Software Testing Plan
Version 1.0
April 3, 2020
EnginAir

Sponsor:
Harlan Mitchell, Honeywell
(harlan.mitchell@honeywell.com)

Mentor:
Scooter Nowak
(gn229@nau.edu)

Team Members:
Chloe Bates (ccb323@nau.edu),
Megan Mikami (mmm924@nau.edu),
Gennaro Napolitano (gennaro@nau.edu),
Ian Otto (dank@nau.edu),
Dylan Schreiner (djs554@nau.edu)

Contents

Chloe Bates ccb323@nau.edu
Megan Mikami mmm924@nau.edu
Gennaro Napolitano gennaro@nau.edu
Ian Otto dank@nau.edu
Dylan Schreiner djs554@nau.edu

1. Introduction

In 1903, the Wright Brothers made history by building and flying the first successful powered airplane. Since then, airplanes have become the most used mode of long-distance transportation, most of which rely on multiple engines, specifically gas turbines, for primary propulsion. These engines require periodic maintenance which are reported to mechanics and engineers via a diagnostic report. Proper upkeep and diagnostic testing are crucial in preventing maintenance delays and keeping the engine functioning properly to ensure a safe flight.

Honeywell, our client, manufactures gas turbine engines, specifically turboshafts, turbofans, and turboprop engines. These engines are typically found in military aviation, helicopters, and business jets. These engines contain an Engine Control Unit (ECU), which saves trending and maintenance data that is condensed into a hardware diagnostic report. Honeywell engine operators are required by contract to download and send this diagnostic report once a month. This process requires:

- a manual port connection using a USB device and a cable,
- transferring the file to a USB drive,
- and sending the file via email.

This process is tedious and collects a small data set containing basic maintenance information. Because this process occurs once a month, it can result in infrequent data collections and missed maintenance opportunities.

To better this process and collect flight data more frequently, Honeywell is currently developing a connected engine product called the Connected Engine Data Access System (CEDAS). CEDAS is hosted on an embedded computer located in the aircraft and allows engines to autonomously upload its engine data wirelessly via WiFi to a cloud. If the WiFi connection is spotty or nonexistent at a certain location, the diagnostics may not be sent. When this happens and the data upload is not on schedule, it is difficult to determine the status of the aircraft; whether it is grounded, inflight, or if there is a potential problem with the engine.

To solve this problem, our team has come up with the application called the Connected Engine Upload Status System (CEUSS). This system is a sophisticated GUI that presents information ingested from three data sources via a server backend and processed via a correlator. The GUI can simulate potential upload locations and indicate the possibility of an upload success. It is also capable of determining when and where all uploads occurred as well as running a diagnostic report to determine the cause of a failed upload.

Like any software system, CUESS will undergo software testing. Software testing is an essential part of software implementation and is used to discover bugs and flaws within the system to correct before deployment. Software testing is also beneficial in determining if a system is behaving as expected and producing the correct results under specific conditions. It can also establish a software's robustness in its ability to handle unexpected or erroneous inputs. This document outlines the testing procedures for unit, integration, and usability testing for CUESS.

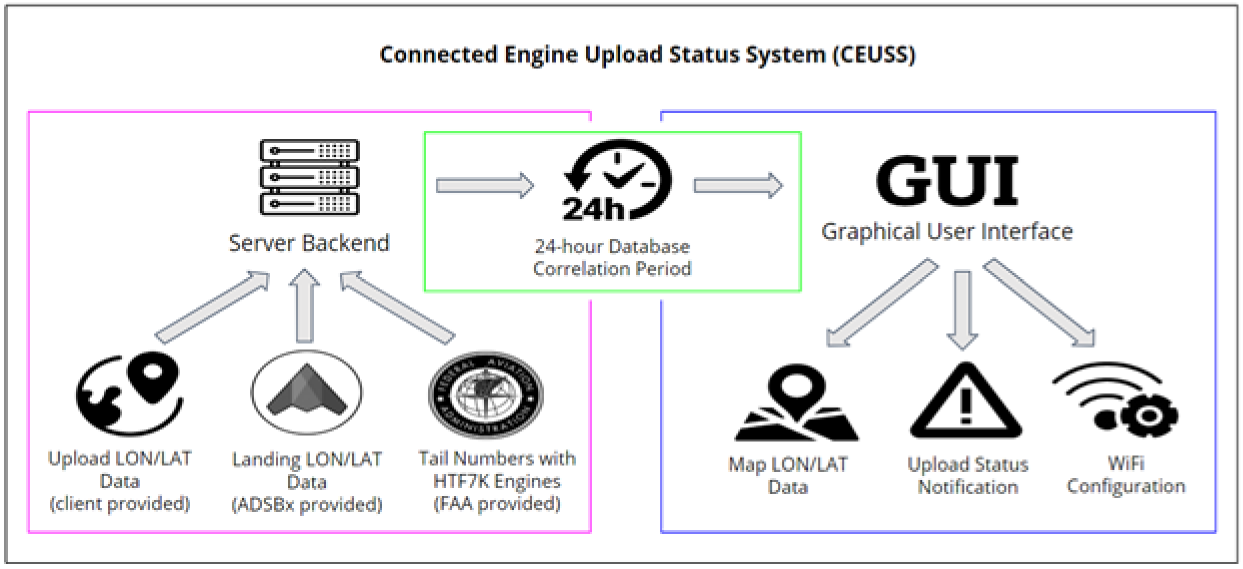

Connected Engine Upload Status System (CEUSS)
Server Backend
24h
24-hour Database Correlation Period
GUI
Graphical User Interface
Upload LON/LAT Data (client provided)
Landing LON/LAT Data (ADSBx provided)
Tail Numbers with HTF7K Engines (FAA provided)
Map LON/LAT Data
Upload Status Notification
WiFi Configuration
Figure 1: Overall Architecture of the Connected Engine Upload Status System

Figure 1 above illustrates the high-level architectural components of CUESS. The pink line outlines the Data Import component, the green outlines the Correlation component, and the blue outlines the GUI Rendering component. The unit tests using JUnit will examine each method in each specific component separately to determine its effectiveness in contributing to correct results and data. The integration testing using Postman will analyze the data flow between the GUI and the database via web API calls and usability testing, automated using the Google Lighthouse Auditing Tool, will run a wide range of tests on the GUI web pages and report each page's success and shortcomings.

Since our end-product is a GUI, usability testing is extremely important since end users will typically only interact with this part. The other crucial testing component concerns the integration of the data import and correlation such that the accuracy of data transfer is maintained and passed to correlation to provide correct results. In order to ensure this, unit tests are produced and executed to identify if helper functions may cause data corruption or introduce other bugs/issues. It is essential that we identify bugs and other faults in our software before handing the product over to the client. If left unresolved, these bugs can go undetected and interfere with the functionality and usability of the program.

2. Unit Testing

In this section, we will outline our plans for Unit Testing. Unit testing is a level of software testing which isolates and tests each individual component of a program, called "units", and verifies that each, functions as intended. Knowing that each individual piece of the source code is valid, a programmer can feel more confident in the components of their software and continue onto integration testing. Unit testing uses a strategy called a "bottom up testing approach" which can make the overall testing process more advantageous because it can catch major flaws at lower level modules early on.

To assist in our execution of unit testing, we will be utilizing a popular testing framework JUnit. JUnit is specifically written for Java programming projects and due to its simplicity and speed, it is widely adopted by and supported by most IDEs. Because the server-side-app/ is written in Java, JUnit makes unit testing this module simple. The server-side-app/ supports two of the main components of CEUSS, the data import and correlation components.

Our unit testing will measure test coverage using CodeCov and JUnit. CodeCov is an open source automatic coverage report generator that supports Java. JUnit includes a built-in code coverage functionality which will help us reach 100 percent code coverage in our tests.

Because we utilized object-oriented programming techniques, methods are the smallest units we will test. Table 1 lists out all the classes whose methods will have a corresponding unit test. In the subsequent sections, we will detail a test plan for each class's methods. ADSBImporter's nested class, ADSBDownloader, will be omitted from unit testing due to its highly coupled structure within the data import component. Because it is highly coupled, however, it is reasonable to say that it will be loosely tested via the other methods being tested.

Connected Engine Upload Status System (CUESS)	
server-side-app/	
Data Import	Correlation
AppEngine ImportExecutor Importer ADSBImporter CEDASImporter TailImporter	Correlator

Table 1: Data Import and Correlation Classes Subject to Unit Testing

Chloe Bates ccb323@nau.edu
Megan Mikami mmm924@nau.edu
Gennaro Napolitano gennaro@nau.edu
Ian Otto dank@nau.edu
Dylan Schreiner djs554@nau.edu

2.1. Data Import

The following sections describe the unit tests conducted for the server-side-app/ data import component. This is responsible for extracting data from three sources and importing it into the database for correlation.

server-side-app/AppEngine:
Table 2 describes the methods being tested in the AppEngine Class which is responsible for creating options for the command line input and determining how to proceed once the input is passed in. It includes a description, an expected result, and a test plan.

server-side-app/AppEngine	
parseArgs()	**Description**: Tests that the command line only accepts the created Options objects ("importTails", "importCEDAS", importADSB", "databaseName", "correlator"). **Test Plan**: Handle the exception (class Options throws an IllegalArgumentException) in the test so that it will not fail: @Test(exception - IllegalArgumentException Class) **Expected Result**: If illegal arguments are passed in, they will be handled
execute()	**Description**: Calls ImportExecutor's execute() with the command line arguments passed in **Test Plan**: Test whether command line arguments were correctly passed in by testing the number of command line arguments. Both, correct and incorrect, number of arguments are being tested. **Expected Result**: If the expected amount of arguments is being passed in then it should pass and vice versa

Table 2: server-side-app/AppEngine Methods and Descriptions

server-side-app/executors/ImportExecutor:
Table 3 outlines all methods in the ImportExecutor class used to create a template for each type of data being imported (ADSB, CEDAS, and Tails). Each method description includes an expected result and a test plan.

server-side-app/executor/ImportExecutor	
execute()	**Description**: Calls getImporter(). This method takes in an input and gives getImporter() the correct information. **Test Plan**: Give multiple bad inputs to make sure that the method can handle them. **Expected Result**: Will fail if any invalid values are given.

getImporter()	**Description**: Selects and runs the correct importer, either ADSB, CEDAS, or Tail. **Test Plan**: Create a test that has typical edge values as parameters, such as file names with lowercase letters or correct file names with no data. **Expected Result**: This test will fail if an incorrect file name is passed in.
makeConnection()	**Description**: Creates and returns a connection to MongoDB. **Test Plan**: Tests the connection made for the passed in database name (i.e. "databaseTest"). **Expected Result**: This test should pass since this function does not rely on a specific database name.

Table 3: server-side-app/executor/ImportExecutor Methods and Descriptions

server-side-app/importers/Importer:

Table 4 outlines all methods in the Importer class used as an abstract class for all other importer classes. Each method description includes an expected result and test plan.

server-side-app/importers/Importer	
execute()	**Description**: This is the abstract class for all importers. It is the layout for the three importer classes that our project uses. **Test Plan**: Make an object that the importer can handle and make sure it is used correctly. **Expected Result**: An exception should be thrown.

Table 4: server-side-app/importers/Importer Methods and Descriptions

server-side-app/importers/ADSBImporter:

Table 5 outlines all methods in the ADSBImporter class used to read in the ADSBx JSON data, filters it, and adds the data to the database. ADSBImporter calls a nested function ADSBDownloader to help with the execution of these tasks. Each method description includes an expected result, exceptions, and test plan.

server-side-app/importers/ADSBImporter	
execute()	**Description**: Will take an ADSBDownloader object and save its contents to the database. **Test Plan**: Create a simple ADSBDownloader object and use it in the method. Check to see if the correct data is extracted. **Expected Result**: A print statement with the correct tail number as this is already being printed by the code.

Table 5: server-side-app/importers/ADSBImporter Methods and Descriptions

server-side-app/importers/CEDASImporter:

Table 6 outlines all methods in the CEDASImporter class used to gather CEDAS uploads from Excel or JSON files and correctly imports the data. Each method description includes an expected result and test plan.

server-side-app/importers/CEDASImporter	
execute	Description: Checks which file type is provided and runs the correct method for it. Test Plan: Write three tests that will check both Excel and JSON formats as well as an invalid format. Expected Result: The invalid file should fail. The other two formats should populate the database with the correct information.

Table 6: server-side-app/importers/CEDASImporter Methods and Descriptions

2.2. Correlation

The following section describes the unit tests conducted for the server-side-app/ Correlation component. This is responsible for making sure that data that is given to the correlator can be handled. It will also ensure that the class produces the correct output.

server-side-app/Correlator:

Table 7 outlines all methods in the Correlator class used to correlate CEDAS and ADSB data to produce a CorrelatedFlight object. Each method description includes an expected result and test plan.

server-side-app/Correlator	
correlate()	Description: Creates the flight outcome field. Test Plan: A hashMap of 2 or 3 flights will be given to the method. We will test if the correct output is given with these flights. Expected Result: Data being correctly created to add to the database.
execute()	Description: Uses MongoDB connection previously created in makeConnection(). It correlates flights with tail numbers. Test Plan: Make sure that every tail number that is provided is correlated. Expected Result: Should have no errors from this method. If any presents, this test fails.
isLandingSituation()	Description:Cchecks if altitude is at a point of descent with the line: `return data.altitude < 6000 && data.speed < 300;`

	Test Plan: Test ADSB input data values with boundary values: 6000 and 300. Expected Result: ADSB values 6000 and 300 will return false, as expected.
processTail()	Description: Creates each individual tail in the flight. Called by execute(). Test Plan: Use a negative number as a passed in process tail number. Expected Result: This should still pass the test and a tail should still be processed.
shouldBeNewFlight()	Description: Checks if lost tracking for over 1 hours (this tells us the plane most likely landed). Called in processTail() Test Plan: Test ADSB input data values with positive and negative time values. Expected Result: ADSB values that have times with negative values will return an error.

Table 7: server-side-app/Correlator Methods and Descriptions

After Unit Testing is complete and each unit is valid and able to complete its purpose, our next set of tests will occur.

Chloe Bates ccb323@nau.edu
Megan Mikami mmm924@nau.edu
Gennaro Napolitano gennaro@nau.edu
Ian Otto dank@nau.edu
Dylan Schreiner djs554@nau.edu

3. Integration Testing

Integration Testing is the next level in software testing where individual units are combined and tested as a whole component. Integration testing is purposed to check the accuracy and precision of interactions and data flow between these individual units. These tests focus on main components/modules of a system and check whether or not the exchange of parameters and return values are occurring properly between each module.

CEUSS is composed of three major components within two main modules. The server-side-app/ module contains the Data Import and Correlation components and the web-app/ module contains the GUI Rendering component. Integration testing between the server-side-app/ and the web-app/ is focused on integrating the correlation MongoDB and the GUI's API such that API calls access and return raw data from the database correctly. Figure 2 below shows this module integration.

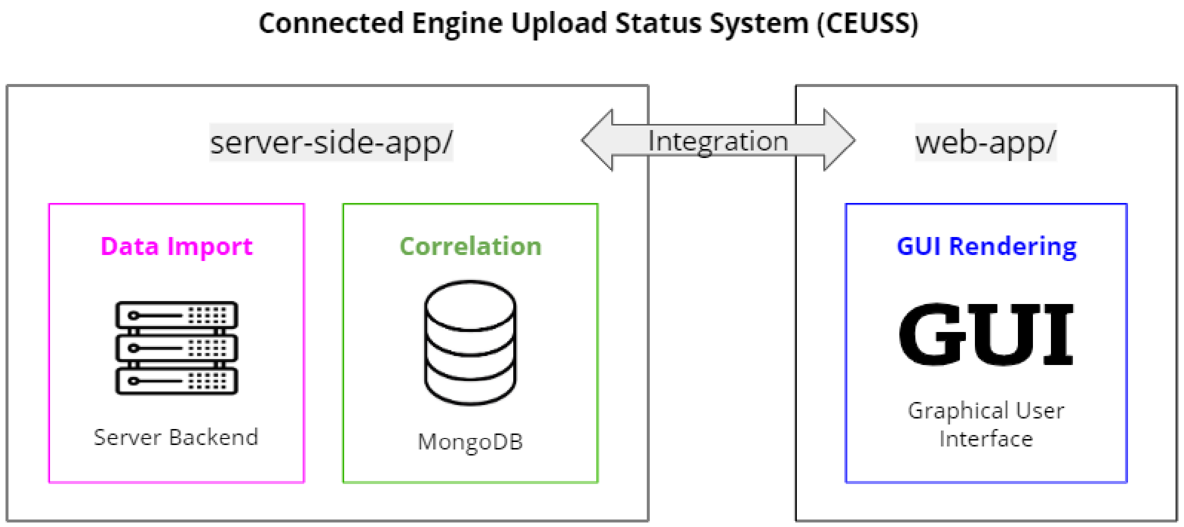


Figure 2: CEUSS Module Integration

Database objects that the API will be accessing and rendering information come from the CorrelatedFlight JSON objects created by the correlator. Because the Data Import and Correlation components are written and executed using external libraries and platforms, we will refrain from testing these ourselves. The Mongoose JSON library used for the importation of data from the database to the front end has extensive unit tests for each of its functions publicly listed on their GitHub repository. MongoDB used for the correlation component has similarly been tested by its core developers to ensure data integrity and accuracy.

To automate this process, we will be using the Postman testing utility, a collaboration platform for API development, used to simulate API requests to debug and inspect the responses. These API requests are in the form of database search queries. The API web client will help to verify and validate that the API returned responses are what is expected. The following circumstances describe the expected API response:

1. If an API call returns an empty response, the result should be an empty JSON array.
2. If an API call returns a result:
   - The result must return, at most, the number of results specified in the search query by the limit parameter and
   - The result must return a result matching a predefined JSON format as specified in the server-side-app/.

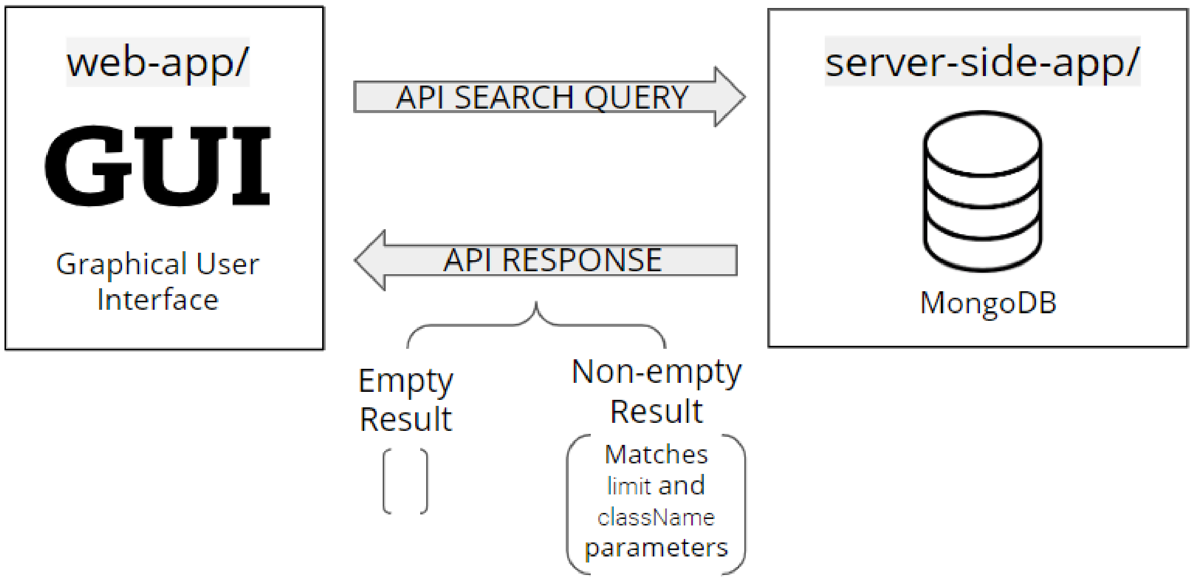


Figure 3: Detailed Integration via API Calls

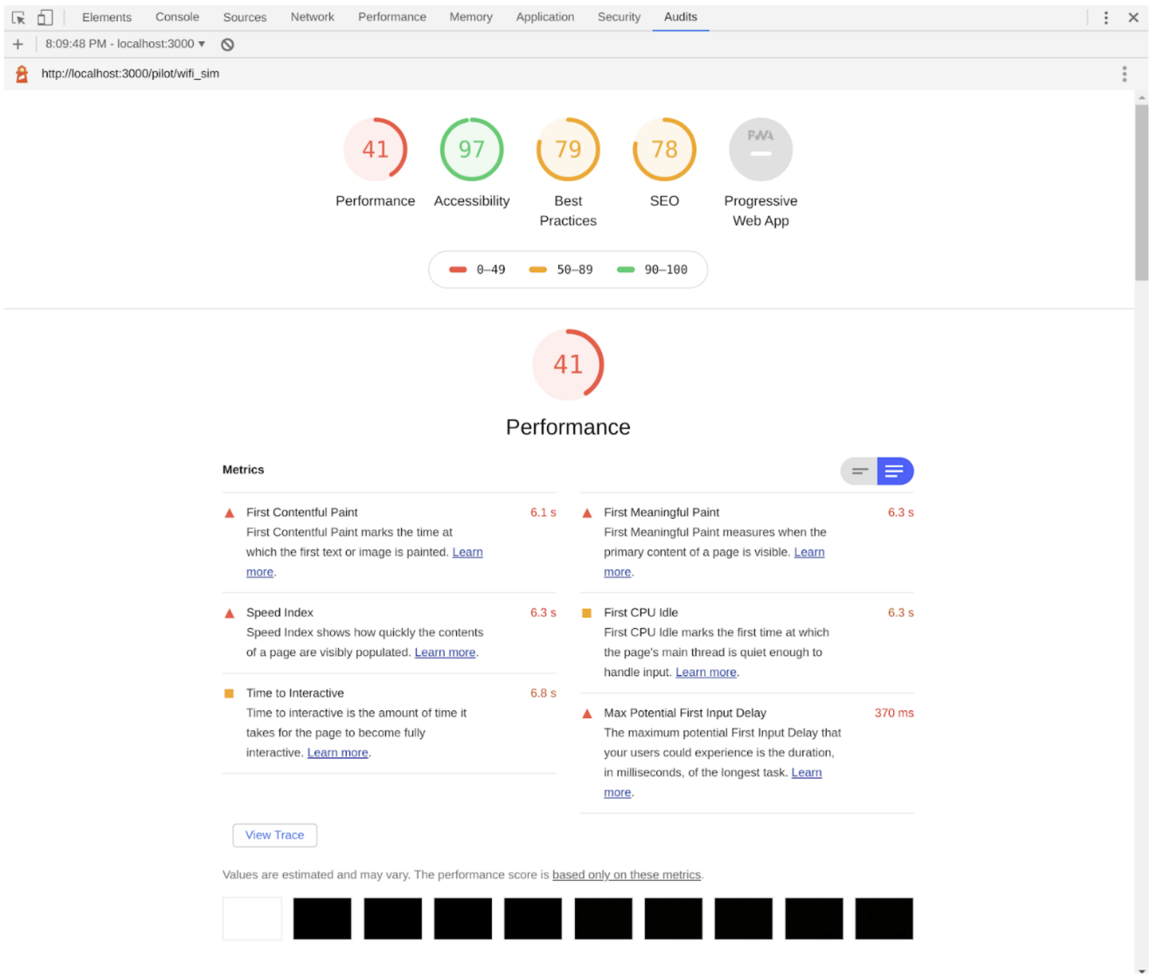
Provided the above contracts are followed in the test, we can be confident that the web-app/ module will be able to render the API result appropriately. Integration between modules has been verified and usability testing can occur.

4. Usability Testing

Usability testing is the last set of tests to be completed in the software testing process. These tests evaluate the friendliness and functionality of the end-product by having real users, preferably the targeted end users, attempt to complete tasks. These tests also evaluate the product's accessibility, desirability, usefulness, and compliance to user's needs. Usability tests are targeted to uncover confusing or unnecessary aspects of the product to improve overall user experience.

Our end-product is a GUI tailored to help engine technicians and aircraft pilots. Each end-user will have a toggle key to access their selected web pages created to help each party complete tasks. Engine technicians will use the GUI to understand more technical aspects of an aircraft upload process (i.e., the reason for upload failure, landing locations, status of upload entry, etc.) while aircraft operators (i.e. pilots) are more concerned with the big picture like where to park to ensure a successful upload. Due to each users' end goal, determines the nature and logistics of the web pages. For example, technician pages would include upload statuses and explanations for upload failures which would be irrelevant and possibly confusing for pilots. Technician pages can include more technical terms and common abbreviations associated with an engine upload whereas pilots would probably need their pages to explain this lingo in layman terms.

During our alpha prototype demo, our GUI underwent a preliminary usability test with our client pertaining to visual appeal and product functionality. Because our product was essentially complete (other than a few bugs not essential to product functionality) and our client did not have any concerns or suggestions to usability modifications, we decided to automate future usability tests. We will be using Google's LightHouse which is an open-source auditing tool used to evaluate a webpage's performance, accessibility and other usability test-like attributes. Lighthouse is executed on a single URL which will generate individual reports. These reports contain information on the website's failing audits, each of which have a reference document explaining why the audit is important and how it can be fixed. After these fixes are made, the audit will be executed again and repeated until all audits are passing. Lighthouse has its own CI to help with regression testing. Figure 4 shows an example of a Lighthouse report executed on a CUESS webpage.



41 Performance
97 Accessibility
79 Best Practices
78 SEO
– Progressive Web App

🟥 0–49 🟧 50–89 🟩 90–100

41

Performance

Metrics

▲	First Contentful Paint First Contentful Paint marks the time at which the first text or image is painted. Learn more.	6.1 s	▲ First Meaningful Paint First Meaningful Paint measures when the primary content of a page is visible. Learn more.	6.3 s
▲	Speed Index Speed Index shows how quickly the contents of a page are visibly populated. Learn more.	6.3 s	■ First CPU Idle First CPU Idle marks the first time at which the page's main thread is quiet enough to handle input. Learn more.	6.3 s
■	Time to Interactive Time to interactive is the amount of time it takes for the page to become fully interactive. Learn more.	6.8 s	▲ Max Potential First Input Delay The maximum potential First Input Delay that your users could experience is the duration, in milliseconds, of the longest task. Learn more.	370 ms

Values are estimated and may vary. The performance score is based only on these metrics.

Figure 4: Screenshot of Lighthouse Audit Report on Pilot Simulation Page

Another important aspect of usability testing is user acceptance testing which determines if all requirements of the system have been met. This affects the user's ability to use the product to satisfy their needs. Focusing on performance requirements specifically, our requirements document indicates that CUESS must satisfy three performance requirements:

- The backend web application shall have an average TTFB of 200ms/request.
- The backend web application shall execute 100 requests/min.
- The graphical interface shall render webpage within 10 seconds.

Lighthouse audit reports contain information on TTFB, requests/min, and rendering speed. These requirements should be easy to verify using this automated usability testing tool.

5. Conclusion

In conclusion, diagnostic reports and periodic maintenance of aircraft engines are essential to establish proper functionality and maintain safe flights. Honeywell's current engine data download process is tedious and results in a small data set and although Honeywell has upgraded to wireless uploading via their CEDAS system, it is limited by the adequacy of WiFi connection. Our solution, CUESS, helps predict when and where an upload should occur and helps predict why an upload failed. This document outlines the software testing plan implemented for this project. CUESS has endured extensive unit testing using JUnit, integration testing using Postman, and usability testing using Google's Lighthouse auditing tool. Due to this software testing plan, we, team EnginAir, are confident that we will be delivering our client a bug free, seamlessly integrated, and user-friendly product that is capable of bettering Honeywell's maintenance process to keep engines working properly.