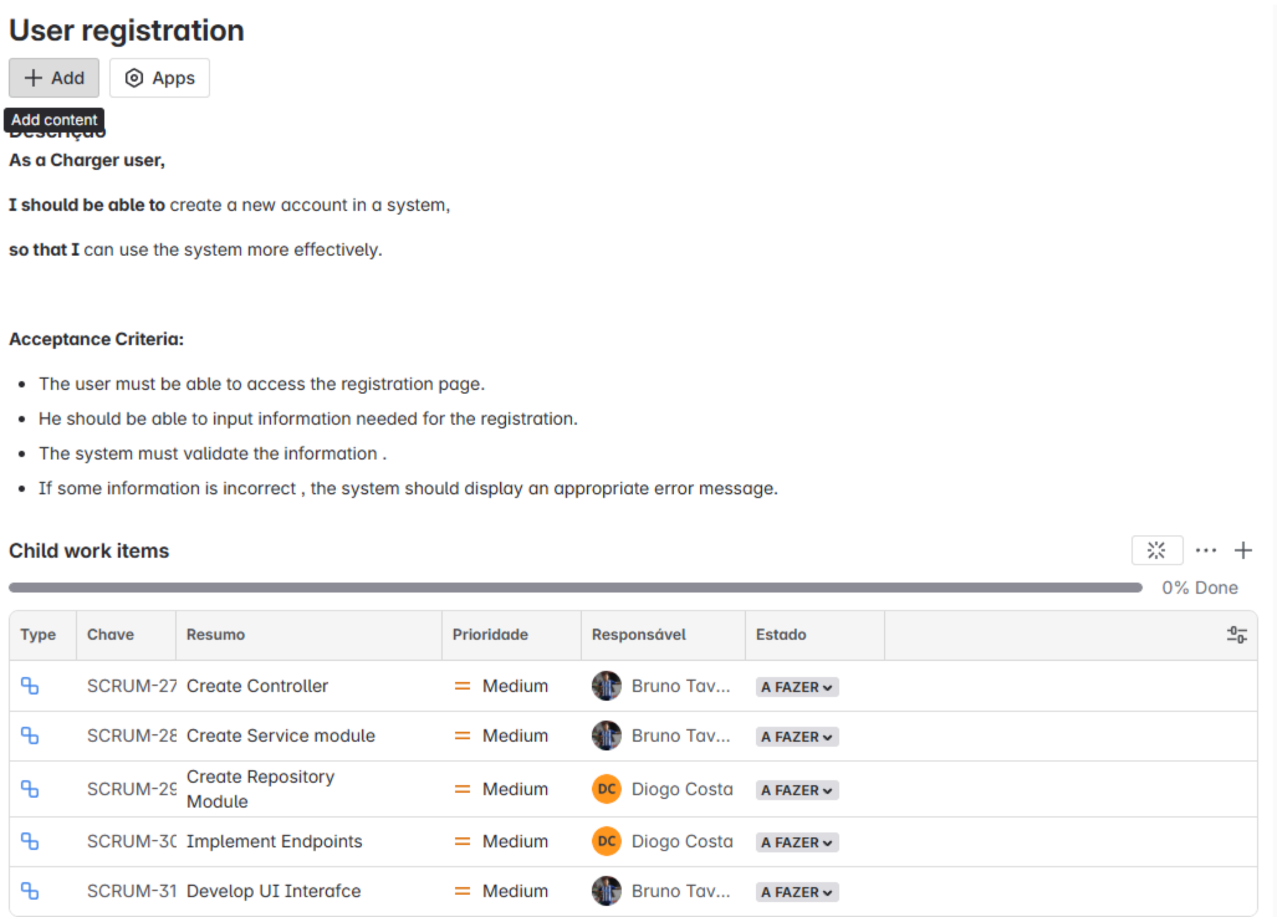
SCRUM-1	User registration	USER MANAGEMENT	EM PROGRESSO	10	
SCRUM-8	User login	USER MANAGEMENT	EM PROGRESSO	10	
SCRUM-42	Step of Charger User API	CODE DEVELOPMENT	EM PROGRESSO	1	A
SCRUM-43	Step of Operator User API	CODE DEVELOPMENT	EM PROGRESSO	1	A
~~SCRUM-54~~	Self Host Runner	QA DEVELOPMENT	CONCLUÍDO	1	A
SCRUM-41	Test User registration	TEST DEVELOPMENT	EM PROGRESSO	2	
SCRUM-46	Test User login	TEST DEVELOPMENT	EM PROGRESSO	3	
SCRUM-52	Step-up of FrontEnd of Charger User	CODE DEVELOPMENT	EM PROGRESSO	1	FP
SCRUM-53	Step-up of FrontEnd of Operator User	CODE DEVELOPMENT	EM PROGRESSO	1	FP
SCRUM-55	Complete Development of QA Manual	DOCUMENTATION DE...	EM PROGRESSO	-	
SCRUM-56	Continue development of product Specification Report	DOCUMENTATION DE...	EM PROGRESSO	-	
SCRUM-57	Update Architecture Diagram		EM PROGRESSO	-	
SCRUM-58	Fix Ci pipeline		EM PROGRESSO	-	



Backlog (8 work items) 32 0 0 Create sprint

SCRUM-9	See personal information	USER ANALYTICS	A FAZER	5
SCRUM-10	Add personal vehicle	USER MANAGEMENT	A FAZER	3
SCRUM-11	See station location	STATION INFORMATI...	A FAZER	5
SCRUM-12	See station Information	STATION INFORMATI...	A FAZER	4
SCRUM-13	Book a slot in a station	BOOKING OPTIONS	A FAZER	5
SCRUM-14	Define a method of payment	PAYMENT OPTIONS	A FAZER	4
SCRUM-15	Station management	SYSTEM MANAGEMENT	A FAZER	4
SCRUM-16	Add new Station	SYSTEM MANAGEMENT	A FAZER	2

+ Create



User registration

As a Charger user,

I should be able to create a new account in a system,

so that I can use the system more effectively.

Acceptance Criteria:

- The user must be able to access the registration page.
- He should be able to input information needed for the registration.
- The system must validate the information .
- If some information is incorrect , the system should display an appropriate error message.

Child work items

0% Done

Type	Chave	Resumo	Prioridade	Responsável	Estado
	SCRUM-27	Create Controller	= Medium	Bruno Tav...	A FAZER
	SCRUM-28	Create Service module	= Medium	Bruno Tav...	A FAZER
	SCRUM-29	Create Repository Module	= Medium	DC Diogo Costa	A FAZER
	SCRUM-30	Implement Endpoints	= Medium	DC Diogo Costa	A FAZER
	SCRUM-31	Develop UI Interafce	= Medium	Bruno Tav...	A FAZER

2 Code quality management

2.1 Team policy for the use of generative AI

Generative Ai is allowed for:

Code Review Assistance,use the AI to suggest improvements or detect possible bugs and anti-patterns and to summarize large pull requests to speed up understanding.

Unit Test Generation, generate unit tests for functions or methods. Help design edge cases and test inputs for individual components.

Documentation Drafting, use the AI to improve docstrings, README files, and internal documentation.

Learning and Troubleshooting, ask the AI to explain unfamiliar concepts, tools, or error messages. Get guidance on how to use a library or framework.

Generative Ai is not allowed for:

Integration or End-to-End Test Generation, do not use AI to write complex integration tests that involve multiple systems because these require context-aware, environment-specific logic that AI may misrepresent.
AI-Generated Architecture Decisions, avoid using AI as the sole basis for making architectural decisions or system design choices.
Code Deployment or CI/CD Scripts, avoid relying on AI to generate production-level deployment scripts.

2.2 Guidelines for contributors

Coding Style
We followed the Google Java Style Guidelines to ensure a consistent and standardized approach to code development across the team. This helped maintain code readability and facilitated collaboration among developers.

Code Reviewing
Code reviews were a crucial part of our development workflow, especially during merge requests. We enforced a policy requiring that at least two other developers, or one with the help of copilot ai ,review the code before it could be merged. The reviewers were responsible for verifying that the code adhered to the style guidelines and included sufficient test coverage.

2.3 Code quality metrics and dashboards

We integrated SonarQube with our GitHub repository to automate code quality checks through static analysis. SonarQube is an open-source platform that detects bugs, bad coding practices, and most importantly security vulnerabilities. It also performs continuous code analysis, allowing us to not only identify issues in the code but also trace when and where they were introduced.

To ensure high code quality, we enforced quality gates within SonarQube. These are predefined criteria that new code must meet before being accepted into the project. If a submission fails to meet the standards set by these gates, SonarQube flags the issue, and the code must be revised until it complies.

The quality gates we enforced included the following metrics:

- **Test Coverage** – Ensuring a minimum level of automated test coverage for new code.
- **Maintainability** – Checking for code smells and maintainability issues.
- **Reliability** – Identifying potential bugs that could lead to runtime errors.
- **Security** – Detecting vulnerabilities and weaknesses in the code.
- **Code Duplication** – Preventing redundant or duplicated code blocks.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

We used Github Flow methodology as a coding workflow for continuous delivery.

This involves the following steps:

1. **Create a branch**: A User Story or a Feature that will start development must have a new branch.

2. **Development of the feature:** The developer can create sub-branches for the feature if they think it is necessary to granularise the development process, in which case they have to merge it into the Feature branch, or if what they are doing is more simple they can commit directly into the branch, making sure the changes are well documented.

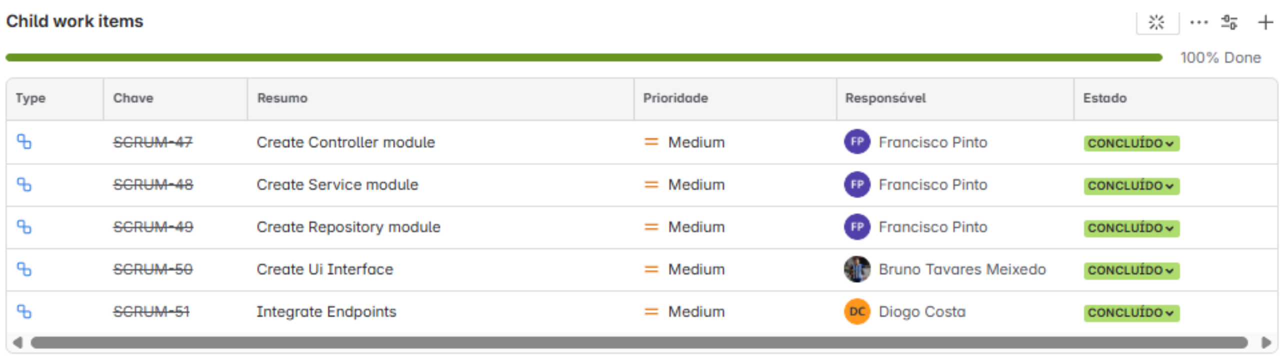
3. **Pull Request**: When the Feature or User Story is completed,the developer then opens a pull request into the “dev” branch. This request must be well documented and tested both manually and with automatic tests.

4. **Code Review**: After the pull request is made, the other team members must analyze the code, if they think it is alright, they can accept the request if not decline it.

5. **Deploy:** At the end of the sprint a release is made which includes the Features that were completed during it.This involves a merge of the “dev” branch into the “main” branch.

Definition of done:

Our branching strategy was closely aligned with our issue tracking in Jira. Each small task typically had its own dedicated branch for development. For Features or more complex User Stories that involved multiple sub-issues and testing, a main branch was created, along with sub-branches to allow developers to divide the development work more effectively. A User Story was only considered complete when all its sub-issues which represented the various components required for its implementation were finished. Once development was done, extensive testing was performed to verify that the system behaved as described in the user story. Only then was the feature considered ready to be merged into the dev branch. This structure ensured clear traceability between code and Jira issues, streamlined collaboration, and maintained high standards of code quality and feature completeness.



Child work items

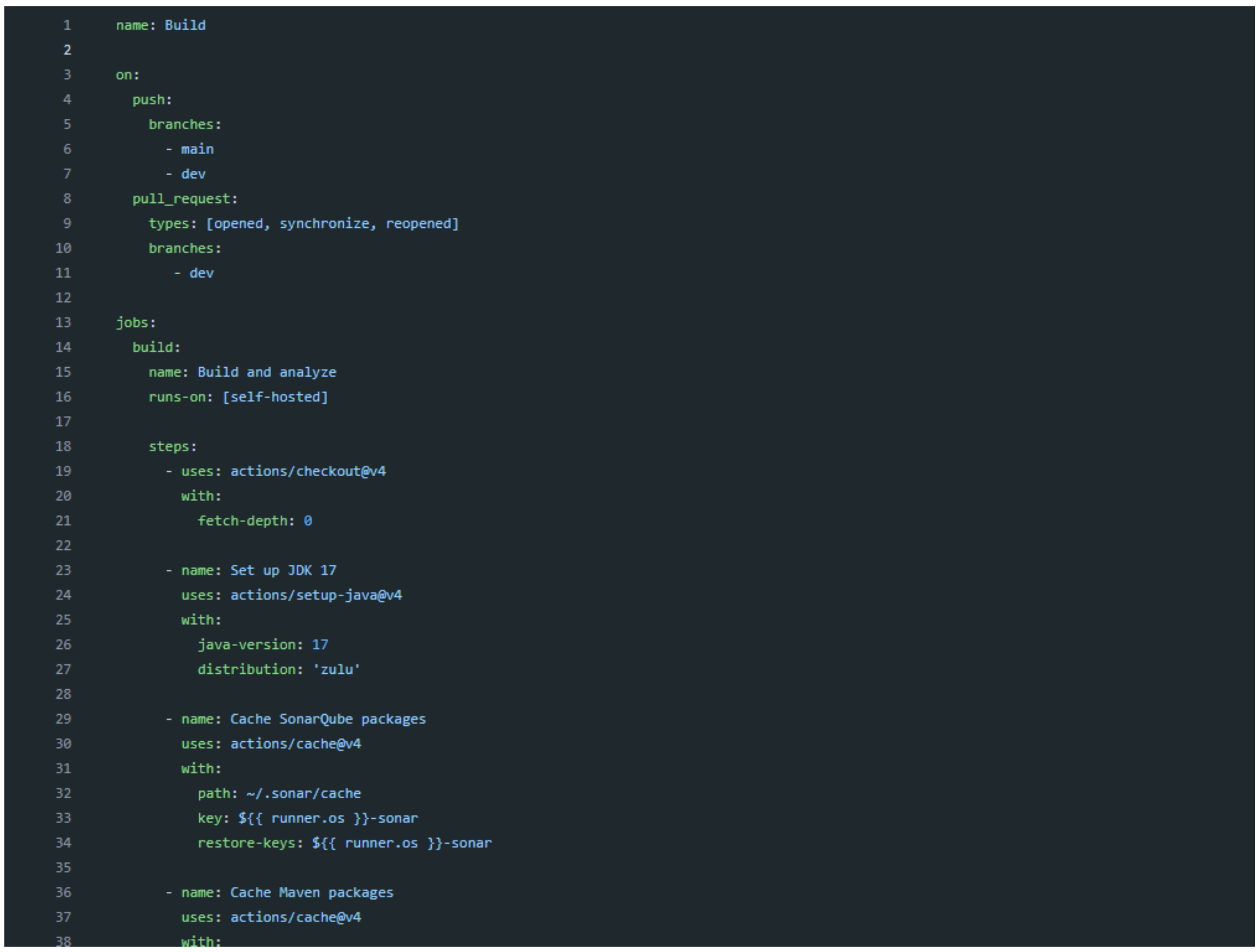
100% Done

Type	Chave	Resumo	Prioridade	Responsável	Estado
	~~SCRUM-47~~	Create Controller module	= Medium	FP Francisco Pinto	CONCLUÍDO
	~~SCRUM-48~~	Create Service module	= Medium	FP Francisco Pinto	CONCLUÍDO
	~~SCRUM-49~~	Create Repository module	= Medium	FP Francisco Pinto	CONCLUÍDO
	~~SCRUM-50~~	Create Ui Interface	= Medium	Bruno Tavares Meixedo	CONCLUÍDO
	~~SCRUM-51~~	Integrate Endpoints	= Medium	DC Diogo Costa	CONCLUÍDO

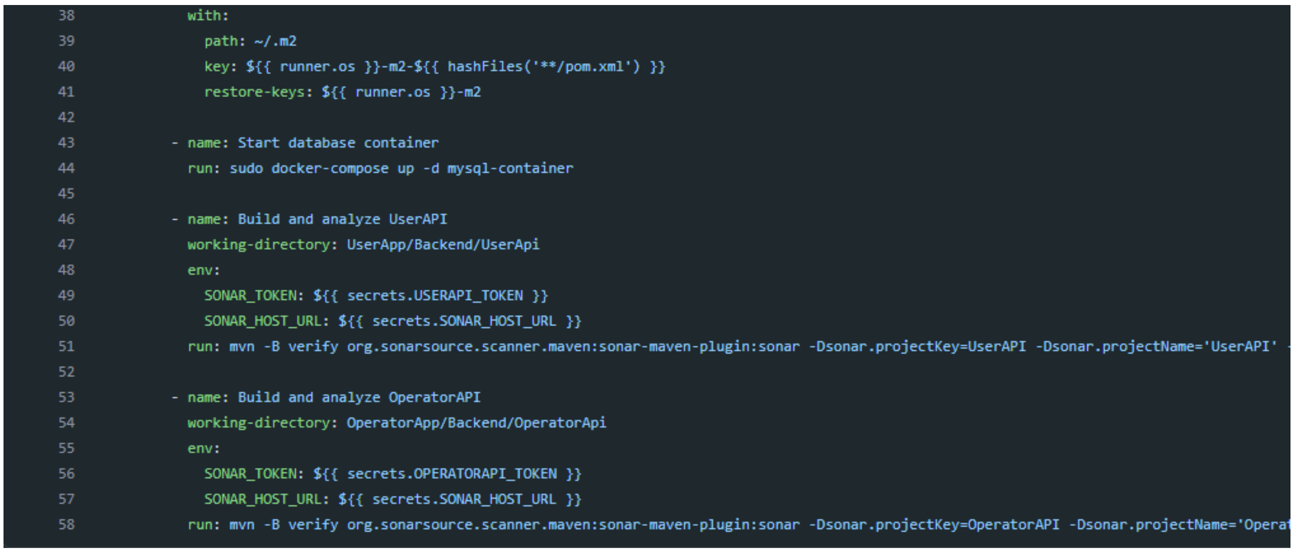
3.2 CI/CD pipeline and tools

Continuous Delivery is an essential practice in modern software development, enabling teams to efficiently, quickly, and reliably deliver code changes. In our project, this practice is supported by a variety of tools and practices, including code versioning and management through GitHub, regular test automation to ensure code stability and quality, and the use of GitHub Actions for automated deployment of the application in test and production environments.

Our project's continuous delivery pipeline is automatically triggered after changes in the "dev" branch, ensuring efficient and reliable delivery of new features and bug fixes to end users on our server. CI Pipeline:

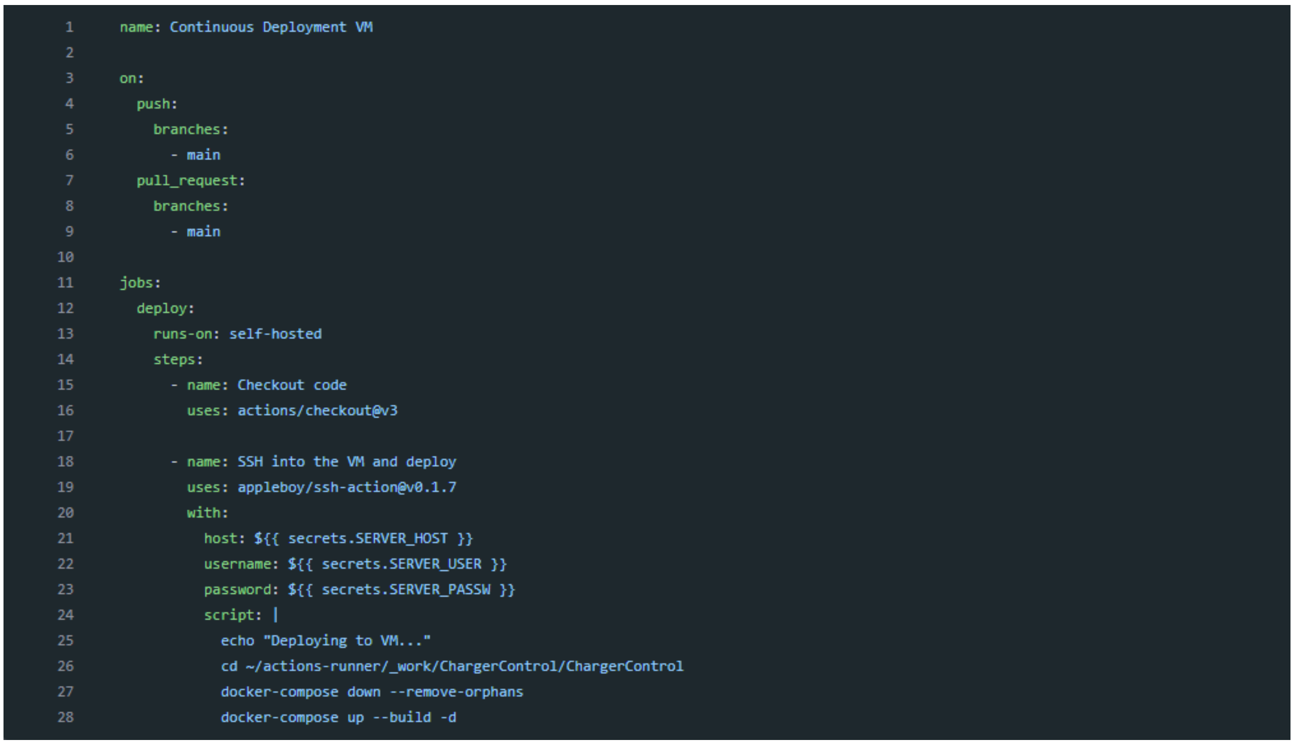


```yaml
name: Build

on:
  push:
    branches:
      - main
      - dev
  pull_request:
    types: [opened, synchronize, reopened]
    branches:
       - dev

jobs:
  build:
    name: Build and analyze
    runs-on: [self-hosted]

    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - name: Set up JDK 17
        uses: actions/setup-java@v4
        with:
          java-version: 17
          distribution: 'zulu'

      - 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:
```



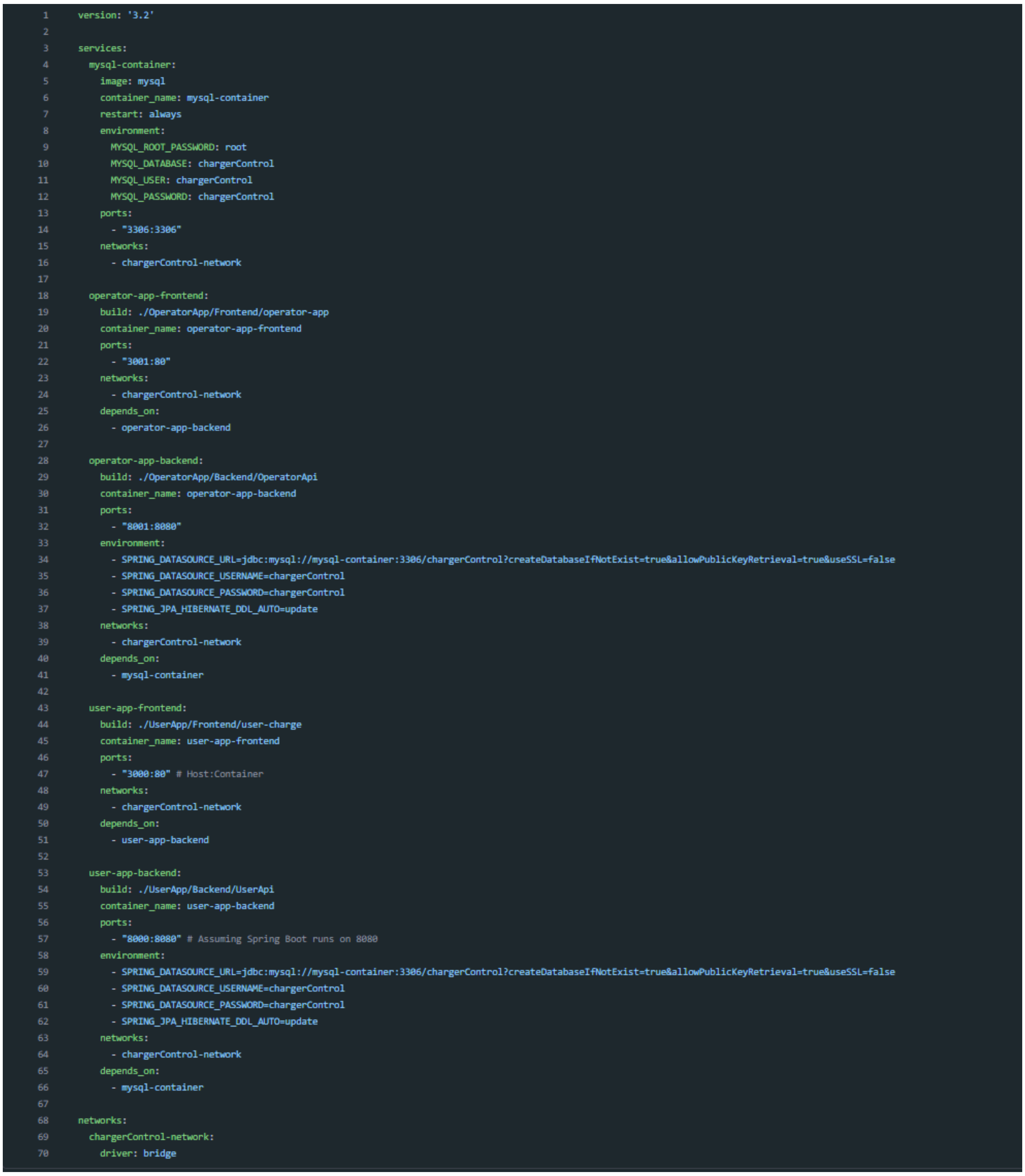
```
        with:
          path: ~/.m2
          key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
          restore-keys: ${{ runner.os }}-m2

      - name: Start database container
        run: sudo docker-compose up -d mysql-container

      - name: Build and analyze UserAPI
        working-directory: UserApp/Backend/UserApi
        env:
          SONAR_TOKEN: ${{ secrets.USERAPI_TOKEN }}
          SONAR_HOST_URL: ${{ secrets.SONAR_HOST_URL }}
        run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=UserAPI -Dsonar.projectName='UserAPI'

      - name: Build and analyze OperatorAPI
        working-directory: OperatorApp/Backend/OperatorApi
        env:
          SONAR_TOKEN: ${{ secrets.OPERATORAPI_TOKEN }}
          SONAR_HOST_URL: ${{ secrets.SONAR_HOST_URL }}
        run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=OperatorAPI -Dsonar.projectName='Operat
```

Docker containers provide a flexible and portable infrastructure, allowing for the creation of isolated and replicable environments for running the application at different stages of the continuous delivery process. In our application, each of the services, both front-end and back-end, is encapsulated in a Docker container. Additionally, in each of the back-end services, there is a Dockerfile that specifies the configuration required to build the corresponding Docker container. This approach ensures that each component of the application is isolated and executed consistently, regardless of the environment in which it is being deployed. Our project's continuous delivery pipeline is automatically triggered after changes in the "main" branch, ensuring efficient and reliable delivery of new features and bug fixes to end users on our server.
CD Pipeline:



```
name: Continuous Deployment VM

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  deploy:
    runs-on: self-hosted
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: SSH into the VM and deploy
        uses: appleboy/ssh-action@v0.1.7
        with:
          host: ${{ secrets.SERVER_HOST }}
          username: ${{ secrets.SERVER_USER }}
          password: ${{ secrets.SERVER_PASSW }}
          script: |
            echo "Deploying to VM..."
            cd ~/actions-runner/_work/ChargerControl/ChargerControl
            docker-compose down --remove-orphans
            docker-compose up --build -d
```

Docker Compose:



```yaml
version: '3.2'

services:
  mysql-container:
    image: mysql
    container_name: mysql-container
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: chargerControl
      MYSQL_USER: chargerControl
      MYSQL_PASSWORD: chargerControl
    ports:
      - "3306:3306"
    networks:
      - chargerControl-network

  operator-app-frontend:
    build: ./OperatorApp/Frontend/operator-app
    container_name: operator-app-frontend
    ports:
      - "3001:80"
    networks:
      - chargerControl-network
    depends_on:
      - operator-app-backend

  operator-app-backend:
    build: ./OperatorApp/Backend/OperatorApi
    container_name: operator-app-backend
    ports:
      - "8001:8080"
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://mysql-container:3306/chargerControl?createDatabaseIfNotExist=true&allowPublicKeyRetrieval=true&useSSL=false
      - SPRING_DATASOURCE_USERNAME=chargerControl
      - SPRING_DATASOURCE_PASSWORD=chargerControl
      - SPRING_JPA_HIBERNATE_DDL_AUTO=update
    networks:
      - chargerControl-network
    depends_on:
      - mysql-container

  user-app-frontend:
    build: ./UserApp/Frontend/user-charge
    container_name: user-app-frontend
    ports:
      - "3000:80" # Host:Container
    networks:
      - chargerControl-network
    depends_on:
      - user-app-backend

  user-app-backend:
    build: ./UserApp/Backend/UserApi
    container_name: user-app-backend
    ports:
      - "8000:8080" # Assuming Spring Boot runs on 8080
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://mysql-container:3306/chargerControl?createDatabaseIfNotExist=true&allowPublicKeyRetrieval=true&useSSL=false
      - SPRING_DATASOURCE_USERNAME=chargerControl
      - SPRING_DATASOURCE_PASSWORD=chargerControl
      - SPRING_JPA_HIBERNATE_DDL_AUTO=update
    networks:
      - chargerControl-network
    depends_on:
      - mysql-container

networks:
  chargerControl-network:
    driver: bridge
```

3.3 System observability

We implemented proactive monitoring using Prometheus and Grafana. System metrics such as CPU usage, memory, disk I/O, and HTTP response times are continuously monitored. Alerts are triggered via Slack and email if CPU usage exceeds 85% or if the database becomes unresponsive. Collected data includes system performance metrics, API error rates, and user interaction logs. This data is reviewed weekly to identify trends and preempt potential issues.

4 Software testing

4.1 Overall testing strategy

We adopted a hybrid approach combining Test-Driven Development (TDD) and Behavior-Driven Development (BDD) to ensure both technical correctness and alignment with user requirements.

Developers write unit tests first, before implementing functionality, following the red-green-refactor cycle. These unit tests were written using frameworks like JUnit,to validate low-level logic and enforce clean, testable code.Code was only considered complete when all unit tests passed, ensuring consistent test coverage and modular design.

For features with clear business scenarios or acceptance criteria, we used Cucumber to write human-readable features.These scenarios were created in collaboration with team members to reflect the expected behavior of the system from the user's perspective.Step definitions were implemented in code to connect these specifications to testable logic, acting as high-level integration tests and acceptance tests.

All tests are fully integrated into the Continuous Integration (CI) pipeline.

On every push or pull request the CI pipeline runs both unit tests and BDD scenarios automatically.If any test fails, the pipeline blocks the merge, ensuring quality enforcement.Test reports are generated for each pipeline run and made available in the CI dashboard .Code coverage tools are used to track how much of the codebase is covered by tests.

Test results are actively reviewed during code reviews, we verify that TDD and BDD tests exist for all new features or bug fixes.If tests fail, they must be addressed before any merge or deployment.

4.2 Functional testing and ATDD

We follow a clear policy for writing functional closed-box tests that simulate user interactions and verify that the system behaves as expected, independently of the internal code structure. These functional tests are made to validate complete user-facing features against requirements, focusing on expected inputs, outputs, and system behavior rather than internal implementation.

4.3 Developer facing tests (unit, integration)

We wroteUnit tests to validate individual functions, classes, or methods in isolation. They focus on verifying small, testable units of logic from our perspective.We use tools like JUnit, code coverage is tracked using tools like JaCoCo.

Developers are expected to write or update unit tests in the following cases:

When developing new modules or business logic, every function or class must have unit tests covering typical and edge cases. When fixing bugs, a regression unit test must be added to reproduce and verify the fix. Before submitting a pull request, unit tests must cover the logic introduced or modified.

Most Relevant Unit Tests in the Project:

-Validation of core business logic
-Utility functions
-Error handling for expected exceptions
-Model validation

We used Integration tests to verify how different parts of the system interact like the database calls, API endpoints, service-to-service communication.

We use tools like REST-Assured and Postman to simulate full request/response cycles, including error cases and edge inputs. We also used a mock environment to avoid affecting production data.The integration tests are part of the CI pipeline, and test reports are generated automatically.

4.4 Exploratory testing

We conduct exploratory tests during and after development sprints to simulate real user behaviors and uncover usability issues or unexpected bugs. These tests are guided by user stories, risk analysis, and recent changes but do not follow rigid scripts.

4.5 Non-function and architecture attributes testing

We wrote performance tests to ensure that the system meets expected speed, responsiveness, and scalability standards under varying levels of load. These tests help identify bottlenecks, latency issues, or resource limitations before release.

We use tools such as:

-k6:for scripting and automating load and stress tests.
-Lighthouse:for frontend performance audits

We made this types of tests:

-Load Testing – to measure system behavior under expected user traffic.
-Stress Testing – to test the limits of the system beyond normal load.
-Spike Testing – to see how the system handles sudden increases in traffic.