universidade de aveiro
departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Eduardo Rosário [119234], Henrique Lopes [119954], Simão Santos [119042], Filipe Marques [120303]

v2025-06-22

Contents

1 Project management

1.1 Assigned roles

The team consists of four members, each with specific responsibilities while maintaining active collaboration in the development of ToolShed:

Filipe Marques - Team Coordinator
Ensure that there is a fair distribution of tasks and that members work according to the plan. Actively promote the best collaboration in the team and take the initiative to address problems that may arise. Ensure that the requested project outcomes are delivered on time

Henrique Lopes - Product Owner
Represents the interests of the stakeholders.
Has a deep understanding of the product and the application domain; the team will turn to the Product Owner to clarify the questions about expected product features.
Should be involved in accepting the solution increments.

Eduardo Rosário - QA Engineer
Responsible, in articulation with other roles, to promote the quality assurance practices and put in practice instruments to measure que quality of the deployment. Monitors that team follows agreed QA practices

Simão Santos- DevOps Master
Responsible for the (development and production) infrastructure and required configurations. Ensures that the development framework works properly. Leads the preparing the deployment machine(s)/containers, git repository, cloud infrastructure, databases operations, etc.

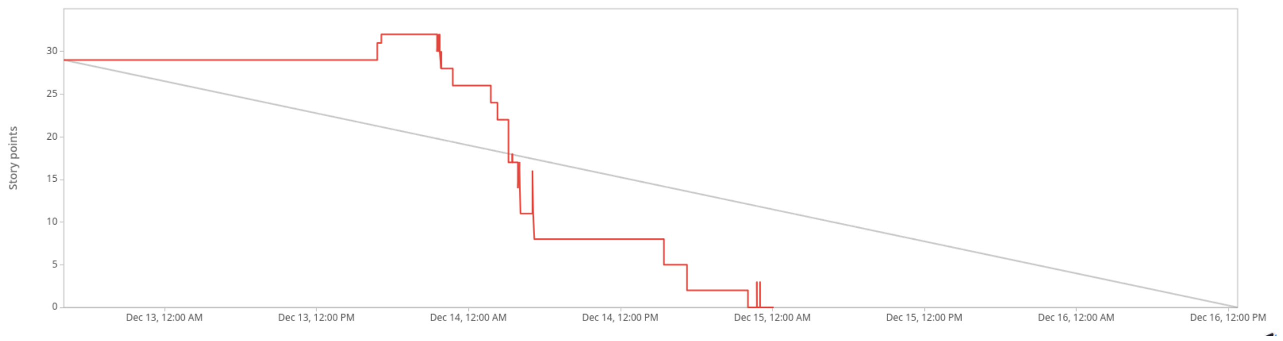
1.2 Backlog grooming and progress monitoring

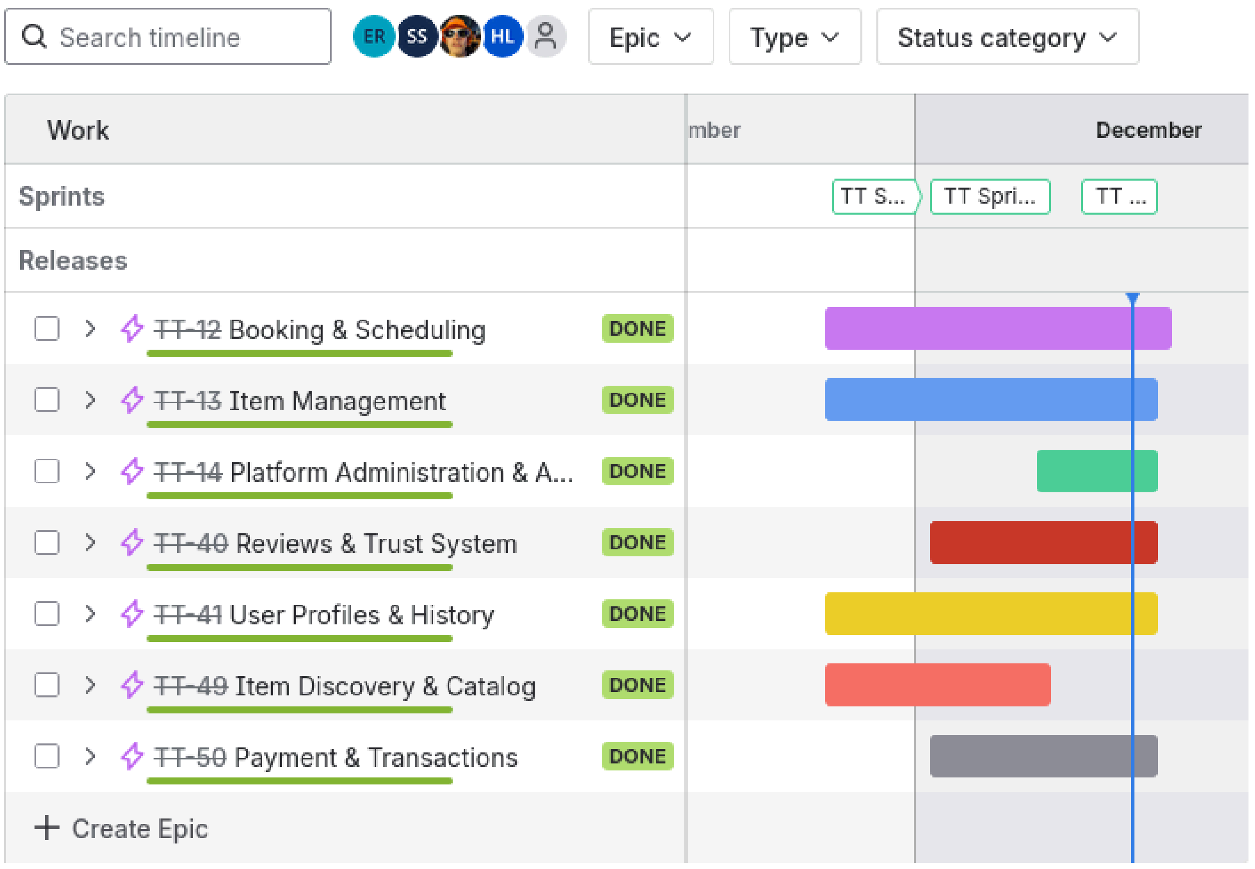
Work is organized in Jira based on User Stories, structured in a backlog, and planned in weekly sprints. In each iteration, stories are reviewed, prioritized, and decomposed into technical tasks to ensure execution readiness. All stories are categorized by functional areas and assigned an owner and effort estimate. Backlog grooming is performed regularly to ensure alignment, requirement clarity, and estimation consistency.

Progress monitoring is conducted via:

- **Story Points:** For effort estimation.
- **Execution States:** To Do, In Progress, In Review, Done.
- **Burndown Charts:** Per sprint analysis.
- **Jira Automation:** Tracking planned vs. completed points.

Additionally, test coverage is continuously monitored via **SonarCloud**. Every pull request triggers an automatic analysis of test coverage, code duplication, security hotspots, and potential vulnerabilities, contributing to continuous improvement and failure prevention.





Work	Status
~~TT-12~~ Booking & Scheduling	DONE
~~TT-13~~ Item Management	DONE
~~TT-14~~ Platform Administration & A...	DONE
~~TT-40~~ Reviews & Trust System	DONE
~~TT-41~~ User Profiles & History	DONE
~~TT-49~~ Item Discovery & Catalog	DONE
~~TT-50~~ Payment & Transactions	DONE

2 Code quality management

2.1 Team policy for the use of generative AI

Generative AI was utilized as a support tool throughout the project, specifically for frontend development assistance, debugging support, technical report generation, code review assistance, and test coverage reinforcement. AI usage never replaced team effort; it was applied exclusively as a support mechanism. It is important to note that AI was not used to generate the application code in its entirety.

DO:

- Ask for a review of code
- Check if code if being covered

DON'T:

- Ask the AI to do entire user stories
- Accept AI generated code without reviewing it

2.2 Guidelines for contributors

Coding Style The project follows a consistent coding style aligned with current best practices to ensure readability, maintainability, and scalability.

- **Backend (Java 21 + Spring Boot 3):** Adopts a layered architecture organized by functionality (Controllers/Boundary, Services, Repositories, and DTOs). Controllers follow the RESTful pattern returning `ResponseEntity`, with OpenAPI documentation. Data validation uses `jakarta.validation`, maintaining separation between JPA entities and DTOs. Tests use JUnit 5, Mockito, and MockMvc (Arrange/Act/Assert pattern).
- **Frontend (TypeScript + React + Vite):** Uses functional components with hooks, organized broadly by domain. Linting is enforced via ESLint, and styling uses Tailwind CSS.

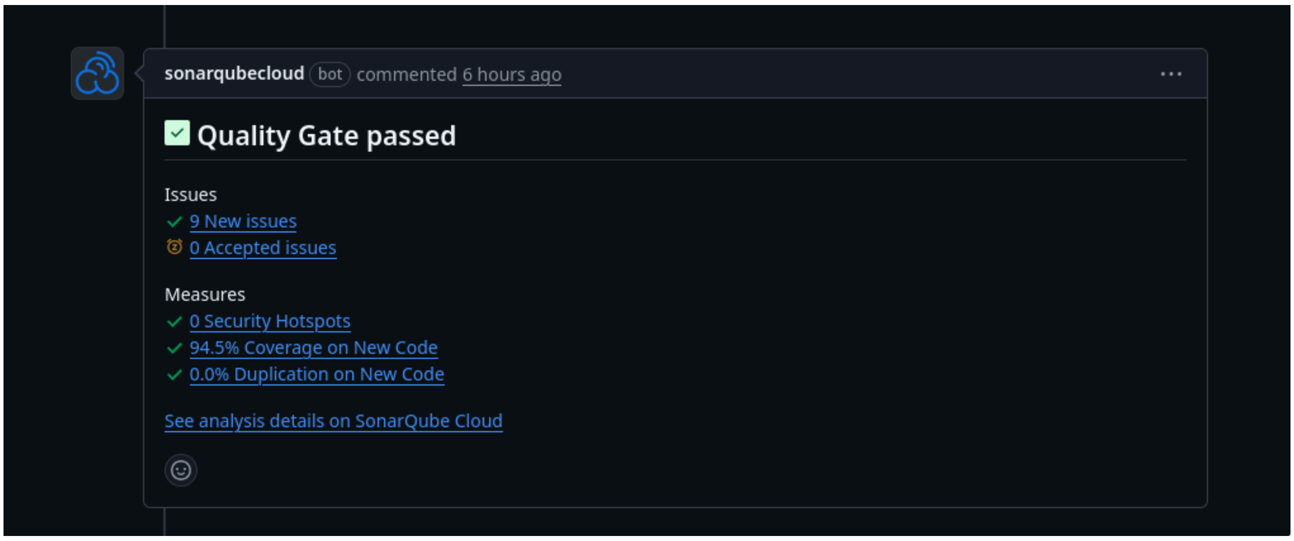
Code Reviewing Code is integrated exclusively via Pull Requests (PRs), with mandatory code reviews before merging into the main branch. Github Copilot can comment on a PR.

Adopted practices include:

- Verification of functional correctness, readability, and compliance with the coding style.
- Validation of adequate tests for developed features.
- **Mandatory manual review** by at least one team member (GitHub Copilot may be used to assist the reviewer).
- Generative AI tools may identify potential issues or optimization suggestions, but the final integration decision remains a human responsibility.

2.3 Code quality metrics and dashboards

Static code analysis and continuous quality monitoring are ensured via the CI pipeline and **SonarCloud** integration.



sonarqubecloud bot commented 6 hours ago

Quality Gate passed

Issues
✓ 9 New issues
0 Accepted issues

Measures
✓ 0 Security Hotspots
✓ 94.5% Coverage on New Code
✓ 0.0% Duplication on New Code

See analysis details on SonarQube Cloud

Every PR automatically evaluates:

- Test Coverage.
- Code Duplication.
- Security Hotspots and Vulnerabilities.

Quality Gates: Strict minimum quality gates dictate the acceptance or rejection of code integration:

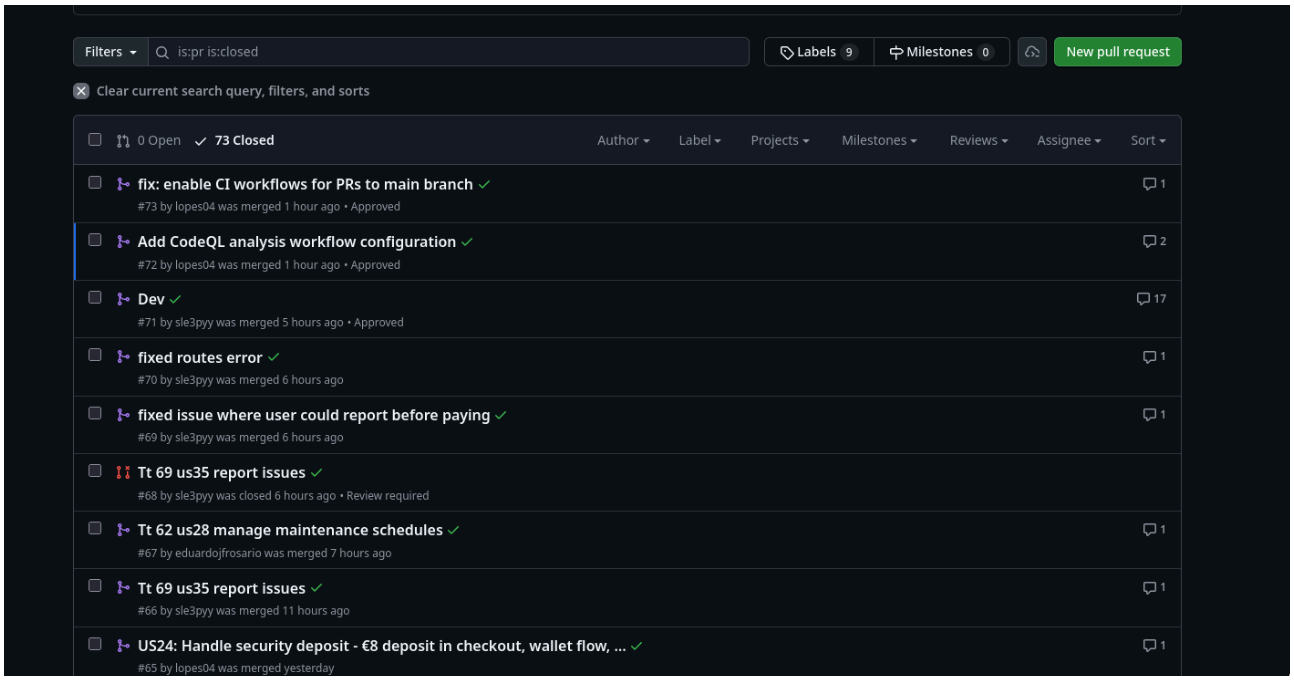
- Minimum Test Coverage (80%).
- 0 Critical Vulnerabilities.
- Limited Code Duplication (<3%).

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Work begins in Jira. Each developer selects a "To Do" User Story, moves it to "In Progress," and creates a feature branch. After local implementation and testing, a Pull Request(PR) is created, moving it to “In Review”, when a PR is accepted it moves to “Done”. The project adopts **GitHub Flow**:

1. One stable main branch (`dev`).
2. One branch per feature/User Story.
3. Integration via Pull Requests.


Filters is:pr is:closed
Labels 9 Milestones 0 New pull request
Clear current search query, filters, and sorts
0 Open ✓ 73 Closed Author Label Projects Milestones Reviews Assignee Sort

fix: enable CI workflows for PRs to main branch ✓ 1
#73 by lopes04 was merged 1 hour ago • Approved

Add CodeQL analysis workflow configuration ✓ 2
#72 by lopes04 was merged 1 hour ago • Approved

Dev ✓ 17
#71 by sle3pyy was merged 5 hours ago • Approved

fixed routes error ✓ 1
#70 by sle3pyy was merged 6 hours ago

fixed issue where user could report before paying ✓ 1
#69 by sle3pyy was merged 6 hours ago

Tt 69 us35 report issues ✓
#68 by sle3pyy was closed 6 hours ago • Review required

Tt 62 us28 manage maintenance schedules ✓ 1
#67 by eduardojfrosario was merged 7 hours ago

Tt 69 us35 report issues ✓ 1
#66 by sle3pyy was merged 11 hours ago

US24: Handle security deposit - €8 deposit in checkout, wallet flow, ... ✓ 1
#65 by lopes04 was merged yesterday

Definition of Done (DoD): A User Story is "Done" only when:

- Functionality meets acceptance criteria.
- Tests are implemented and passing.
- CI Pipeline executes without failure.
- SonarCloud Quality Gates are passed.
- Code is merged into the main branch via approved PR.

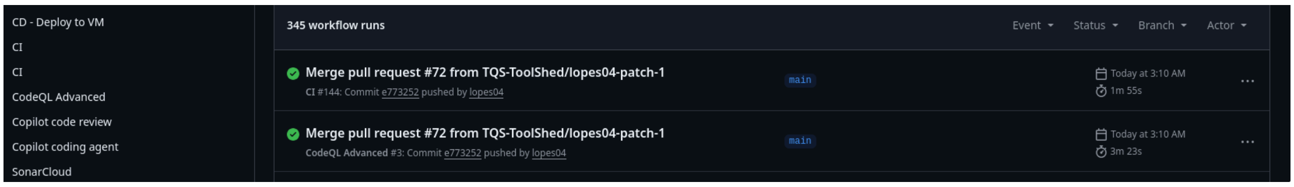
3.2 CI/CD pipeline and tools

Continuous Integration (CI)

The project uses **GitHub Actions** (`.github/workflows/build.yml`). It triggers on push and PR events to the `dev` branch using Ubuntu runners with a PostgreSQL 15 service.

- **Backend:** Automatically checks out code, sets up JDK 21, installs dependencies, and runs full Maven tests. A dedicated job runs SonarCloud analysis (using caching and `SONAR_TOKEN`).
- **Tools:** JaCoCo (Coverage), SonarCloud (Quality/Security), JUnit 5/Mockito/MockMvc (Testing).
- **Frontend:** ESLint configured for local validation.

Continuous Delivery (CD) *Note: Currently, the project utilizes Dockerfiles and `docker-compose.yml` for local container orchestration. Automated deployment to a staging environment is part of the roadmap for post-MVP iterations.*



CD - Deploy to VM
CI
CI
CodeQL Advanced
Copilot code review
Copilot coding agent
SonarCloud

345 workflow runs

Merge pull request #72 from TQS-ToolShed/lopes04-patch-1
CI #144: Commit e773252 pushed by lopes04 — main — Today at 3:10 AM — 1m 55s

Merge pull request #72 from TQS-ToolShed/lopes04-patch-1
CodeQL Advanced #3: Commit e773252 pushed by lopes04 — main — Today at 3:10 AM — 3m 23s

3.3 System observability

Currently, observability relies on SonarCloud for code health. Runtime observability (logs/metrics) is managed via container logs during local execution.

4 Continuous testing

4.1 Overall testing strategy

The strategy follows a layered approach, ensuring progressive validation from business logic to the API. We combine Unit Tests, Integration (Slice) Tests, and Context Loading tests. Execution is automated in the CI pipeline, acting as a mandatory gateway for code integration.

The project contains end to end (E2E) tests that use Cucumber and BDD with Playwright.

4.2 Acceptance testing and ATDD

Acceptance tests are defined from a black-box perspective, validating User Story acceptance criteria.

- **Current Status:** Tests are manually validated against requirements.
- **Roadmap:** Implementation of automated BDD (Cucumber) and E2E (Playwright) frameworks.

4.3 Developer facing tests (unit, integration)

Unit Testing: Follows a white-box approach focusing on business logic (Service Layer) with mocked dependencies.

- **Tools:** JUnit 5, Mockito, AssertJ.
- **Policy:** Mandatory for all business logic; must cover success, edge cases, and error scenarios.

Integration Testing: Validates the interaction between layers using Spring Boot Slice Tests:

- **Repositories:** Tested against an H2 in-memory database.
- **Controllers:** Tested via `MockMvc` to validate HTTP responses and JSON structures without full server startup.

4.4 Exploratory testing

The project strategy prioritizes **Automated Regression** over manual scripted testing. Consequently, the team does not conduct dedicated manual exploratory testing phases. Instead, a **Developer-Led Verification** approach is adopted

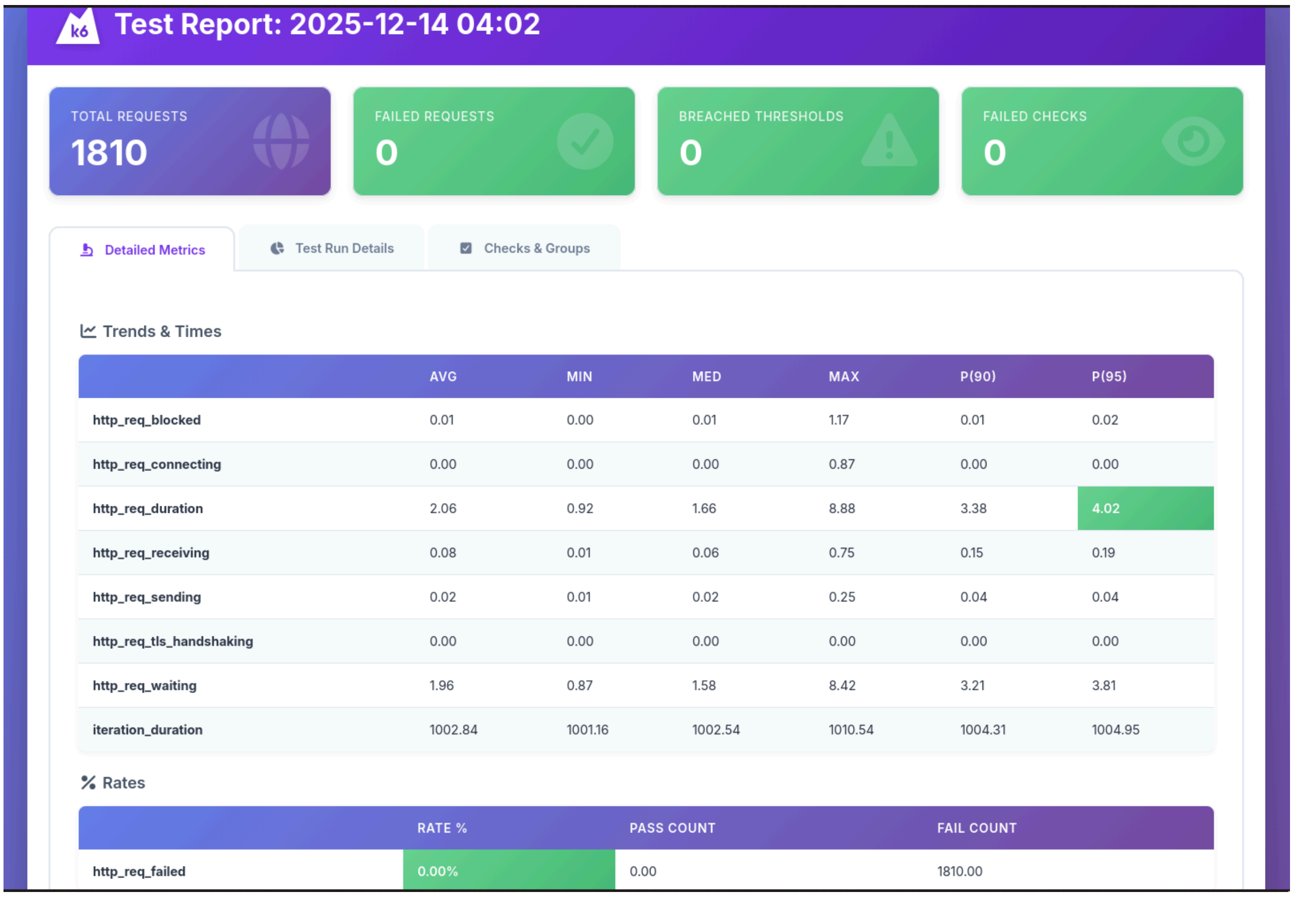
- **Implementation Validation:** Developers manually verify the happy path and edge cases of a feature locally during development before committing code.
- **Reliance on Automation:** Once code is pushed, the team relies on the extensive suite of Unit and Integration tests in the CI pipeline to catch regressions, rather than manual re-testing.
- **Pull Request Review:** Visual verification is performed ad-hoc during the Code Review process by the reviewer, focusing solely on UI/UX correctness that cannot be easily asserted via code.

4.5 Non-function and architecture attributes testing

To validate the architecture's reliability, the team performs **Load Testing** using **K6**.

Policy:

- Performance tests are executed against critical endpoints (e.g., Login, Item Search) to ensure response times remain under acceptable limits (e.g., < 500ms) under concurrent load.
- **Tool:** K6 scripts allow us to simulate multiple virtual users accessing the API simultaneously.



Test Report: 2025-12-14 04:02

TOTAL REQUESTS	FAILED REQUESTS	BREACHED THRESHOLDS	FAILED CHECKS
1810	0	0	0

Trends & Times

	AVG	MIN	MED	MAX	P(90)	P(95)
http_req_blocked	0.01	0.00	0.01	1.17	0.01	0.02
http_req_connecting	0.00	0.00	0.00	0.87	0.00	0.00
http_req_duration	2.06	0.92	1.66	8.88	3.38	4.02
http_req_receiving	0.08	0.01	0.06	0.75	0.15	0.19
http_req_sending	0.02	0.01	0.02	0.25	0.04	0.04
http_req_tls_handshaking	0.00	0.00	0.00	0.00	0.00	0.00
http_req_waiting	1.96	0.87	1.58	8.42	3.21	3.81
iteration_duration	1002.84	1001.16	1002.54	1010.54	1004.31	1004.95

Rates

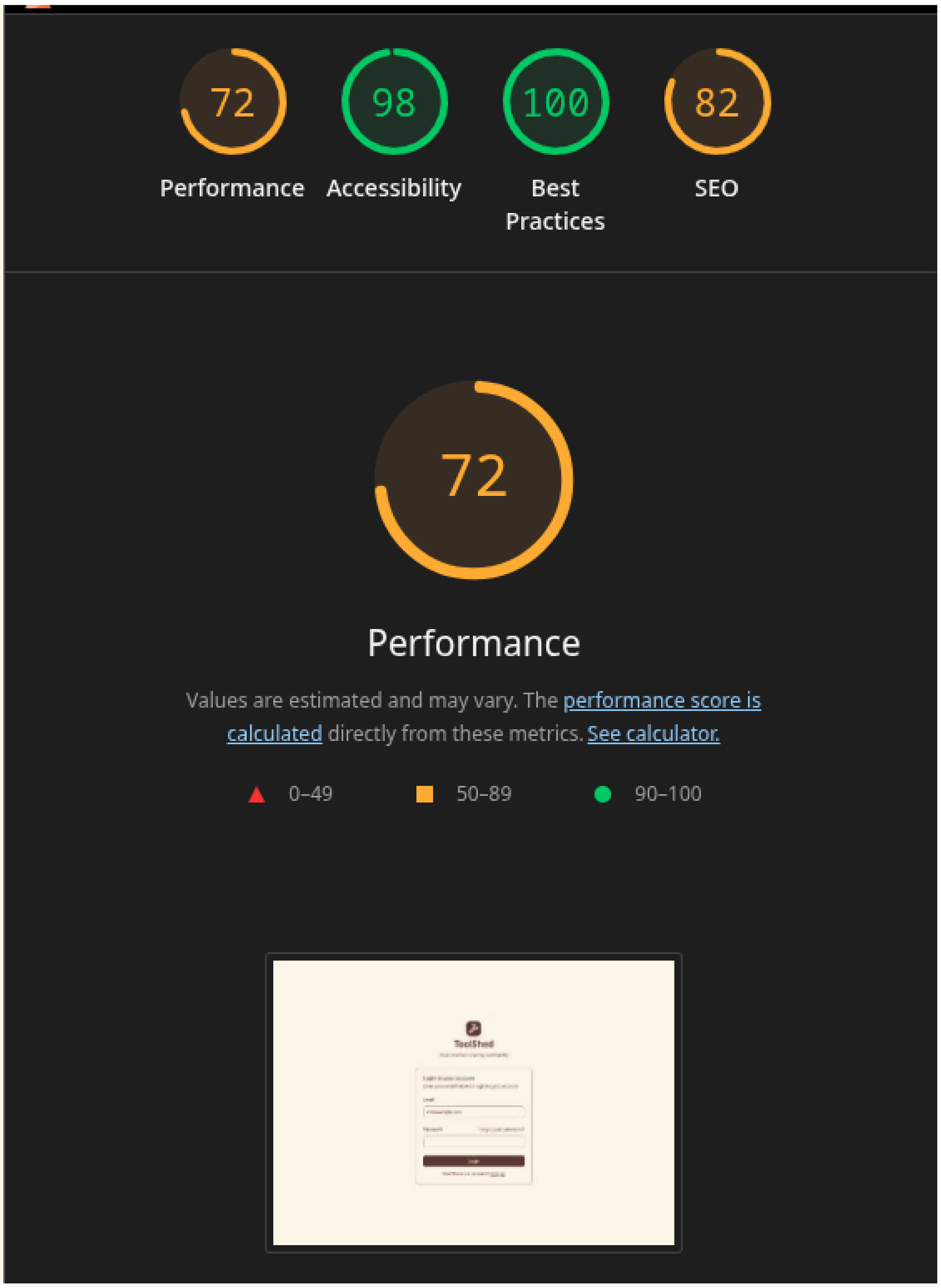
	RATE %	PASS COUNT	FAIL COUNT
http_req_failed	0.00%	0.00	1810.00

To ensure a high-quality user experience and compliance with web standards, the team utilizes **Google Lighthouse** to audit the frontend application. This tool allows us to validate non-functional requirements related to performance, accessibility, and best practices.

Policy: Critical pages (e.g., Home Page, Item Details, Login) are subject to Lighthouse audits before major releases. The team aims for a "Green" score (>90) in the Accessibility and Best Practices categories.

Metrics Monitored:

- **Performance:** Analysis of Core Web Vitals, specifically *Largest Contentful Paint (LCP)* and *Cumulative Layout Shift (CLS)*, to ensure the UI loads quickly and remains stable.
- **Accessibility:** Validation of ARIA labels, color contrast ratios, and keyboard navigation support to ensure the platform is usable by all users.
- **SEO & Best Practices:** Checks for HTTPS usage, correct metadata, and secure JavaScript practices.


72
Performance

98
Accessibility

100
Best Practices

82
SEO

72

Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 ■ 50–89 ● 90–100