PRONTO

Testing Policy and Quality Assurance metrics

Name	Student Number
Shashin Gounden	u21458686
Tyrone Sutherland-MacLeod	u21578878
Bandisa Masilela	u19028182
Andile Ngwenya	u20612894
Jonel Albuquerque	u21598267

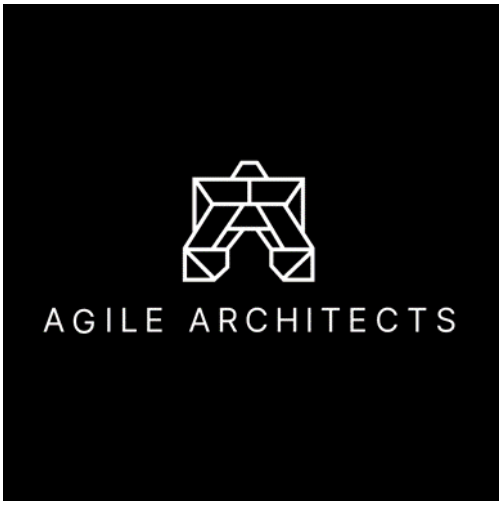


Table of Contents

Quick links:

Github Repo : https://github.com/COS301-SE-2023/Pronto

1. Introduction

Throughout the course of this capstone project, we have designed and catered to the quality requirements laid out in our architectural requirements doc. By following our coding standards, testing policy and quality assurance testing, we have been able to minimise risk to our project and ensure that we produce an application to the highest standard.

2. Unit and Integration testing policy

Our team has chosen two code testing tools, namely Jest and Cypress, to carry out various types of tests including unit, end-to-end (E2E), and integration testing. Both of these tools are capable of automating the testing process and are actively used in the project. To streamline the testing process, GitHub actions serve as the pipeline, ensuring that all tests are executed before any code is merged into the release branch.

Our target for test coverage is to have it exceed 75% by the conclusion of the project.

For unit testing, we employ Jest. This tool is responsible for thoroughly testing each service within the project, scrutinising all functions and methods for potential issues. Jest allows for automated testing and promptly notifies us of any test failures whenever there are alterations in the code or the underlying functionality it relies on.

When it comes to integration testing, both Jest and Cypress are employed. These tools facilitate the evaluation of the project's services as a whole. The nature of these tests may vary, with some being live tests and others being mocked tests. Live tests are utilised when the system has access to external resources required for testing, particularly during the development of API functions or services. In contrast, mocked tests are used to assess functionality independently of the environment and external integrations. Mocked tests are especially valuable in cases where live testing is costly or when numerous tests are needed to comprehensively evaluate a function and live resource usage becomes prohibitively expensive. In such instances, the test environment simulates the live resource, providing data and conditions to uncover and address potential bugs within functions and integrations involving external dependencies.

It's worth noting that GitHub's continuous integration system doesn't always allow for live integration tests with access to actual resources. Consequently, we often resort to using mocked tests. Additionally, there are instances where certain tests may be disabled, depending on whether they can be accommodated by the GitHub actions pipeline. If a live test cannot be executed due to the unavailability of a required resource within the actions environment, it may result in a failed merge attempt. Nevertheless, GitHub actions offers a reliable and consistent environment where all automated tests are conducted, generating logs to promptly identify any test failures resulting from code changes or other factors.

3. Coding standards

The coding standards doc can be found here for full details.

4. Quality assurance metrics and testing

The 5 necessary quality requirements that we identified were:

1. Performance
2. Availability
3. Security
4. Maintainability
5. Useability

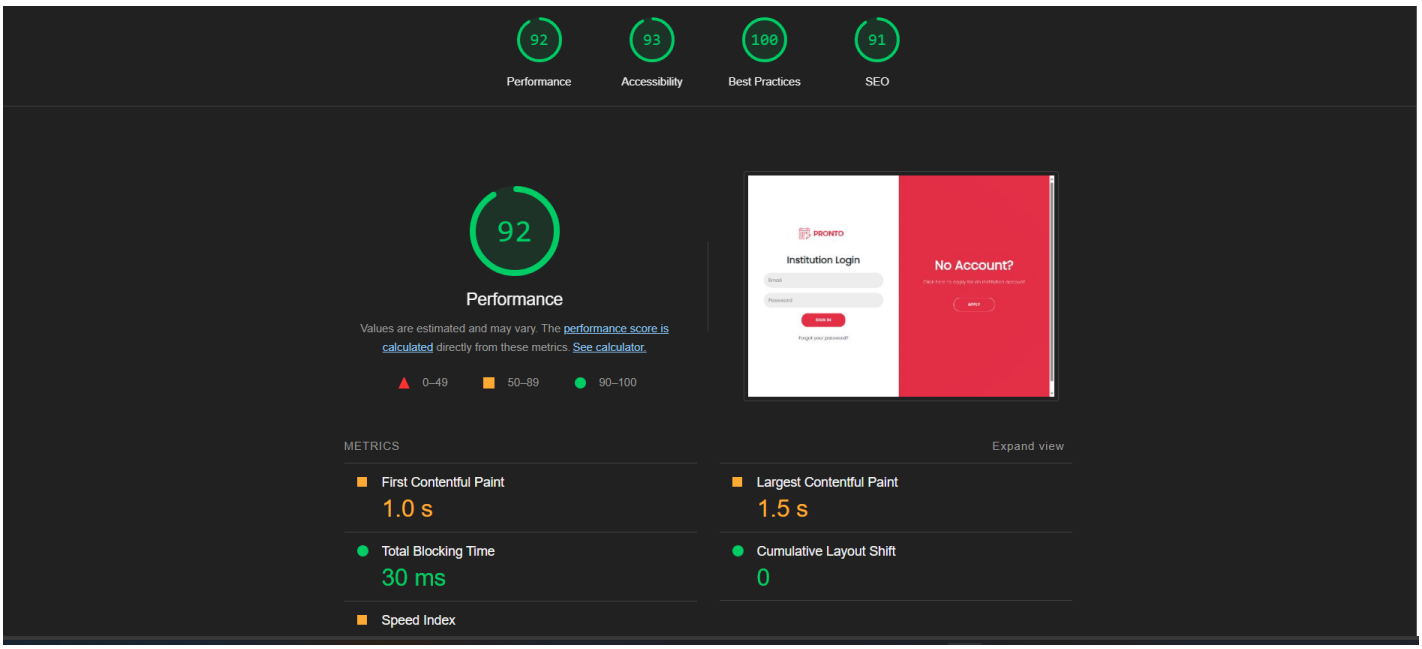
4.1 Performance

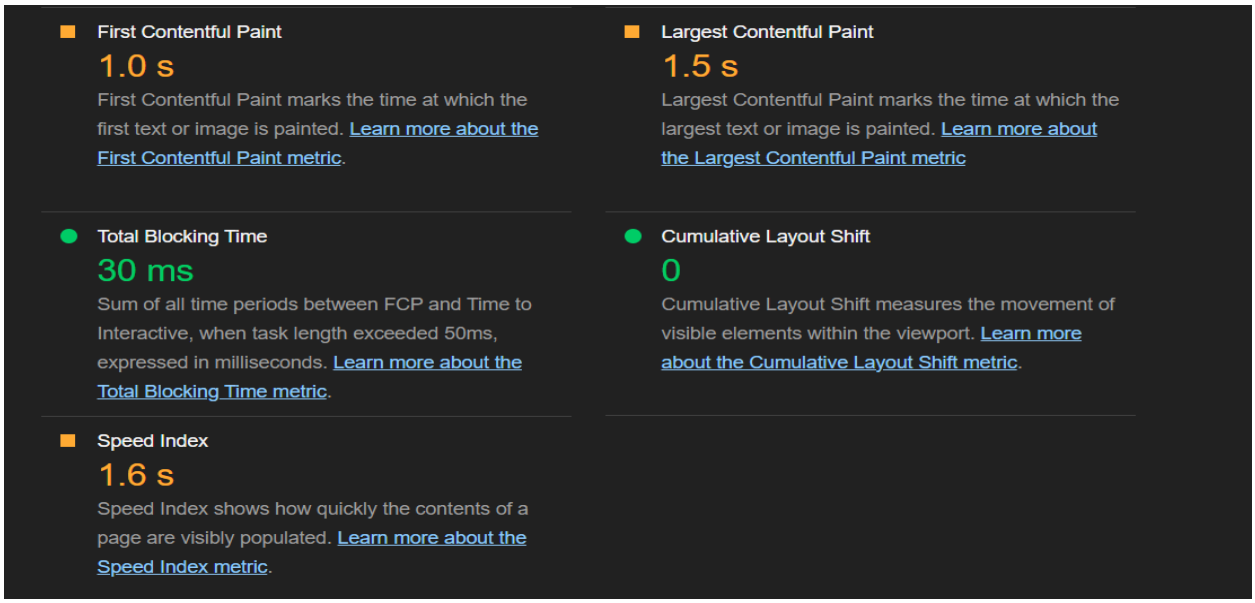
Definition: the app should be scalable and the different clients of the product must be fast and highly responsive.

How we achieve this:

- Autoscaling of the services we have utilised are able to accommodate changing traffic on a regular basis.
- End-to-End testing is conducted regularly to test server response times and ensure that users can perform each use case in an acceptable time frame.

Measurement of performance: lighthouse performance testing from google





4.2 Availability

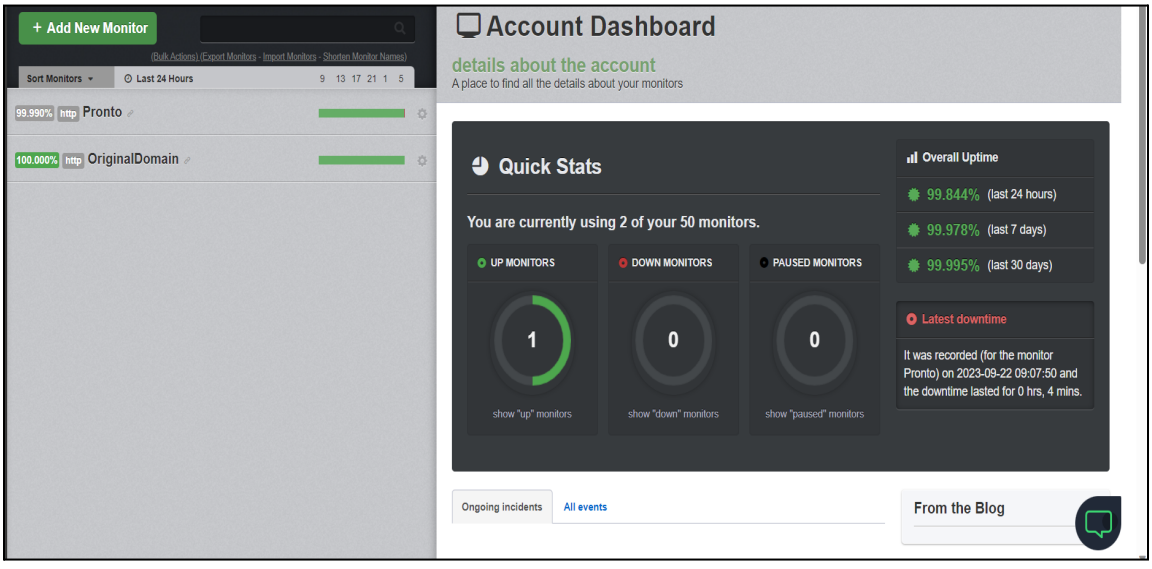
Definition: The product should always be available for all users and any unavailability communicated. The website and application should have an uptime of at least 99%.

How we achieve this:

- Services used for deployment have high reliability and uptime, and our serverless architecture aids us greatly in this regard.
- Any planned maintenance will occur outside of normal university hours or schedules.
- Planned downtime will be indicated or communicated with clear UI changes.

Measurement of availability:

We have measured our applications uptime for several weeks and can quantify that the uptime has been well above 99%, more specifically our uptime sits at over 99.9% with the only recorded downtime being due to small maintenance changes.



4.3 Security

Definition: User data should be secured at rest and in transition in order to prevent unauthorised access and comply with local laws and regulations (POPI).

How we achieve this:

- Confidentiality is ensured by employing system accounts and restricting users' access to system data. This ensures that Users cannot see other Users data.
- Role based authentication is employed to group users into user pools during sign up, which provides limited access to data modification and restricts unauthorised access to certain components of the app. By separating these users, protected routes have been used to secure each part of the application from users that do not have the right to access them.
- Integrity is maintained by encrypting data in transition by utilising HTTPS and at rest (AES 256 is used by Cognito, with salt added).

- JSX expressions {} automatically takes care of encoding HTML before rendering, thus preventing cross site scripting and injection attacks.

Measurement of security: Our security can be quantified by comparing and testing it against the OWASP top 5 list of common vulnerabilities (Broken access control, Cryptographic failures, Injection, Insecure design and Security misconfiguration). Here are some justifications for how our system compares against this list:

- Broken access control, Cryptographic failures and Security misconfiguration:
  In order to aid us in the counteraction of these three vulnerabilities, we have made use of AWS Amplify, which integrates with AWS Cognito to greatly provide and enhance security related to the authentication of users and encryption of data. By implementing role based access and making use of Cognito's identity pools and token based authentication, we have been able to secure the authentication process and protect different parts of the application based on roles. Cognito encrypts data in transit using HTTPS, ensuring that data exchanged between the client and AWS Cognito is secure and cannot be intercepted by attackers during transmission. It also employs At-Rest encryption, in which user data such as profiles are stored securely in AWS infrastructure, and encryption is managed transparently by AWS services.Tokens issued by AWS Cognito (such as JSON Web Tokens or JWTs) are typically signed and may contain encrypted claims. The encryption of token claims ensures that sensitive information within tokens is protected from tampering. On top of all these AWS security features that we have been able to utilise, we prevent security misconfiguration in a number of different ways. Our review process allows us to pick up and secure any vulnerabilities in code changes, and we regularly review and audit our amplify configurations. We log and monitor our resources to help detect and respond to security misconfigurations and potential threats, and have carried out various penetration tests to identify and fix any vulnerabilities in the system.

- Injection:
  Our system is not inherently vulnerable to any common “SQL-injection-like” attacks, as our data layer consists of Amazon S3 and DynamoDB. Amazon S3 is an object storage service and primarily stores files and data as objects in buckets. It is not a database, and it does not execute code or process queries in the same way a relational database does, therefore, traditional injection attacks like SQL injection or NoSQL injection are not applicable to it, and thanks to our secure authentication process its security risk is very limited. DynamoDB is a managed NoSQL database service, and while it is designed to be highly secure, it is still possible to encounter injection attacks in NoSQL databases, by means such as NoSQL injection. We counteract these by using parameterised queries and performing input validation and sanitisation where needed. On top of all of this, React JSX expressions {} automatically takes care of encoding HTML before rendering, thus preventing cross site scripting and injection attacks.
- Insecure design:
  Since the beginning of the project we have followed the practices of secure software development lifecycle, taking great consideration to security when designing our systems architecture and throughout the applications implementation. By combining a security oriented architecture, using services that provide highly secure features, and following all the best security practices and coding standards, we have been able to design and develop a highly secure system.

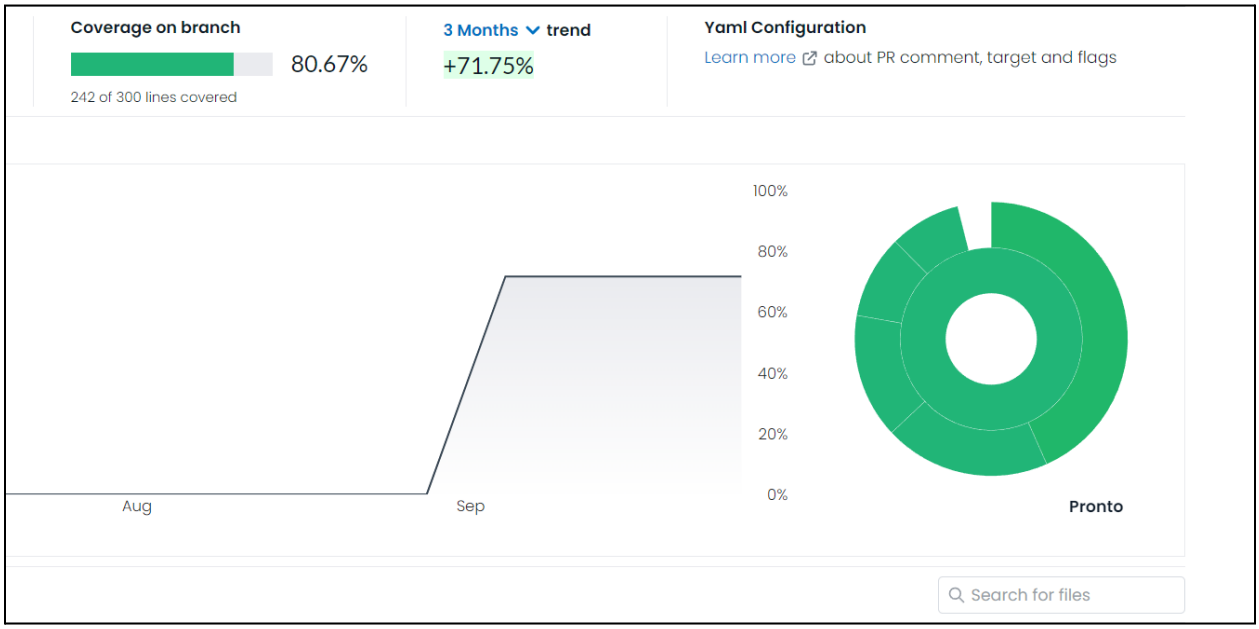
4.4 Maintainability

Definition: The software should be easily maintainable by both internal, or new external party(s), with clear concise documentation and code. Our aim to help with maintenance is to have sufficient code coverage, with as many critical tests to be carried out as possible, with at least 70% code coverage.

How we achieve this:

- Code linters and standard linting rules were applied to ensure code uniformity.
- A 3-person review procedure was used to ensure that all solutions provided are efficient and employ the best procedures.
- New features implemented have been explained thoroughly with each Pull Request, which will also be tested and evaluated by at least 3 parties prior to deployment.
- The application is serverless in it's core, which means that other than optimising implementation, there won't be a server to maintain.
- The component based architecture makes the frontend and backend code much more modular, readable and easy to understand which makes maintenance easy for both experienced and fresh programmers working on the project. It also makes unit testing easier, which enables us to continuously and automatically receive test statuses, thus allowing us to pick up and sort out any issues right away.

Measurement of maintainability:
We have thorough unit and integration tests on the backend and frontend, and have measured our code coverage at 80.6% at the time of demo 4. Team members or people new to the project can have the tests running automatically whilst working on the code, which will fail should anything be done incorrectly. Referring to the tests can also help any developer understand how the important code functions and thus greatly increase the whole code base's maintainability.



4.5 Useability

Definition: The software should be easy to use and navigation/use must be efficient, even for users who are not technologically inclined/ experienced.

How we achieve this: By carrying out useability interviews and questionnaires, we were able to receive invaluable feedback from different users, and this helped us design a user-friendly, aesthetic design.

Measurement of useability: During our usability tests we timed participants as they carried out each use case of the app and recorded the results. As highlighted in the architectural requirement, our aim is to have each feature to be located and used as intended in under 20 seconds of beginning the activity. By recording these times to carry out an activity, and observing the users behaviour we were able to analyse different features of the app and alter them as needed.