

# SOFTWARE SHARKS

---

## Testing Manual

for

## NINSHIKI

Version 1.0.0 - Demo #5 Draft

---

A guide to the testing process used in the development of the Ninshiki system, including the test procedures, tools and frameworks, as well as the functional test cases and the history of various functional and nonfunctional requirement testing.

Compiled By

### The Software Sharks Team

Coetzer MJ

Bester TJ

Orrie O

Lew J

Matodzi MC

Department of Computer Science

The University of Pretoria



UNIVERSITEIT VAN PRETORIA  
UNIVERSITY OF PRETORIA  
YUNIBESITHI YA PRETORIA

Denkleiers • Leading Minds • Dikgopolo tša Dihalefi

For

**Bramhope International School of Innovation**

COS 301 Team: Software Sharks

Department of Computer Science, University of Pretoria

<https://github.com/OrishaOrrie/SoftwareSharks>

This testing manual document was drafted under the supervision of involved lecturers according to the assessment guidelines of the final year Computer Science module: COS 301 - Software Engineering, presented by the Department of Computer Science in the faculty of Engineering, Built Environment and Information Technology at the University of Pretoria during the first and second semesters of the year 2018.

First release, September 2018

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Testing Process</b>	<b>3</b>
2.1	Functional . . . . .	3
2.1.1	General Testing Process . . . . .	3
2.1.2	Application Testing Process . . . . .	3
2.2	Non-Functional . . . . .	4
2.2.1	Performance . . . . .	4
2.2.2	Security . . . . .	6
2.2.3	User Interface . . . . .	6
2.2.4	Image Prediction . . . . .	6
<b>3</b>	<b>Testing Tools</b>	<b>7</b>
3.1	Functional . . . . .	7
3.1.1	Application Testing Tools . . . . .	7
3.2	Non-Functional . . . . .	8
3.2.1	Performance . . . . .	8
3.2.2	Security . . . . .	9
3.2.3	User Interface . . . . .	11
3.2.4	Image Prediction . . . . .	12
<b>4</b>	<b>Test Cases</b>	<b>13</b>
4.1	Functional . . . . .	13
4.1.1	Application Test Cases . . . . .	13
<b>5</b>	<b>History</b>	<b>14</b>
5.1	Functional . . . . .	14
5.2	Non-Functional . . . . .	16
5.2.1	Performance . . . . .	16
5.2.2	Security . . . . .	19
5.2.3	User Interface . . . . .	19
5.2.4	Image Prediction . . . . .	21

## 1 Introduction

This is a document describing all the test procedures used throughout the development of the Ninshiki system. First, the specific testing procedures will be discussed for each functional subsystem, as well as for each non-functional requirements. Then, the tools and frameworks used to carry out these procedures will be elaborated. Thirdly, the test cases/functions used to check that each requirement is met will be defined. Finally, the history and location of test logs of each subsystem will be discussed.

## 2 Testing Process

The following section discusses the processes undertaken to test the various functional and nonfunctional requirements of each subsystem. The details of each test is discussed, including the method and frequency of each test.

### 2.1 Functional

#### 2.1.1 General Testing Process

The development of the system follows the principles of Test-Driven Development. The general testing process is as follows:

1. For each use case, a test function is written that acts as a user completing a task.
2. Initially, all test functions fail. If they don't, they are rewritten so that they fail.
3. Starting from the highest priority use cases, the functional classes and programs that solve each use case are implemented, so that the respective test functions succeed.
4. This is repeated until all functions are implemented, all use cases are solved, and all test functions pass successfully.

This procedure is followed when writing test functions and test cases for each subsystem.

#### 2.1.2 Application Testing Process

Since the web and mobile application is built using the Angular framework (the Ionic framework used for mobile is based on Angular), the code-base for each subsystem is very similar, differing only in the way images are captured or selected from gallery. For this reason, this section will encapsulate the functional testing procedure for both subsystems.

The functionality of the applications are tested against the system use cases, with one set of functions written for each use case. These test functions comprise the end-to-end tests, and are run using Protractor and written with Jasmine. The test cases are described in section 3. Complete, automated end-to-end tests are performed after the addition of a new feature or after the completion of a change to the app.

Each use case is also assigned to a component of the app. This means that each component needs to be unit tested. Unit testing is done to ensure that the internal structure of the application serves its purpose, handles errors correctly, and has no flaws. A set of unit test functions are written for each component in an isolated manner. These tests are run via Karma and written with Jasmine. Automated unit tests are run after any changes are made to a single component.

The full set of unit tests and end-to-end tests are run when a new set of code is pushed to the GitHub repository master branch. This is done automatically via Travis CI, which executes a script on a remote PC and sends a report to the developers upon completion. This ensures that both kinds of tests are done for the application without the need for personal resources. It also prevents faulty software from being deployed.

## 2.2 Non-Functional

### 2.2.1 Performance

This section describes the process of testing the performance of the various subsystems.

**Web App Performance Testing** The performance of the web application is tested to check whether the response time is fast enough so that the user does not get frustrated, but rather can use the web app efficiently. Two tools are used to test the performance of the web app: the one runs tests on the local browser, the other runs tests in a hosted environment. The second tool provides a generally more accurate and consistent report, since local testing can be influenced by things such as anti-virus scanners, browser extensions, etc.

Google Lighthouse Audit is used to test response time locally. The Performance category of the audit provides different metrics such as first meaningful paint, first CPU idle, and time to interactive. All of these metrics are used in conjunction to determine whether the overall site response time meets the performance requirements. If the report indicates

that at least four out of six metrics are good (green), then the performance test has been passed.

WebPagetest is used to test response time from a hosted environment. It runs performance tests much like Lighthouse, but the location, browser, and bandwidth quality of the test can be specified. WebPagetest also specifies the performance details of the test in more depth. It also provides grades that indicate whether a set of metrics is at a good level. If the report indicates that the first byte time received a "B" or higher, then the performance test has been passed.

Both tools are used to perform tests each time the web app is deployed and hosted on Firebase, so that it can be tested as if it were an actual website. The Lighthouse Audit is part of the Travis continuous integration script.

**Mobile App Performance Testing** Much like the web app, the response time of operations in the mobile app needs to be tested to see that it aids, not deters usability. Two different tools are also used for the same purpose: one tool is used to test performance locally and the other is used to test in a hosted environment.

Lighthouse is used once again to test the performance of the mobile app locally. This is possible because the app is built with Ionic, which uses the web-based Angular framework. Therefore, the app can be treated as a website before it is packaged as a native mobile app. Testing is done the same way as with the web app Lighthouse testing. The Lighthouse Audit is conducted every time the app is to be deployed, just before the app is packaged into a native app.

Once the packaged native app is produced (as an APK file), a second tool called Firebase Test Lab is used to check app performance on eight different kinds of Android phones, with different OS versions and different screen sizes. The Test Lab produces various performance-related time-series graphs. These graphs indicate how RAM and CPU usage vary in different sections of the app. This is useful in optimizing the performance of different operations of the app. These tests are performed every time an APK is built, i.e. every production release.

Once the app is deployed and used, Firebase collects various analytics related to performance and crashes. The Firebase dashboard displays

these performance metrics and crash reports in a neat, easy to read format.

These three tools are used to ensure that performance requirements are met at three different stages of the app development life-cycle.

### **2.2.2 Security**

In-house penetration was performed by utilizing the SQLMap and NMap command line tools. Identifying malformed URI's, potential injections and open access points. These tools were run on in a linux environment against the hosted server.

### **2.2.3 User Interface**

The user interface is one of the most important aspects of an application. The look and feel is what will make users want to use the application. Since the application has been created for everyday use by warehouse workers, it should be simple to use with neutral colours.

The two main functions are the image prediction and weight analysis since those what the workers will be using the most often. Both of these have a very simple interface as well as instructions on how to use them in order to make the interface even simpler.

In order to test these features, usability tests were done throughout the life-cycle of the project. In this usability test, a few aspects such as efficiency, satisfaction as well as productivity were tested.

### **2.2.4 Image Prediction**

This section describes the testing process for the image classifier model. This tests the accuracy of a model by feeding it a set of test images, at least one for each class, and comparing the predicted label of that image to the actual class label. Correct predictions are ones that predict the actual class as the most likely. Top 8 predictions are ones that predict the actual class as within the top 8 most likely classes.

A Python script is used to run the entire set of predictions, while the results are recorded manually in a spreadsheet. This process is completed every time a new model is built.

## 3 Testing Tools

Various testing frameworks and tools are used to automate unit and end-to-end tests, as well as visualize the tests for non-functional requirements. This section describes each tool used, as well how they are configured and why they were chosen.

### 3.1 Functional

This section describes the tools used when testing that the subsystems meet the use cases.

#### 3.1.1 Application Testing Tools

As mentioned in section 1, there are different tools used for each kind of testing. Each tool is described below.

**Jasmine** Jasmine is a testing framework that allows for test functions to be written in JavaScript/TypeScript that supports Test-Driven Development. Jasmine allows for functions to be written in a low-level, easy-to-understand format. Multiple JavaScript test scripts can be run by a single Jasmine HTML file. The actual test functions for both the unit and end-to-end tests for the mobile and web applications are written using Jasmine. Jasmine test files are setup and configured automatically upon creation of an Angular project. Unit tests for each component and service can be found in the respective component spec.ts file, which is found in the component folder. They are executed by running the command 'ng test'.

Jasmine was selected to write test functions because it is the recommended testing framework for Angular.

**Karma** Karma is an automated test runner that integrates with Angular and Jasmine. It runs a separate browser with all Jasmine unit tests running. Karma automatically re-runs all Jasmine test functions each time a change is made to the development environment. Karma is also pre-configured upon creation of an Angular project.

Karma was also selected due to it being the recommended testing tool for Angular. The combination of Jasmine and Karma aids Test-Driven Development, and makes the unit testing process simple and efficient.



**Protractor** Protractor is an end-to-end testing framework that runs test functions on a real browser automatically. This means that tests are performed as if they were done in a real environment. Protractor makes end-to-end testing easy by automating things such as waiting, as it smartly detects when changes to a page have been made. Protractor can easily be configured with an Angular project and often comes pre-configured.

Protractor was selected due to its compatibility with Jasmine and Angular in general. It is also useful for end-to-end testing event-driven applications.

**Travis CI** Travis CI is a continuous integration service that automates integration and deployment processes. Travis executes a script every time a push is made to the Master branch of the GitHub repository. The script may execute commands that assist in various processes such as unit testing, end-to-end testing, and deployment. This script file can be found and edited in the root directory of the repository.

Travis was selected due to its ability to automatically run the full suite of test cases by executing the appropriate Karma and Protractor commands. It can also be setup so that if no major faults are discovered, it automatically deploys the system (in the case of this project, it would deploy the web app to Firebase for hosting).

## 3.2 Non-Functional

### 3.2.1 Performance

**Google Lighthouse Audit** Lighthouse is a tool available as a Node module with a command line interface and as part of Google Chrome browser Developer Tools. It performs an audit on any given website via a URL, and allows for the mobile or desktop version of the website to be tested. The audit produces an HTML report with scores for five different categories, namely: Performance, PWA (Progressive Web App), SEO (Search Engine Optimization), Best Practices, and Accessibility. Not only do these scores give a good indication of the performance of the web app, the report also produces guidelines and optimization tips to help improve each of the five aspects. Lighthouse was selected due to its easy to understand reports, helpful guidelines, and ability to be used with Travis-CI.

A Lighthouse Audit can be run by entering the Developer Tools on selected browsers, navigating to the Audit tab, and specifying a few prefer-

ences. Alternatively, the audit can be run from the command line by using "lighthouse <url>", provided the Lighthouse npm package is installed.

**WebPagetest** WebPagetest is a website that performs a performance review on a given website via the URL. The location, bandwidth quality, and browser on which the test is run can be specified, and then the test is performed in that specific environment. The tool produces a detailed report, including waterfall graphs, which help developers understand the process of the website loading onto the user's browser. The report also gives grades to various performance attributes of the website, so that these attributes can be identified and improved upon. WebPagetest was selected as a tool due to it being recommended by the Lighthouse documentation, and because it produces more consistent and accurate test results than Lighthouse.

WebPagetest can be accessed from <https://www.webpagetest.org/> and simply requires the URL of the website to be input.

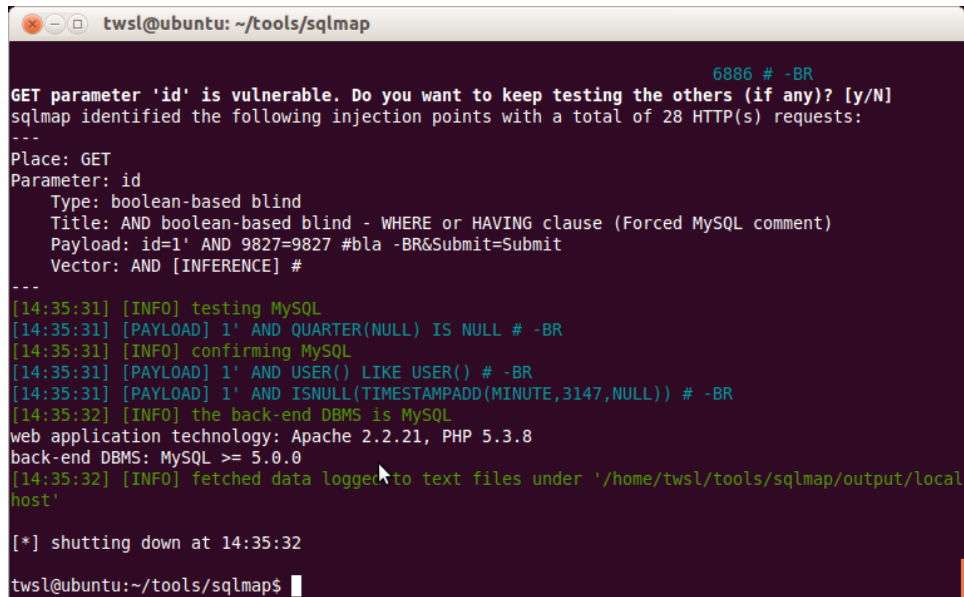
**Firebase Test Lab** Firebase is a platform that provides various services that power up the development and deployment of mobile and web applications, such as performance analysis on live app sessions. One of these services used in this project is called Test Lab, which simply requires an APK to be uploaded in order for a mobile application to be tested. Test Lab automatically runs what it calls Robo tests on real devices, which runs the app, programmatically presses all the buttons on the app in sequence, and produces performance graphs and a video recording of the test. This helps developers understand the performance of the app on various device models. Test Lab was chosen because Firebase was already in use for the deployment of the app, and because it makes performance testing on various device models simple and efficient, without having the need to obtain an infrastructure or find users with different devices.

Test Lab simply requires a Firebase project to be created and for any APK to be uploaded. The only settings that need to be configured are the number and type of devices that are to run the test.

### 3.2.2 Security

**In-House Penetration Testing** The security of the web application end point was tested via several brief tests, detailed below, aimed at gaining unintended access to files or functions.

- SQLMap: A SQLMap script was run to identify possible injection parameters in malformed URI's
- NMap: A NMap script was run against the server to identify possible open ports and unwanted access points



```
twsl@ubuntu: ~/tools/sqlmap

6886 # -BR
GET parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [y/N]
sqlmap identified the following injection points with a total of 28 HTTP(s) requests:
---
Place: GET
Parameter: id
  Type: boolean-based blind
  Title: AND boolean-based blind - WHERE or HAVING clause (Forced MySQL comment)
  Payload: id=1' AND 9827=9827 #bla -BR&Submit=Submit
  Vector: AND [INFERENCE] #
---
[14:35:31] [INFO] testing MySQL
[14:35:31] [PAYLOAD] 1' AND QUARTER(NULL) IS NULL # -BR
[14:35:31] [INFO] confirming MySQL
[14:35:31] [PAYLOAD] 1' AND USER() LIKE USER() # -BR
[14:35:31] [PAYLOAD] 1' AND ISNULL(TIMESTAMPADD(MINUTE,3147,NULL)) # -BR
[14:35:32] [INFO] the back-end DBMS is MySQL
web application technology: Apache 2.2.21, PHP 5.3.8
back-end DBMS: MySQL >= 5.0.0
[14:35:32] [INFO] fetched data logged to text files under '/home/twsl/tools/sqlmap/output/local
host'

[*] shutting down at 14:35:32
twsl@ubuntu:~/tools/sqlmap$
```

Figure 1: SQLMap Report

```
root@debian:/home/master# nmap 192.168.100.1-5
Starting Nmap 6.47 ( http://nmap.org ) at 2016-11-10 18:29 CST
Nmap scan report for 192.168.100.1
Host is up (0.00092s latency).
Not shown: 994 closed ports
PORT      STATE      SERVICE
22/tcp    filtered  ssh
23/tcp    filtered  telnet
53/tcp    open       domain
80/tcp    open       http
49152/tcp open       unknown
49153/tcp open       unknown
MAC Address: 2C:AB:00:F7:75:4E (Unknown)

Nmap scan report for 192.168.100.5
Host is up (0.0035s latency).
All 1000 scanned ports on 192.168.100.5 are closed
MAC Address: D0:C1:B1:8F:86:06 (Samsung Electronics Co.)

Nmap scan report for 192.168.100.3
Host is up (0.000015s latency).
Not shown: 999 closed ports
PORT      STATE      SERVICE
111/tcp   open       rpcbind

Nmap done: 5 IP addresses (3 hosts up) scanned in 100.05 seconds
root@debian:/home/master#
```

Figure 2: NMap Report

### 3.2.3 User Interface

To test the user interface, Usability testing was done where a Google form was created and a series of questions relating to the user interface were asked. These tests were done on people from different backgrounds and skill levels.

In order to better the user interface even further, a meeting was held, with a lecturer at the University of Pretoria, who specializes in User interface design as well as User experience. He helped us to properly place different features of both the web app as well as the mobile app in correct places so that users will find it easier to access these features.

From these results, an analysis was done and changes to the user interface were completed. Once these changes were completed, another usability test was done and this cycle continued throughout the life-cycle of the project.

### **3.2.4 Image Prediction**

No special test tools are used to test the accuracy of a model. The model is tested by running a script that runs the Predict method on each test image.

## 4 Test Cases

The following section describes the test cases used to check whether each aspect of the requirements are met. The majority of these formal test cases are for the unit and end-to-end test code for the functionality testing.

### 4.1 Functional

This section describes the individual test functions for each of the use cases. The locations in the GitHub repository for the test cases of each subsystem are also indicated.

#### 4.1.1 Application Test Cases

The following use cases each have associated end-to-end test functions. These tests are all executed in sequence by running the command 'ng e2e' in the ss-imagerec-web folder. The tests functions can be found here:

<https://github.com/OrishaOrrie/SoftwareSharks/blob/master/Application%20Source/ss-imagerec-webapp/e2e/app.e2e-spec.ts>

1. Use Case: Capture an Image

- (a) Test Function: If the "Webcam" button is clicked, then check if the webcam is opened and check if the video capture is displayed on the page.
- (b) Test Function: If the "File Select" button is clicked, then check if the file explorer has been opened and check if a selected image is displayed on the page.

2. Use Case: Submit Image

- (a) Test Function: If an image is displayed, then it must have been captured or selected, so check if the "Submit" button is displayed.
- (b) Test Function: If the "Submit" button is clicked, then check if the text on the button changes to "Predicting..."

3. Use Case: Receive/View Results

- (a) Test Function: If an image has been submitted, then check whether the window scrolls down to the results table.
- (b) Test Function: If an image has been submitted, then check whether entries exist in the results table.

- (c) Test Function: If an image has been submitted, then check whether an entry contains a link that can be clicked, opening a page from the Bramhope catalogue in a new tab.

#### 4. Use Case: Weight Analysis

- (a) Test Function: If an input field is touched or dirtied, check that the "Item Amount" check is updated dynamically.
- (b) Test Function: If an input field is made blank, check that the status bar is updated to display an appropriate message.
- (c) Test Function: If a negative value is input, check that the status bar is updated to display an appropriate message.
- (d) Test Function: If the input weight for an empty bucket is greater than the input weight for a filled bucket, check that the status bar is updated to display an appropriate message.
- (e) Test Function: If the input tuple for (single\_item, empty\_bucket, filled\_bucket) is (5, 5, 20), check that the status bar is updated to display "Amount of Items: 3".

#### 5. Use Case: Contact Company

- (a) Test Function: If an input field is touched or dirtied, then check whether validation is done on the field dynamically.
- (b) Test Function: Check whether invalid email field values are rejected.
- (c) Test Function: If all input fields are valid, then check whether the "Submit" button is enabled.
- (d) Test Function: If the "Submit" button is clicked, then check whether a loading spinner is displayed.
- (e) Test Function: If the "Submit" button is clicked, then check whether the HTTP POST function is called.
- (f) Test Function: If a query has been submitted, then check whether the server response is a status 200 code.

## 5 History

This section contains the locations of test logs for each of the various testing processes conducted. These locations are in the form of external links or the links to the GitHub repository.

### 5.1 Functional

This section displays the repository location and examples of test logs for the different kinds of functional tests.

As described in the previous sections, the Karma test runner is used to run unit tests for the application programs. Tests reports can be found here: [https://github.com/OrishaOrrie/SoftwareSharks/tree/master/Application%20Source/ss-imagerec-webapp/test\\_reports/karma\\_tests](https://github.com/OrishaOrrie/SoftwareSharks/tree/master/Application%20Source/ss-imagerec-webapp/test_reports/karma_tests) (note that the repository will need to be cloned to view these reports). An example Karma test report can be found below in Figure 1.



Figure 3: Karma Test Report

Protractor is used to run end-to-end tests for the application programs. Test reports cannot be saved, as reporting is done in the terminal window. An example test report can be found below in Figure 2.

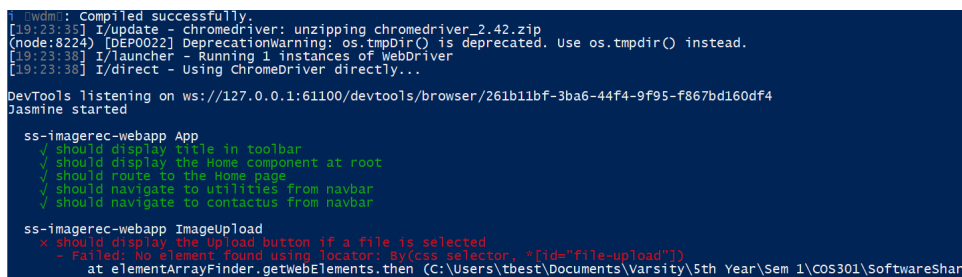


Figure 4: Protractor Report



Travis CI keeps a log of all script results for each build attempt. The build history can be found here:

<https://travis-ci.org/OrishaOrrie/SoftwareSharks/builds>. An example build report can be found below in Figure 3.

```

4524 $ ng test --watch=false
4525 10% building modules 1/1 modules 0 active(node:6445) DeprecationWarning: Tapable.plugin is deprecated. Use new API on `.hooks`
instead
4526 10 00 2018 10:20:02.775:INFO [karma] Front-end scripts not present. Compiling...
4527 3 69% building modules 535/540 modules 5 active _ode_modules/hash.js/lib/hash/sha/512.js 10 00 2018 10:20:02.275:INFO
[karma] Karma v2.0.0 server started at http://0.0.0.0:9876/
4528 10 00 2018 10:20:02.279:INFO [launcher] Launching browser Chrome with unlimited concurrency
4529 10 00 2018 10:20:02.279:INFO [launcher] Starting browser Chrome
4530 10 00 2018 10:20:11.237:INFO [Chrome 69.0.3497 (Linux 0.0.0)] Connected on socket BgvQ0t-omkpt9ot1AAAA with id 945096
4531 Chrome 69.0.3497 (Linux 0.0.0): Executed 0 of 28 SUCCESS (0 secs / 0 secs)
4532 10 00 2018 10:20:15.885:WARN [web-server]:404: /assets/tukslogo.png
4533 e 69.0.3497 (Linux 0.0.0): Executed 1 of 28 SUCCESS (0 secs / 0.462 secs)
4534 10 00 2018 10:20:16.240:WARN [web-server]:404: /assets/tukslogo.png
4535 e 69.0.3497 (Linux 0.0.0): Executed 2 of 28 SUCCESS (0 secs / 0.799 secs)
4536 e 69.0.3497 (Linux 0.0.0): Executed 3 of 28 SUCCESS (0 secs / 1.131 secs)
4537 e 69.0.3497 (Linux 0.0.0): Executed 4 of 28 SUCCESS (0 secs / 1.425 secs)
4538 e 69.0.3497 (Linux 0.0.0): Executed 5 of 28 SUCCESS (0 secs / 1.689 secs)
4539 e 69.0.3497 (Linux 0.0.0): Executed 6 of 28 SUCCESS (0 secs / 1.96 secs)
4540 e 69.0.3497 (Linux 0.0.0): Executed 7 of 28 SUCCESS (0 secs / 2.282 secs)
4541 e 69.0.3497 (Linux 0.0.0): Executed 8 of 28 SUCCESS (0 secs / 2.556 secs)
4542 e 69.0.3497 (Linux 0.0.0): Executed 9 of 28 SUCCESS (0 secs / 2.866 secs)
4543 e 69.0.3497 (Linux 0.0.0): Executed 10 of 28 SUCCESS (0 secs / 3.147 secs)
4544 e 69.0.3497 (Linux 0.0.0): Executed 11 of 28 SUCCESS (0 secs / 3.373 secs)
4545 e 69.0.3497 (Linux 0.0.0): Executed 12 of 28 SUCCESS (0 secs / 3.611 secs)
4546 LOG: 'Checking if the model is ready'
4547 'Checking if the model is ready'
4548 Chrome 69.0.3497 (Linux 0.0.0): Executed 12 of 28 SUCCESS (0 secs / 3.611 secs)
4549 LOG: 'Checking if the model is ready'
  
```

Figure 5: Travis CI Report

## 5.2 Non-Functional

### 5.2.1 Performance

**Web App** The history of Lighthouse Audit reports can be found in the GitHub repository here: [https://github.com/OrishaOrrie/SoftwareSharks/tree/master/Application%20Source/ss-imagerec-webapp/test\\_reports/lighthouse\\_report](https://github.com/OrishaOrrie/SoftwareSharks/tree/master/Application%20Source/ss-imagerec-webapp/test_reports/lighthouse_report)

An example Lighthouse Audit report can be found in Figure 4 below.

```
4524 $ ng test --watch=false
4525 10% building modules 1/1 modules 0 active(node:6445) DeprecationWarning: Tapable.plugin is deprecated. Use new API on `.hooks`
      instead
4526 10 00 2018 10:20:02.190:INFO [karma] Front-end scripts not present. Compiling...
4527 3 69% building modules 535/540 modules 5 active _code_modules/hash.js/lib/hash/sha/512.js 10 00 2018 10:20:02.275:INFO
[karma] Karma v2.0.0 server started at http://0.0.0.0:9876/
4528 10 00 2018 10:20:02.279:INFO [launcher] Launching browser Chrome with unlimited concurrency
4529 10 00 2018 10:20:02.279:INFO [launcher] Starting browser Chrome
4530 10 00 2018 10:20:11.237:INFO [Chrome 69.0.3497 (Linux 0.0.0)] Connected on socket BgvQ0t-omkpT9ot1AAAA with id 945096
4531 Chrome 69.0.3497 (Linux 0.0.0): Executed 0 of 28 SUCCESS (0 secs / 0 secs)
4532 10 00 2018 10:20:15.885:WARN [web-server]:404: /assets/tukslogo.png
4533 e 69.0.3497 (Linux 0.0.0): Executed 1 of 28 SUCCESS (0 secs / 0.462 secs)
4534 10 00 2018 10:20:16.240:WARN [web-server]:404: /assets/tukslogo.png
4535 e 69.0.3497 (Linux 0.0.0): Executed 2 of 28 SUCCESS (0 secs / 0.799 secs)
4536 e 69.0.3497 (Linux 0.0.0): Executed 3 of 28 SUCCESS (0 secs / 1.131 secs)
4537 e 69.0.3497 (Linux 0.0.0): Executed 4 of 28 SUCCESS (0 secs / 1.425 secs)
4538 e 69.0.3497 (Linux 0.0.0): Executed 5 of 28 SUCCESS (0 secs / 1.689 secs)
4539 e 69.0.3497 (Linux 0.0.0): Executed 6 of 28 SUCCESS (0 secs / 1.96 secs)
4540 e 69.0.3497 (Linux 0.0.0): Executed 7 of 28 SUCCESS (0 secs / 2.282 secs)
4541 e 69.0.3497 (Linux 0.0.0): Executed 8 of 28 SUCCESS (0 secs / 2.556 secs)
4542 e 69.0.3497 (Linux 0.0.0): Executed 9 of 28 SUCCESS (0 secs / 2.866 secs)
4543 e 69.0.3497 (Linux 0.0.0): Executed 10 of 28 SUCCESS (0 secs / 3.147 secs)
4544 e 69.0.3497 (Linux 0.0.0): Executed 11 of 28 SUCCESS (0 secs / 3.373 secs)
4545 e 69.0.3497 (Linux 0.0.0): Executed 12 of 28 SUCCESS (0 secs / 3.611 secs)
4546 LOG: 'Checking if the model is ready'
4547 'Checking if the model is ready'
4548 Chrome 69.0.3497 (Linux 0.0.0): Executed 12 of 28 SUCCESS (0 secs / 3.611 secs)
4549 LOG: 'Checking if the model is ready'
```

Figure 6: Lighthouse Audit Report

The WebPagetest reports are displayed on the website itself and cannot be exported. An example report can be found in Figure 5 below.

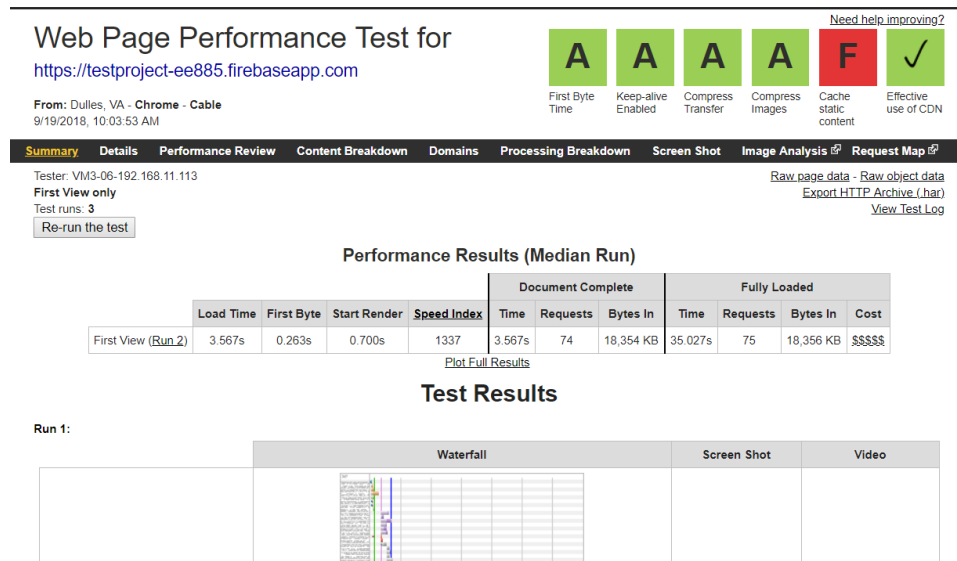


Figure 7: WebPagetest Report

**Mobile App** The Firebase Test Lab reports are displayed on the Firebase console and cannot be exported either. A few screenshots of the Test Lab report and the general performance analytics dashboard can be found in Figures 6 and 7 below.

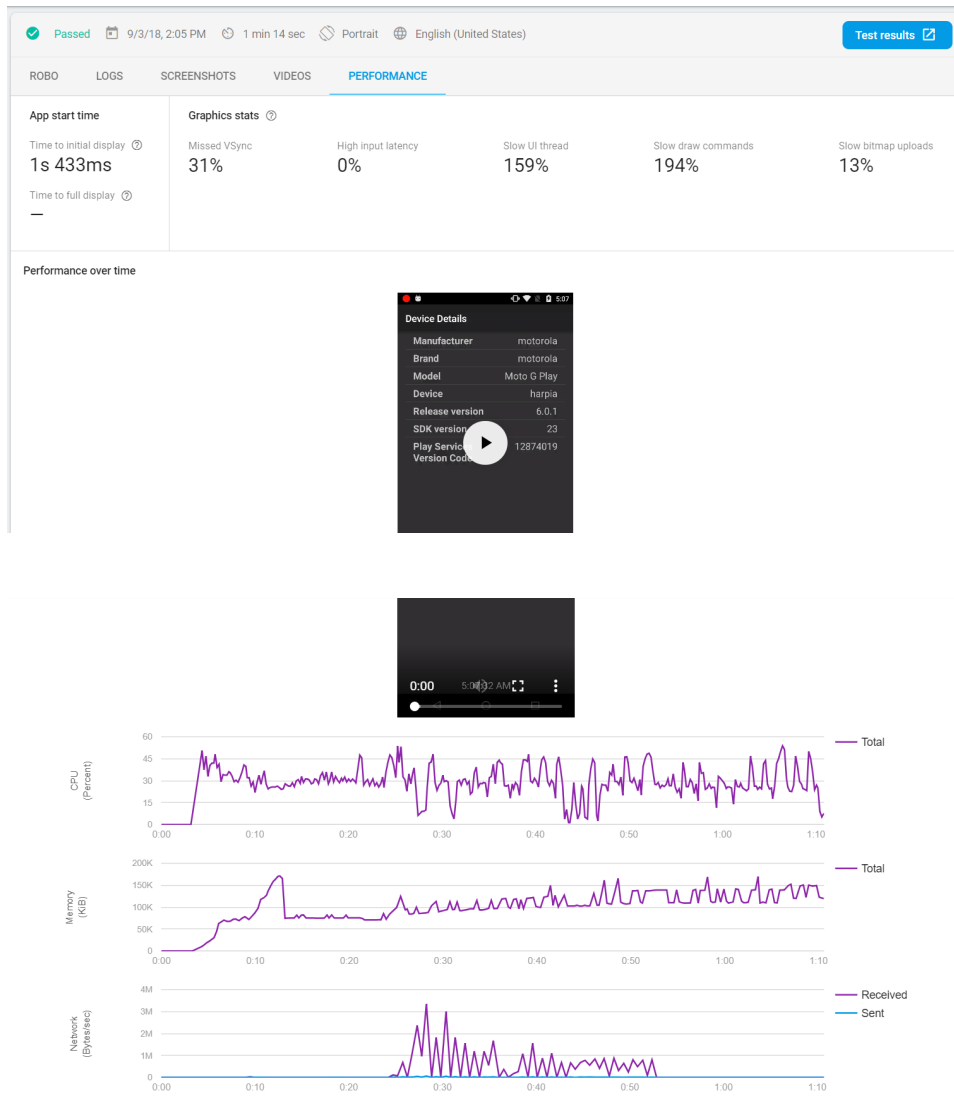


Figure 8: Robo Test reports

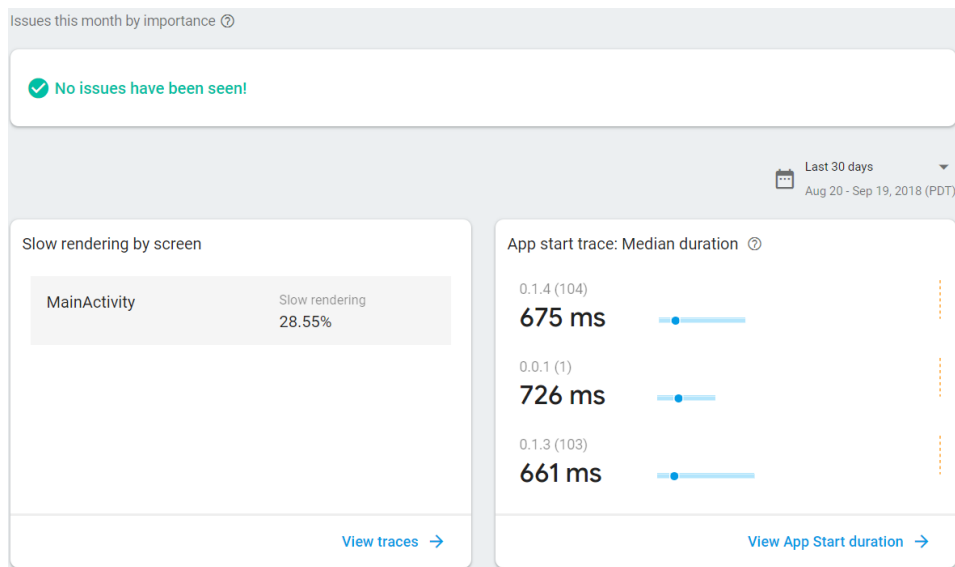


Figure 9: Firebase Performance Dashboard

### 5.2.2 Security

To be added.

### 5.2.3 User Interface

The following screenshot is a fragment of one of the Google forms which was used for the usability test.

Which part of the app did you find easiest to use?

- ☐ Image Prediction
- ☐ Contact Us Page
- ☐ Item Amount Calculator

Did the instructions on the Home page panels help you understand how the

- ☐ Yes
- ☐ No
- ☐ Not sure

Figure 10: Fragment of App feedback form

Feedback for the UI was also received through the Google app as well as through the application. The screenshots which follow are examples of the type of feedback received through these two channels.

Customer called Steven Lew has a query Inbox x



**softwaresharks@gmail.com**  
to me ▾

Query: Email unavailable

Feedback type: undefined

Change the predict button to analyse or recognise. It is not predicting, it ia telling me what it is.

Change the whole background, it does not look appealing. The prediction page looks like someone took a picture of their computer screen.

Figure 11: Example of email feedback



Charlie Claassen

★★★★★ 4 September 2018



I like what you were going for, but it still needs a lot of fine tuning. It is fine since it is still early in development. I just have some suggestions. First in the home tab the third image I can't read the text. White text on a bright background is bad design. The other 2 images are fine. Then with the actual image recognition. I tried it by taking a picture of a keyboard and a chair but it gave me a bunch of useless results. Not one of the options was the actual object. I do not know how your IA works or if it was a fault with my crappy camera, but still you need to look into that. Also the item amount calculator what is the point of it or how do I use it? F you could provide an explanation or even a simple example of how to use it to show users how to use it. These are just some suggestions from a fellow COS 301 Student. Other than that it looks good, much better than my project so far.

Figure 12: Example of Google Play Store review

#### 5.2.4 Image Prediction

A history of the image prediction tests and experiments can be found in the following spreadsheet:

[https://docs.google.com/spreadsheets/d/1pw4XgpSJdXgClEND7m05QIjq\\_caEajBUzQxC5zkqKlY/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1pw4XgpSJdXgClEND7m05QIjq_caEajBUzQxC5zkqKlY/edit?usp=sharing)

The graphs produced by the training process can be found in the GitHub repository here: <https://github.com/OrishaOrrie/SoftwareSharks/tree/master/Application%20Source/Backend/Keras%20Training%20Model/tests>

The following section is an in-depth retrospective of the different image prediction models used throughout the system's development.

**1. Initial Model** The Clarifai general model API was used to predict the classes of images initially, however this was far too unreliable and inaccurate for the project domain.

**2. Inception v3 Base model with Warehouse Photos dataset** The next classifier used in the project was built using TensorFlow and Keras as a framework and API respectively. These technologies allowed for the development of a whole new custom model, tuned to the needs of the domain. The model was built using Transfer Learning with Google's Inception v3 model as a base.

The model was trained on a dataset composed of real photos taken at a warehouse by the project members. The dataset comprised of 80 classes, with between 10-50 photos per class. Image Augmentation was used during the training process to increase the dataset to around 20000 images.

The performance of this model was better than the Clarifai general model, however still inaccurate for a majority of classes. Although training and validation accuracy were at an acceptable level, test accuracy involving images with white backgrounds, such as those found on the Internet, was extremely low, possibly due to the color backgrounds of the warehouse photos.

1. Test Image	Correct - 21.83%
2. Test Image - White Background	Incorrect - 0.0%
3. Validation Image	Correct - 79.88%
4. Internet Image	Correct - 43.40%
Training Accuracy	85.14%
Validation Accuracy	72.99%

**3. Inception v3 Base model with scraped dataset** The problem identified with the model in section 3 was that the warehouse photos dataset was far too small to be used with the Inception v3 architecture. This resulted in the model overfitting the small dataset and thus having poor generalization.

To solve this problem, a new, bigger dataset was required. Since capturing even more photos of the current classes was impractical and time-consuming, and there were no existing datasets on the Internet related to the application domain, the method of web image scraping was used.

An image scraper program was built using Node.js to automatically scrape the first 400 images from Google Images, for a specific search term. This program was run 57 times, once for each of the classes (only a subset of the warehouse photos classes were used, as many of the classes had no correlation when searched for on Google Images).

The resulting dataset comprised of 60 classes, with between 300-400 images per class, totalling to around 20000 images. This total was further multiplied during the testing process with images augmentation.

Several models were trained on this dataset with different hyperparameters, including different learning rates, different loss functions, and different optimizers. Several training processes showed signs of overfitting (increasing training accuracy but decreasing validation accuracy) after five epochs, which resulted in poor generalization. This can be seen below in Figure 11.

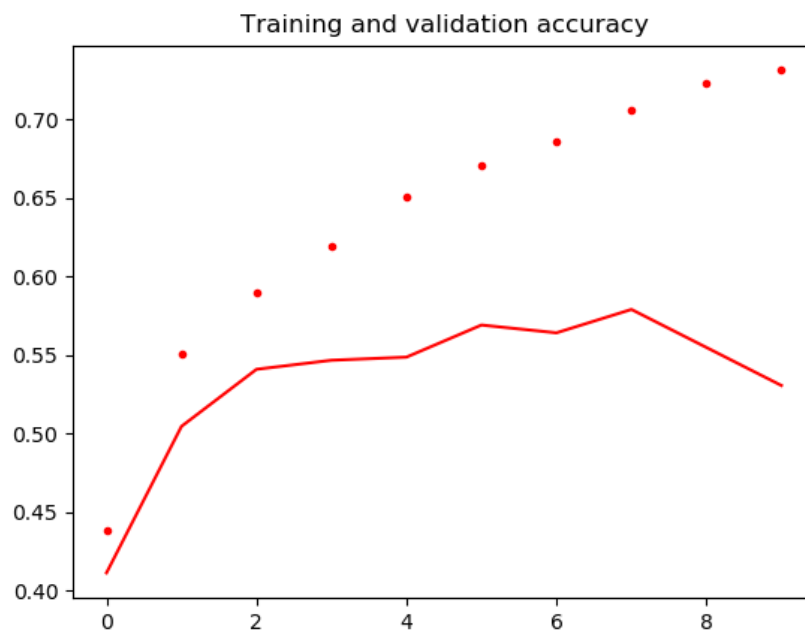


Figure 13: Setting 10 Training and Validation Accuracy

The model that was found to be the best was named "Setting 15", and had the following performance results for the training process.

Training Accuracy	95.61%
Validation Accuracy	62.74%

The following tests results were produced, with the test images made up of samples of the warehouse photos dataset. Top 1 accuracy means that the correct class was predicted, while top 8 accuracy means that the actual class fell within the top 8 predicted classes.

Top 1 Accuracy	95.61%
Top 4 Accuracy	63.33%
Top 8 Accuracy	78.33%



In the graph below (Figure 12), ranked predictions refer to the percentage of classes predicted between 2nd and 8th, while unranked predictions refer to the percentage of classes outside the top 8 predictions.

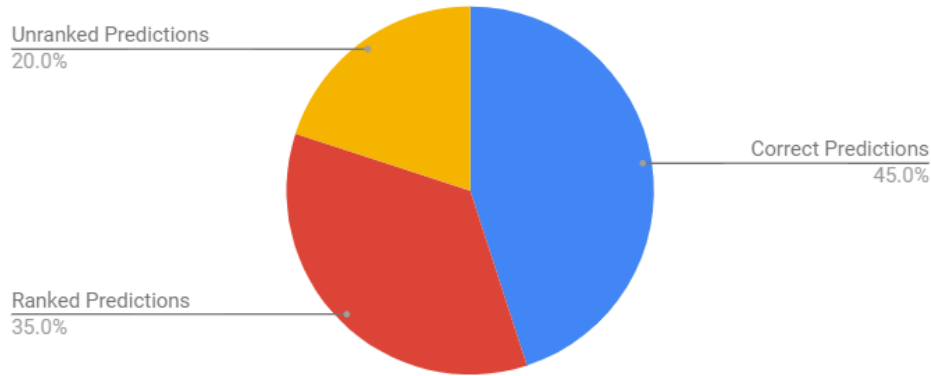


Figure 14: Setting 15 Test Accuracy

**4. MobileNet model with scraped dataset** Although the “Setting 15” model had a successful training period and an improved test accuracy, generalization was still not good enough to be used. Another problem was the large size of the model (166 MB), which made it difficult to deploy and slow when loading into memory.

A solution to this was to train a model using a different base architecture. The MobileNet architecture is different to the Inception v3 architecture in that it is designed to be small and less resource intensive. The Inception v3 network is designed for pure accuracy, and thus requires a large dataset, processing power, and training time, all of which was not available to the team.

Transfer learning was used once again to build a model called “Setting 16”, this time with the MobileNet base. The results for this model were very similar to “Setting 15” in terms of training results, as can be seen below.

Training Accuracy	93.89%
Validation Accuracy	63.59%

Test results dropped off a bit from “Setting 15”, although this tradeoff in favour of prediction speed and model availability was considered to be worthwhile.

Top 1 Accuracy	38.33%
Top 4 Accuracy	58.33%
Top 8 Accuracy	65.00%

**5. New scraped dataset** After deploying the "Setting 16" model on the app, feedback was received with regards to the prediction performance and accuracy. Although performance was generally considered to be good and usable, the accuracy for certain classes were still poor.

To solve this problem, each of the classes and their respective test accuracy were inspected. It was found that classes such as "safety gloves" and "cutting disc", which were distinct from any other classes, were consistently predicted accurately. Sets of classes that were very similar, such as "air quick coupler" and "female quick coupler", had really poor accuracy. It was found that these sets had many overlapping images, as was verified by Google Images' search results.

It became clear that a new dataset was required with a new list of classes. These classes had to be distinct enough so that there were few overlapping images, but still specific enough for example to distinguish between a countersunk screw, a self-tapping screw, and a cheesehead screw. The classes were also specifically hand-picked from Bramhope's (the client's) online catalogue, so that there was a direct correlation to the application domain.

A new model was trained with the MobileNet base, called "Setting 17". The model had improved training results over "Setting 16", as can be seen below.

Training Accuracy	92.63%
Validation Accuracy	74.02%

Testing and generalization is to be conducted once a set of test images are collected. Initial testing for existing test images indicates outstanding results.