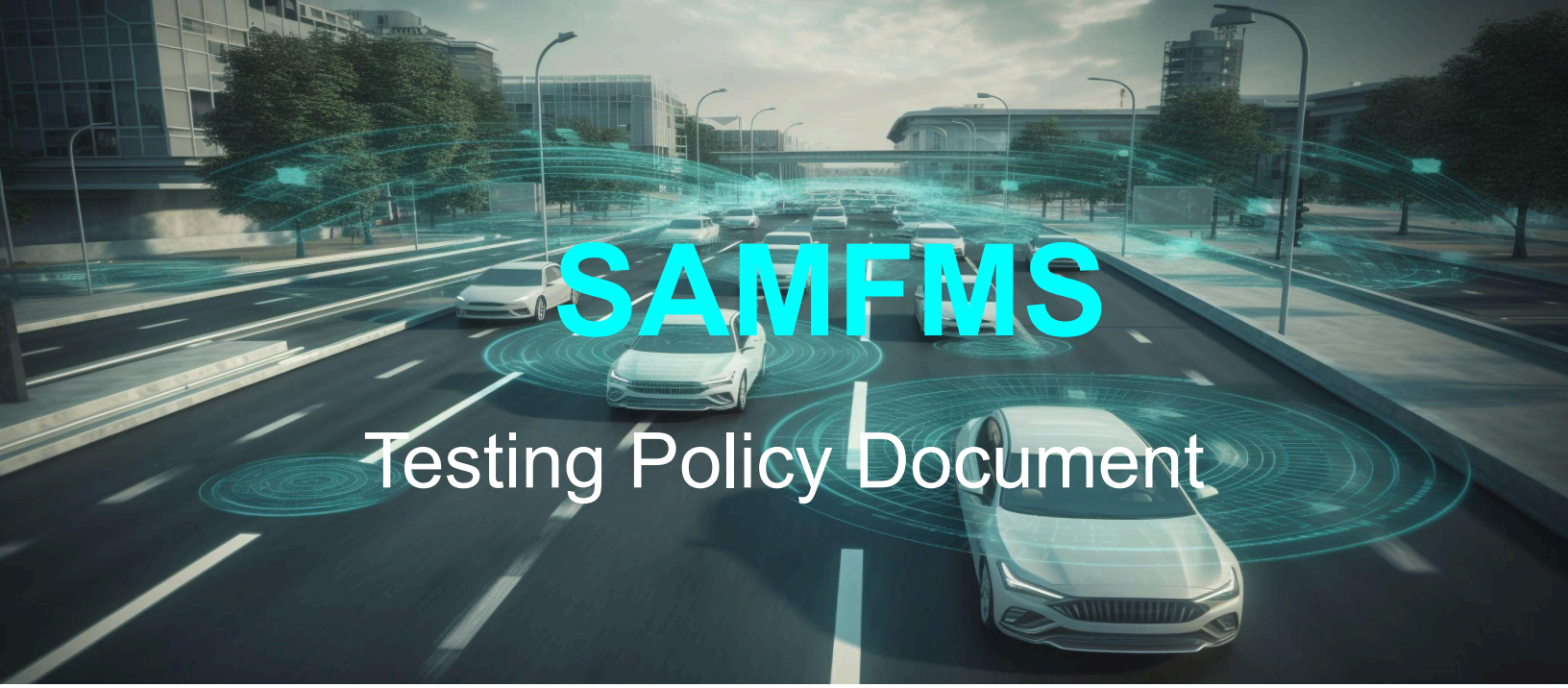


SAMFMS

Testing Policy Document

Firewall Five

Team Member	Student number
Herman Engelbrecht	u22512374
Morné van Heerden	u21482153
Laird Glanville	u22541332
Stefan Jansen van Rensburg	u22550055
Nicolaas Johan Jansen van Rensburg	u22592732

Contents

1. Introduction

This document outlines the testing policy for the SAMFMS project. It will highlight methods, technologies and CI used to automate the testing process, ensuring reliability and correctness.

2. Unit Testing

Unit tests were written for all the SBlocks' services, testing the functionality of the services they each SBlock provides. Each Sblock only required 65% coverage, as much of the code in each Sblock are models or schemas. The core services they provide are thoroughly covered and tested.

```
----------------------------------------------------------------------
TOTAL                                                7112   2498    65%
181 passed in 5.58s
```

```
Required test coverage of 40% reached. Total coverage: 68.58%
============================== 149 passed in 4.74s ==============================
```

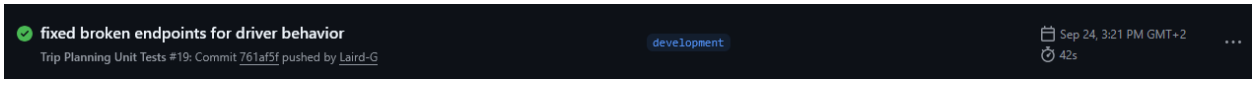
```
113  ================================ 174 passed in 9.52s ================
```

```
65  TOTAL                                                4573   1629    64%
66  142 passed in 25.82s
```

```
98  Required test coverage of 40% reached. Total coverage: 69.57%
99  ================================ 227 passed in 5.65s ================
```

Tests are written after the functionality is written, this ensures that we have a flexible approach to the functionality, able to change it when we want. The tests are highly isolated, making sure to not depend on anything.

Unit tests are run automatically using Githhub Actions. To save on actions, tests are only run when their corresponding services are updated on Github. The tests are run only on pushes to major branches; development and main.

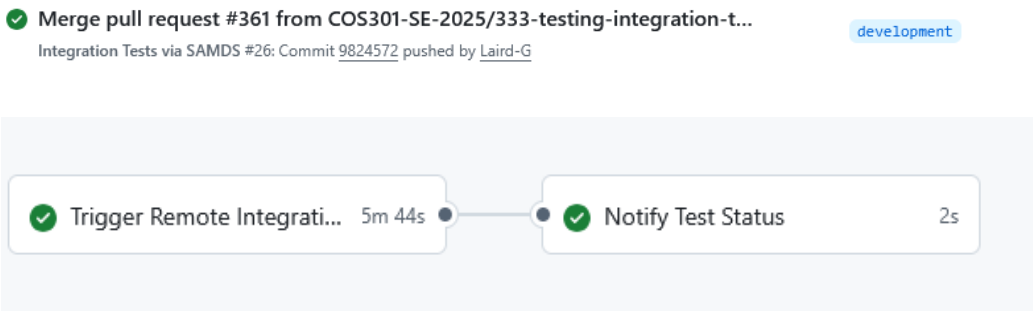


3. Integration Tests

Integration testing was used to make sure that all sub-systems were running correctly together, and their communication was correct and error free.

In this system integration tests were done as following: Whenever the server got updated, it automatically redeployed after certain conditions were met. When either the main or development branch got updated, provided they were different, all containers would go down, then integration tests were re-run with the new containers, and if they passed, the server would spin up completely. Otherwise it would not let the merge/push occur. This also doubled as regression testing, as it detected any unexpected changes in behavior caused by new code.

The automatic integration tests were run using github actions, whenever a commit was pushed or merge to major branches.

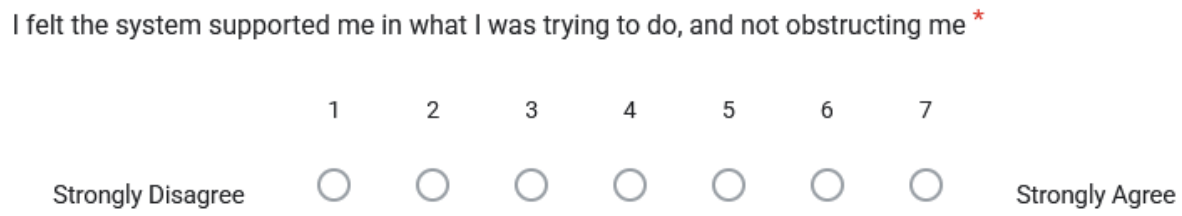


4. Non Functional Requirements Testing

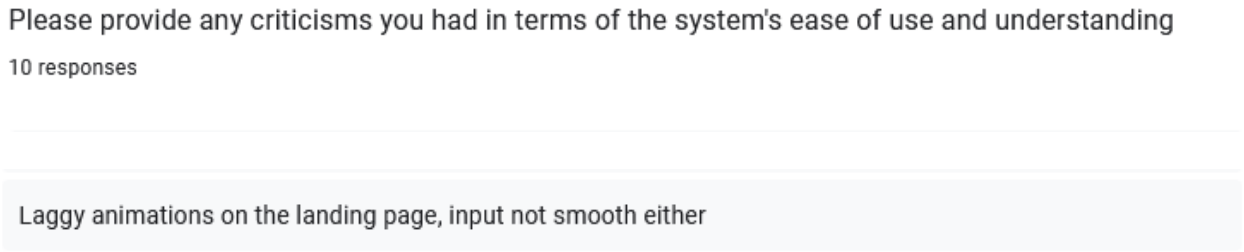
4.1 Usability Testing

Usability testing was deemed necessary because, due to the complex and large status of the app, if non-technical users had to use to for their jobs, it needs to simple, and clear to them what they're doing, as well as running on various different devices.

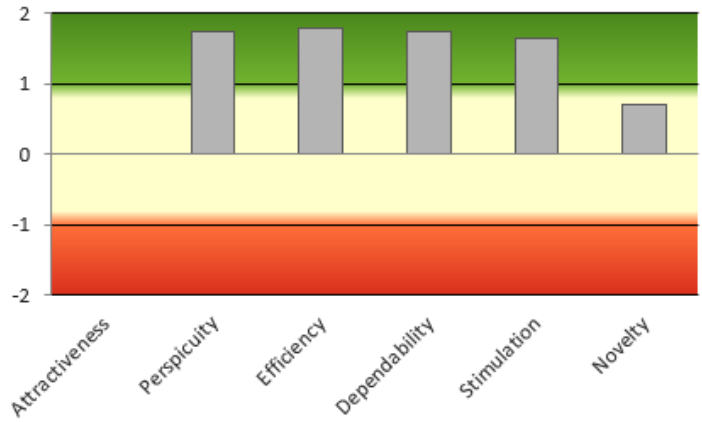
The usability testing focused on the desktop web app. We wanted to check that common tasks feel clear, simple, and efficient. We measured ease of use and speed, whether the app behaves as expected, and whether the interface gets in the way.



We used a short, structured questionnaire after the tasks were performed. It was a questionnaire that asked you to rate various aspects of the system on a scale from 1 to 7, and a dedicated criticism section to help with specifics. Questions helped capture simplicity, efficiency, supportiveness, "works as expected," and completeness, with a few that reflect more on aesthetics. Because time was limited, we did not run long interviews or think-alouds; instead we collected ratings and a short written criticism focused on ease of use and understanding.



We made use of the UEQ spreadsheet by Dr. Martin Schrepp, which helped us visualise what the general users thought of our system, and where we should make improvements.



These results showed us a generally positive reception to the layout, speed and interest our system provided. The specific criticisms also let us know what to focus on, like performance and making sure our app is better signposted for the user.

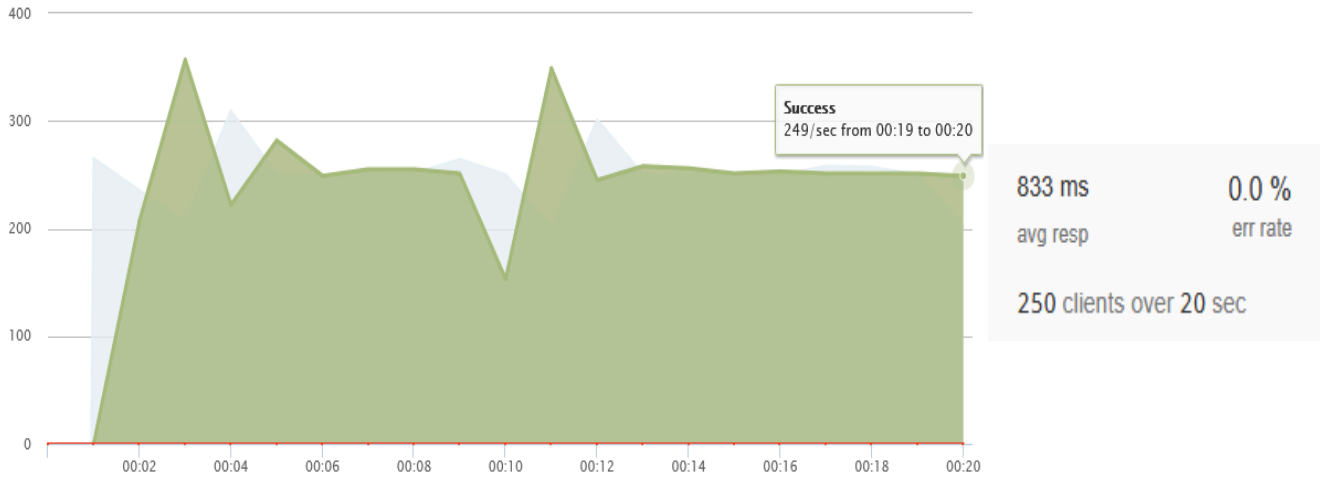
4.2 Performance Testing

In a system like this, which updates in real time with many vehicles on the road, performance is critical. Admins and managers need live information to keep track of drivers, and drivers need a system that responds quickly and predictably. Performance testing helps confirm that the system can handle expected demand without slowing down or failing under pressure.

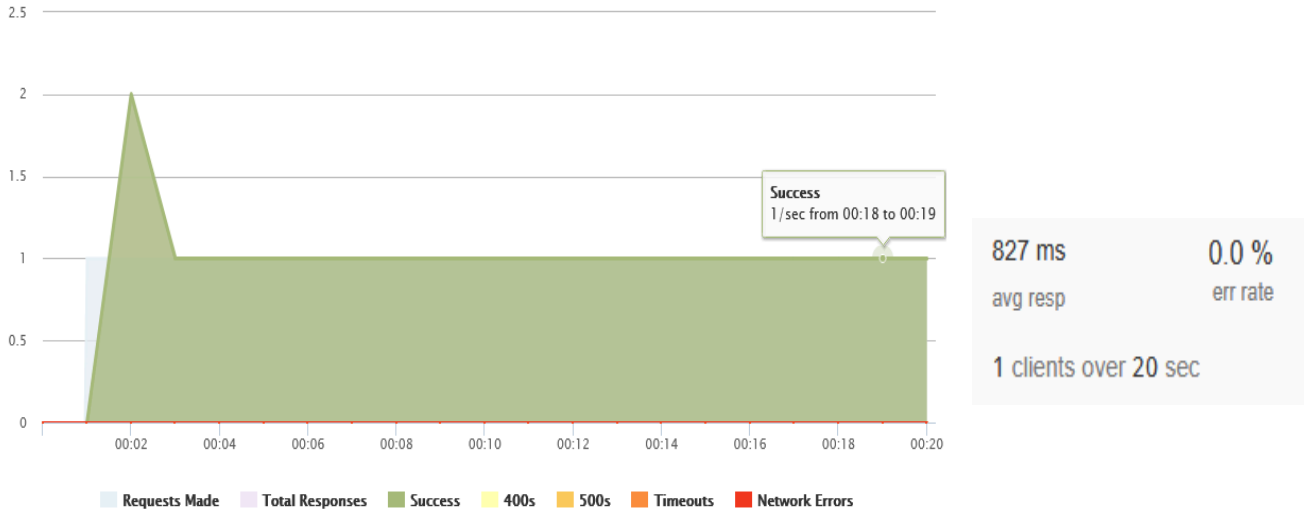
We carried out load testing to measure capacity. The system is designed for small and medium businesses; at the higher end, this could mean a fleet of up to 50 vehicles. We used Loader.io to simulate concurrent access to our endpoints, including a heavier one that returns multiple vehicles and their information. We tested with 250 requests per second and compared this to a baseline of 1 request per second. The goal was to see whether higher load introduced latency, errors, or instability.

The results showed that the system could handle 250 requests per second just as well as 1 request per second, with no increase in latency and no errors recorded. This suggests our backend can comfortably manage the expected load for our target user base, with headroom for growth.

Note: The latency numbers shown in the screenshots are affected by the test servers being outside South Africa, so the absolute values are not representative. What matters is the comparison: the latency at 250 rps was essentially the same as at 1 rps, which confirms stable performance under load.



250 Requests per second



1 Request per second.

4.3 Reliability Testing

In the system, there is a driver behavior aspect, that measures the safety of the driver, and how they perform when driving for the company. This detects aggressive accelerations and aggressive braking when performed, as well as things like going over the speed limit.

In such a system, it is necessary for the braking and the acceleration detection to be reliable and correct. We do not want the drivers to be sending too many unnecessary notifications to their managers about potentially reckless driving, when they are in fact doing nothing wrong at all. This could jeopardise their job security, even if they have not made any mistakes.

The acceleration and braking are tested using accelerometers in the drivers phone while the SAMFMS Driver App is open. To make sure that we are resistant against false positives and detect all the true positives, we put our algorithm up against a large sample of data, totalling about 9.4 million samples of driving.

We managed to get about an 85% accuracy rate which means, that out of all alerts, 85% were caused by the presence of hard braking or acceleration.

```
ACCURACY TEST RESULTS
=====================

Processing Statistics:
   Total Samples:          9,473,282
   Processed Samples:      9,473,282
   Processing Time:        4.67 hrs

Confusion Matrix:
   True Positives (TP):    11,904
   True Negatives (TN):    7,004,765
   False Positives (FP):   4,032
   False Negatives (FN):   1,222,881

Error Rate Analysis:
   FN over TN:             17.46%
   FP over TP:             33.87%

Performance Metrics:
   Accuracy:               85.12%
   Precision:              74.70%

Violation Type Performance:
   Acceleration:
     Accuracy:  82.7%
     Precision: 73.5%
   Braking:
     Accuracy:  87.8%
     Precision: 75.3%
```