TQS: Quality Assurance manual

Gabriel Santos [113682], Guilherme Santos [113893],

João Gaspar [114514], Shelton Agostinho [115697]

v2025-05-14

Contents

1 Project management

1.1 Assigned roles

Gabriel Santos - Product Owner
Guilherme Santos - QA Engineer
João Gaspar - DevOps master
Shelton Agostinho - Team Coordinator

1.2 Backlog grooming and progress monitoring

Organizing Work in JIRA

- **Hierarchy logic**
  We structure all work around Epics, each representing a major feature or functional area (e.g., Station Discovery, Booking, Charging Session). Under each Epic, we create the User Stories defined in our product backlog. These describe the user-facing goals and acceptance criteria. Each User Story is then implemented through its subtasks that break down the user story into pieces. This allows us to track technical progress toward the completion of each functional requirement. These sub-tasks are used to manage the step-by-step work of individual developers and ensure visibility into the development process.
- **Backlog Refinement**
  At the start of each sprint we hold a **Backlog Grooming** session: the Product Owner and team review upcoming Stories, confirm acceptance criteria, and ensure each Story is "Ready" (clear description, estimate, dependencies resolved).

Tracking Progress

- **Story Point Estimation**
  In our Sprint Planning we assign **story points** to each Story, reflecting relative effort. This helps the team commit to a realistic amount of work.
- **Sprint Board**
  Our Scrum board has columns **To Do**, **In Progress**, **In Review** and **Done** to ensure Stories flow smoothly to completion.

Requirement-Level Coverage Monitoring

- **Manual Traceability via Sub-tasks**
  We do not yet integrate a dedicated test-management plugin in JIRA. Instead, each Story contains sub-tasks. When all sub-tasks are complete, the Story is completed
- **Linking Code & Tests**
  Developers link their Pull Requests and CI build results back to the corresponding JIRA issue. This provides a manual but clear traceability.

2 Code quality management

2.1 Team policy for the use of generative AI

We allow the use of generative AI tools to **support** code writing, testing, and documentation, as long as the output is:

1. Understood by the developer;
2. Reviewed critically before insertion;

3. Compliant with the team's style.

DO's:

1. Use AI to suggest **simple** code or test templates.
2. Use it for improving readability or formatting code into the team's style.
3. Use it for catching and solve **simple** errors/malpractices

DON'Ts:

1. Don't copy-paste AI-generated code **without review** or adaptation.
2. Don't use AI for **critical logic** or configurations without validation.
3. Don't use AI to bypass code review without personal evaluation.

2.2 Guidelines for contributors

Coding style

Our coding style is based on the **Google's Java Style Guide** with the help of some tools for automatic formatting:

1. We use **camelCase** for variables and methods;
2. We use **PascalCase** for classes and interfaces;
3. All other formatting (indentation, braces, spacing, etc.) is automatically handled using **Prettier** in every commit, which runs in the CI pipeline.

Code reviewing

Code review must be done **every time** that is made a Pull Request (PR) to the **dev** branch

1. All code must be submitted **via PRs**;
2. At **least one peer** must approve the PR;
3. AI tools (like GithubCopilot) may assist in reviews, but human review is **mandatory**.

2.3 Code quality metrics and dashboards

Backend (BE): We use **SonarQube** to perform automated static code analysis on all backend commits. SonarQube reports on issues such as bugs, vulnerabilities, code smells, duplication, and also aggregates code coverage data from our test suites although we have not yet defined explicit coverage thresholds (quality gates) for blocking merges.

Frontend (FE):

1. **Prettier** is integrated into the pipeline as an opinionated code formatter that automatically enforces consistent styling (indentation, braces, spacing) on every save. It provides a "format-on-save" safety net but offers limited configurability;
2. **ESLint** runs after Prettier to catch linting errors and enforce code-quality rules. By default it applies the community-recommended rule set, but it can be extended or tuned.

Dashboard & Quality Gates:

The SonarQube dashboard consolidates **BE** and **FE** analysis results for an overall view of code health.

In future iterations, we will define and enforce minimum coverage and issue thresholds as part of our GitHub pull-request quality gates.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow
Work flows directly from JIRA into our GitHub repository following a GitHub-Flow variant, but with branch creation automated by JIRA. For each Issue, JIRA provides a “Create branch” button that generates a properly named feature branch (e.g. `EDISON-123-search-stations`) off the shared `dev` branch. Developers then simply pull and checkout that branch locally:

1. Select a Story/Issue in JIRA in the **Board**;
2. **Click “Create branch”** in the JIRA UI selecting dev in the “Branch from” section
3. Select a Sub-task or Major Task related to the Story
4. Click “Create branch” now selecting the **Issue’s** branch that was created in the “Branch from” section
5. Now just git pull and git checkout <sub-task-branch-name> and start coding
6. When a sub-task is done, make a PR to merge into the Task’s branch in Github and merge the branch.
7. After all the sub-tasks of a task are done, make a PR to the dev, make sure that there are no merging errors by doing git merge dev and fixing any conflicts and **assign a peer for Review.**

Definition of done
A User Story is only marked **Done** in JIRA when **all** the following conditions are satisfied:

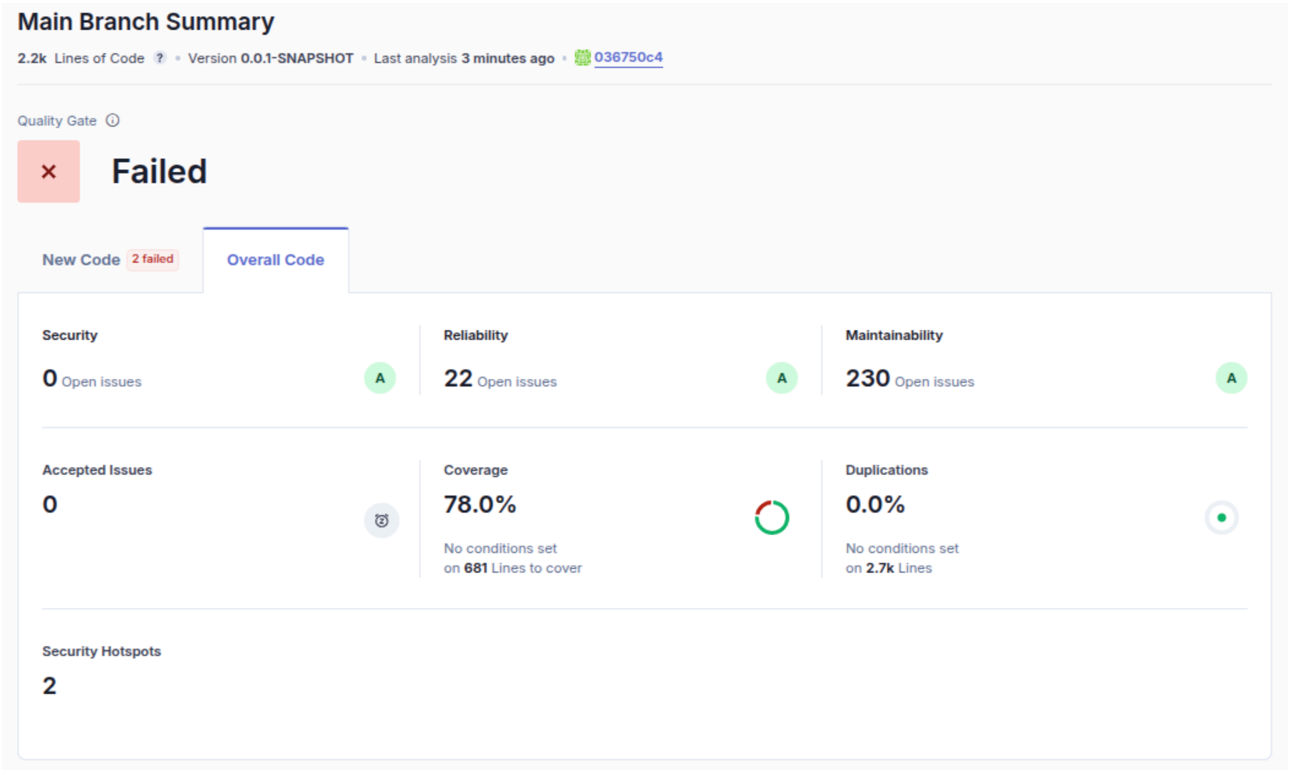
- **Code implemented** meets every acceptance criteria.
- **Sub-tasks** are all resolved and closed.
- **Automated tests** (unit and integration) execute successfully in CI.
- **Static analysis** (Prettier, ESLint, SonarQube) reports no new blocking issues.
- **Pull Requests** for sub-tasks and story branches have all approving reviews (those who have reviews).
- **Documentation** is updated to reflect any new APIs or configuration.
- **Merged into `dev`** without conflicts, ensuring the integration branch remains stable.
- **Demo-ready**: the feature can be deployed to our staging environment for validation and sprint review.

3.2 CI/CD pipeline and tool

Continuous Integration

As mentioned before, we have a CI pipeline that ensures that new additions comply with our defined coding style and doesn’t break existing code.
To ensure coding style in both backend and frontend, we used Prettier, which automatically formats the edited code before committing new changes, this style is defined in *.prettierrc* file, therefore if our coding style changes, we just have to enforce them by editing the rules present in that file. This pre-commit is done through the usage of a tool called Husky that allows a definition of a script to run pre-commit, this pre-commit hook runs a script called lint-staged that passes all

staged files through our code formatter, Prettier for both Frontend and Backend, and the linter ESLint for the frontend.
We also use static code analysis tools: ESLint in the frontend and SonarQube in the backend. However, SonarQube is run only when a new PR is created/updated or when a commit is made to the dev and main branches, through Github actions.
Alongside SonarQube, we also run the tests in the same workflow.
For the Frontend, we also defined a workflow that runs the Linter to do a final check and also builds our React + TS + HTML application into HTML + JS build files.


Main Branch Summary

2.2k Lines of Code · Version 0.0.1-SNAPSHOT · Last analysis 3 minutes ago · 036750c4

Quality Gate

Failed

New Code 2 failed | Overall Code

Security: 0 Open issues — A
Reliability: 22 Open issues — A
Maintainability: 230 Open issues — A

Accepted Issues: 0
Coverage: 78.0% — No conditions set on 681 Lines to cover
Duplications: 0.0% — No conditions set on 2.7k Lines

Security Hotspots: 2

Figure 1. Sonarcloud interface

In Figure 1., is shown the final assessment of our code at the end of Sprint 4. It is visible that we fail the quality gate, this is due to a minimum of 80% coverage to new code being set natively by Sonarcloud, however we missed it by a small margin that should be addressed in the future, we also have some maintainability issues that should be also addressed but they should be fast to fix.

Continuous Deployment

To ensure our application gets updated in every major release, which is defined as a push to the main branch, we adopted a CD pipeline based on a self-hosted Github Actions.
When the main branch is updated, a special workflow runs that, in our VM, stops the existing dockerized containers that hosted the previous version of our application, and builds new containers to host the new updated application with the updated injected environment variables.
This automated process ensures that our application is always up to date and minimizes downtime.
In an event where our database schema also changes, we implement flyway to allow us to define a migration process that ensures the new schema is implemented while preserving existing data.

3.3 System observability

To ensure observability, we needed to insert 3 tools into our application:
- Prometheus: That allows our backend to send desired statistical data of the operations made in our system, defined by us, this data is mainly the system load such as CPU and memory usage, number of booking requests, registrations, stations, logins, rate of requests etc.
- Grafana: Used to create a dashboard to present the data gathered by Prometheus in a visual and appealing form, through queries written in PromQL (Prometheus Query Language)

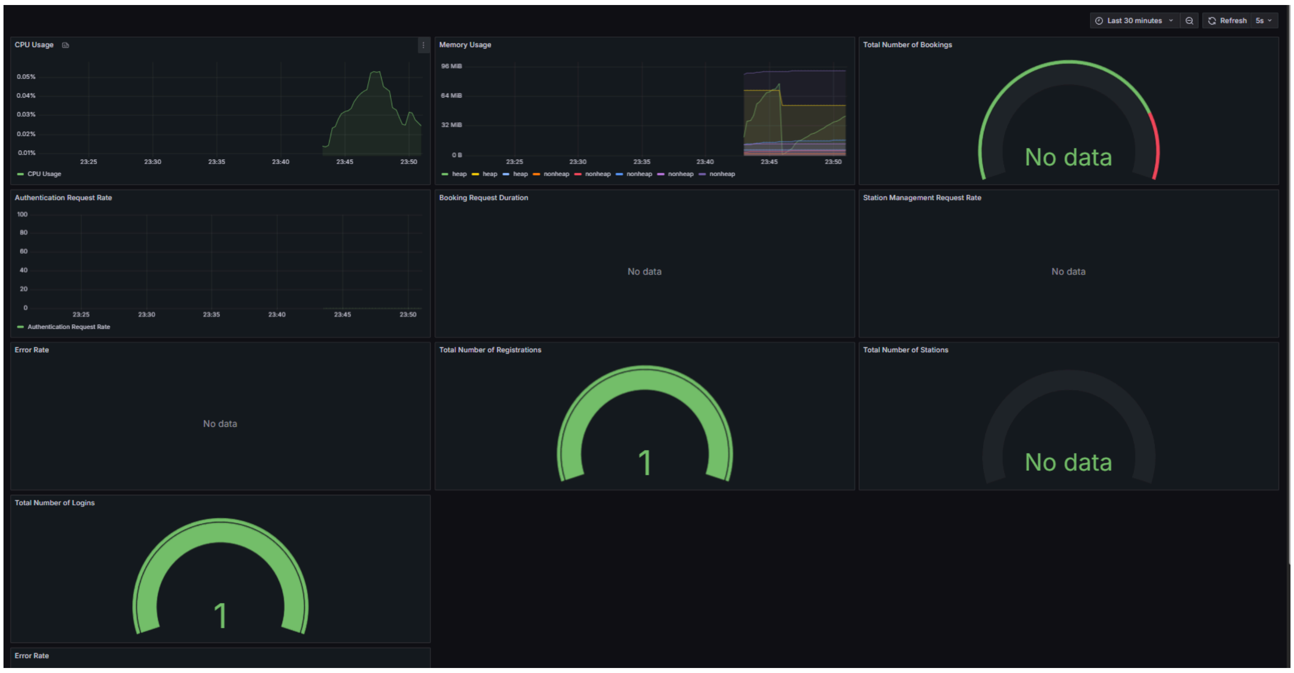


Figure 2. Grafana Dashboard

- Nagios: Used to make intermittent health checks to all our services and endpoints, to ensure that downtimes are detected early and fixed as soon as possible, by sending a notification to the developers team if a downtime is detected.

4 Software testing

4.1 Overall testing strategy

Our team adopted a **mixed testing strategy**, combining different approaches and tools to ensure that functional correctness, integration stability, and business behavior are all well covered.

We follow a **Test-Driven Development (TDD)** methodology for new features: tests are written first, fail initially, and are then used to drive the implementation of the necessary classes and logic. Once the implementation is complete, the tests pass, providing immediate feedback on correctness.

In parallel, we also apply **Behavior-Driven Development (BDD)** practices using **Cucumber**, which allows us to express user-level behavior in business-readable scenarios. These scenarios help validate not only the technical logic but also ensure alignment with the functional requirements defined in the user stories.

Testing Tools and Frameworks

- Backend: We use JUnit, Testcontainers, and Test Containers for isolated and repeatable tests. Flyway is used to apply migrations in test environments to mirror real database behavior.

- Frontend: Linting and formatting are enforced using ESLint and Prettier, and the testing workflow is being incrementally expanded.

- Xray (JIRA Plugin): Test cases and execution results are tracked and documented within JIRA using Xray, allowing us to map coverage back to requirements and user stories.

CI Integration
All tests are automatically executed as part of the **CI pipeline**:
- **Backend CI**: Integrated with **SonarCloud** to analyze code quality and coverage.
- **Frontend CI**: Validates formatting and code quality using **Prettier** and **ESLint**.

The team collectively ensures that all scenarios (normal flows, edge cases, and exceptions) are covered with meaningful tests. Pull requests are only approved when test coverage and behavior are considered sufficient by peers.

4.2 Functional testing and ATDD

Our project adopts **Acceptance Test-Driven Development (ATDD)** to validate system behavior from the user's perspective. Functional tests are written as **closed-box tests**, focusing on verifying that the system meets its expected behavior based on user stories and acceptance criteria.

When are they written?

Functional tests are created during the Sprint, before the implementation of the related feature. Developers are responsible for writing them in parallel with the development of the user story.

Tools and Practices

- **Cucumber** is used to define acceptance tests in Gherkin syntax, directly reflecting user story criteria.
- Functional scenarios are also tracked in **Xray (JIRA)**, ensuring traceability between requirements and test coverage.

4.3 Developer facing tests (unit, integration)

Unit Testing Policy

We adopt an **open-box** approach for unit testing. Developers are required to write unit tests for all new classes and methods with business logic. These tests are written during or immediately after implementation and follow the principles of **Test-Driven Development (TDD)**.

- **Technologies Used**: JUnit 5, Test Containers, Mockito

Integration Testing Policy

We apply a **mixed open/closed box** strategy for integration testing, depending on context. Tests validate interactions between application components (e.g., REST controllers, services, database).

- **Technologies Used**: Spring Boot Test, Testcontainers, Flyway, Cucumber, Selenium, Rest Assured, Mockito

4.4 Exploratory testing

Although our primary testing strategy is based on scripted and automated tests, we also incorporate **exploratory testing** as a complementary practice, particularly during the final stages of each sprint or before major merges into main.

Strategy

Exploratory testing is conducted **informally and without predefined test cases**, focusing on identifying unexpected behavior, usability issues, or edge cases that may not be captured by automated tests.

- **Who performs it**: Any team member can carry out exploratory sessions.
- **When it's done**: Typically at the end of a sprint, during review or demo preparation, or after merging a larger feature into dev.

Reporting

Findings from exploratory testing are documented directly in JIRA as:

- **Bug reports** (with steps to reproduce if applicable);
- **Improvement** suggestions.

4.5 Non-function and architecture attributes testing

To test how well our architecture would handle this first phase of our project, where we assume that we won’t have a large number of users, we decided to do some load tests through a tool called K6 that allowed us to define, in javascript, three main tests, to load our system in the four critical main endpoints: Register, Login, Station Search and Booking endpoints. Therefore we, with a variable number of Virtual Users between 20 and 60, sent several packets with valid data to these endpoints over the course of 20 minutes, and got the following results, in K6 dashboard

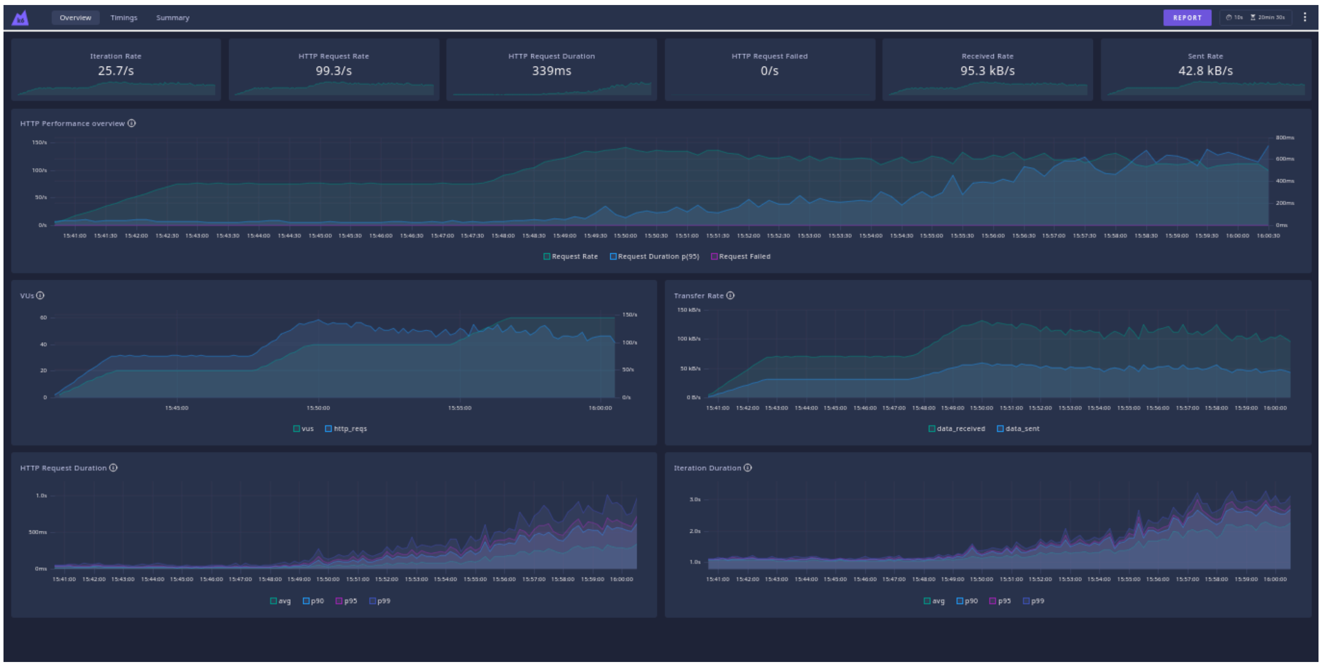
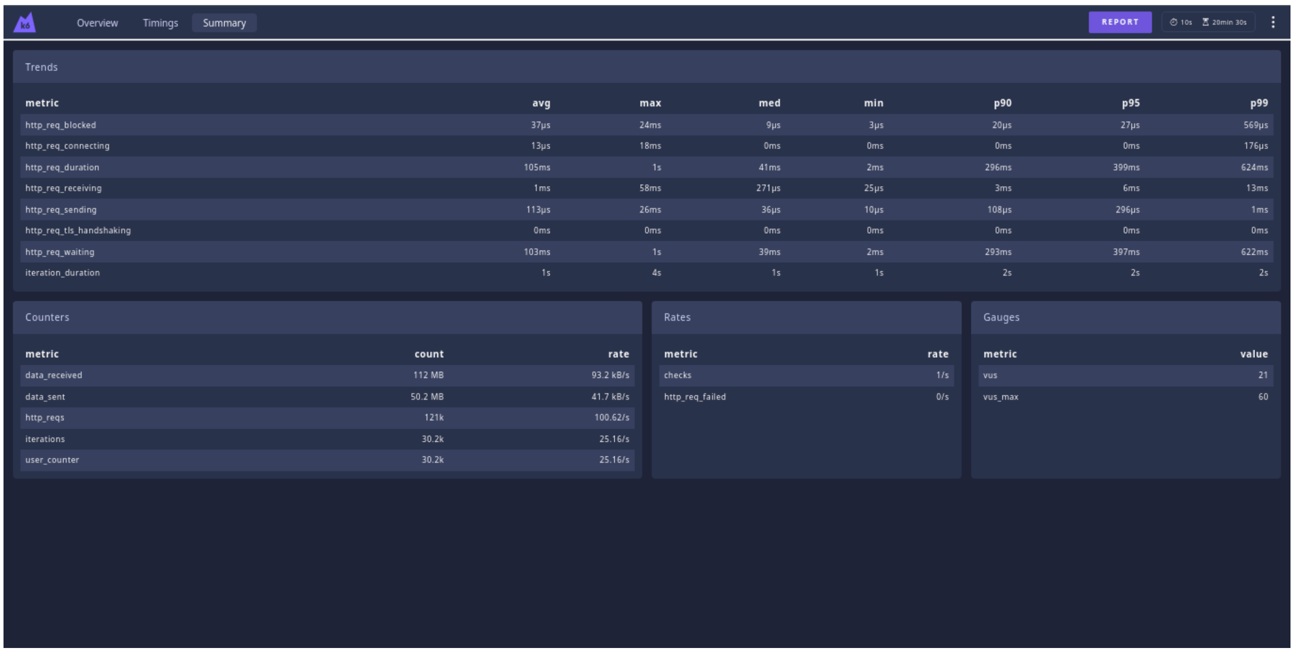


Figure 3. Graphs of the load over time



Trends

metric	avg	max	med	min	p90	p95	p99
http_req_blocked	37µs	24ms	9µs	3µs	20µs	27µs	569µs
http_req_connecting	13µs	18ms	0ms	0ms	0ms	0ms	176µs
http_req_duration	105ms	1s	41ms	2ms	296ms	399ms	624ms
http_req_receiving	1ms	58ms	271µs	25µs	3ms	6ms	13ms
http_req_sending	113µs	26ms	36µs	10µs	108µs	296µs	1ms
http_req_tls_handshaking	0ms	0ms	0ms	0ms	0ms	0ms	0ms
http_req_waiting	103ms	1s	39ms	2ms	293ms	397ms	622ms
iteration_duration	1s	4s	1s	1s	2s	2s	2s

Counters

metric	count	rate
data_received	112 MB	93.2 kB/s
data_sent	50.2 MB	41.7 kB/s
http_reqs	121k	100.62/s
iterations	30.2k	25.16/s
user_counter	30.2k	25.16/s

Rates

metric	rate
checks	1/s
http_req_failed	0/s

Gauges

metric	value
vus	21
vus_max	60

Figure 4. Table with statistical data of the response of the loadtests

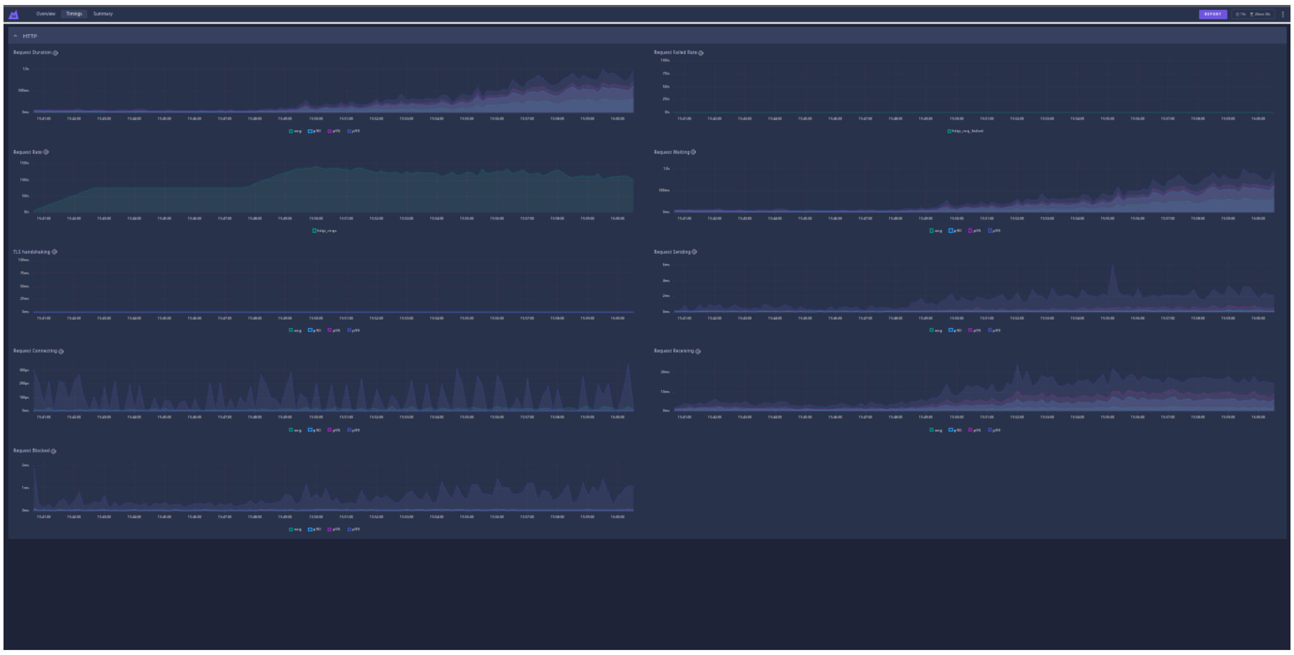


Figure 5. Detailed requests info of the loadtests

In this loadtest, we were able to understand that for a first deployment, our system would be able to handle large concurrent requests, however, if our system is to grow in user numbers, then we would need to update our system and deployment architecture to handle that increase in load. Nonetheless, we believe our system handled the provided stress tests well.