



As-Built Report

Version 1.0

April 28, 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

1. Introduction	3
2. Process Overview.....	4
3. Requirements	5
4. Implementation Overview	6
5. Architecture Overview	7
6. Module and Interface Description	9
6.1. Backend Module.....	9
6.2. Front end Module	17
6.2.1. Technician	18
6.2.2 Aircraft Pilot	25
7. As-Built vs As-Planned	28
8. Testing	29
8.1. Unit Testing	29
8.2. Integration Testing.....	30
8.3. Usability Testing.....	31
9. Project Timeline	32
10. Future Work	33
11. Conclusion	34
Appendix A: Development Environment and Toolchain	35
Hardware	35
Toolchain	35
Setup	36
Prerequisites.....	36
Initializing the Environment.....	37
Production Cycle.....	38

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 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 turboprops. 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:

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

This process is tedious and collects a small data set containing basic maintenance information. Because this only 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 has developed a connected engine product called the Connected Engine Data Access System (CEDAS). CEDAS is hosted on an embedded computer located in the aircraft which 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, in flight, 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). CEUSS is composed of a server backend and a GUI front end engineered to help engine technicians understand where and why potential upload failures occur. CEUSS can also help pilots visualize where to park their aircraft for the highest chance of upload success. CEUSS accomplishes this by relying on a server backend correlator which produces a file containing the upload status and a diagnostic test results that the GUI renders to the user.

2. Process Overview

For our capstone project to be successful, we had to develop team roles and procedures. This information is outlined in our Team Standards Document completed at the beginning of the Fall 2019 semester.

Our team roles are listed below:

- Megan assumed the role as team lead and was responsible for leading team meetings, working on all documentation and deliverables, and communicating with the client.
- Chloe was responsible for creating and maintaining the weekly task reports as well as contributing as a “code monkey”.
- Ian was coined scrum master and was responsible for managing all code development and deployment.
- Gennaro and Dylan assisted as “code monkeys” and helped with documentation when needed.

Team procedures included scheduling weekly meetings and outlining communication methods, development tools, and documentation standards. Our weekly team meetings were held on Sundays over Discord’s voice chat feature and our primary communication between team members were conducted via a text message group chat, for convenience. Our team decided to utilize the Git version control and issue tracking system on GitHub.com under our project repository, <https://github.com/EnginAir>. All deliverables and documentation were located on a shared Google Drive and were edited using Google docs and Google slides which were exported to PDF versions and archived on our website.

3. Requirements

After outlining our team standards, our team focused on requirements acquisition through weekly conference calls and continuous email threads with our client. We decided to describe our requirements in the form of user stories to more easily explain the functionality for the different types of end-users, engine technicians and airplane operators. Engine technicians are more interested in knowing the technical aspects of an aircraft upload process (i.e., upload failure explanations, landing locations, status of upload entry, etc.) while the aircraft operators (i.e. pilots) are more concerned with the big picture like where to park the aircraft to ensure a successful upload.

The eight high-level user stories are listed below:

As an engine technician, I want to be able to:

- view all aircraft landing locations, every 24 hours.
- visualize all flights that are currently in progress.
- simulate various locations and their corresponding WiFi configuration.
- know the status of each landing/upload entry.
- visualize the status of each upload entry
- run a report to determine the cause of a failed upload.

As an aircraft pilot/operator, I want to be able to:

- visualize locations on where to park the aircraft for an upload success.
- simulate locations and their WiFi strength.

In breaking down the requirements to fulfill each user story, we were able to extract a total of 30 functional and 6 performance requirements as described in the Requirements Document version 2.0. Table 1 describes the system and user requirements and lists out the requirements types. The system requirements were broken down into 3 main modules, the Database, Backend Server, and GUI.

Requirements			
System			User
Database	Backend Server	GUI	
Describes the data type and format contained in the database component	Describes the correlation component expected results	Describes logistics for GUI functionality	Describes GUI functionality from the user's perspective
9 Functional 1 Performance	7 Functional 2 Performance	3 Functional 3 Performance	11 Functional

Table 1: Outline of System and User Requirements

4. Implementation Overview

To complete the requirements described above, our system requires five main technical components including a Flight Database, Command Line Application, Database Management, Web Application (Front/Back End), and Graphical Illustration. Based on a technology feasibility conducted in the Fall 2019 semester, the chosen technologies for this project are listed below:

Public Flight Database: ADSB Exchange - ADSB Exchange is the world's largest source of unfiltered flight data. Unfiltered data includes military and private aircrafts which allows for a better overview of all Honeywell engines. ADSBx data reports contain flight information such as aircraft tail numbers and longitude and latitude coordinates which are downloadable as JSON files. ADSBx records data on flights every minute resulting in a huge dataset.

Command Line: Java - Java is a general-purpose object-oriented programming language and was chosen to be the command line application language primarily due to our team's experience with the language. We plan on using the Apache POI, which is an open source library for handling excel formatted documents and can help easily integrate some of the core functionality of the backend client.

Database: MongoDB - MongoDB is a document database which stores data in a JSON format. This makes importing ADSBx data extremely convenient and is easy to use. MongoDB also provides "drivers" which allows programs to interact with the database. Since MongoDB has Java and Node.js drivers, integration with the command line and web app back end should be simple.

Web App Front End: NGINX - NGINX is a popular caching and general-purpose web server. Because it is a caching server, it supports proxying to backend services which allows for load balancing. These are important to ensure stability during server updates and to handle lots of traffic by API calls and large quantities of return data.

Web App Backend: Node.js - Node.js is a commonly used variant of JavaScript designed to run in a server environment and has a standard server framework called Express which allows for asynchronous caching of HTTP requests. Node.js will interact easily with MongoDB given that it natively supports JSON as its object notation.

Graphical Illustration: OpenStreetMap - OpenStreetMap is a mapping API like the Google Maps API. OpenStreetMap provides a free editable map of the world and when used together with JavaScript libraries like Leaflet or OpenLayers, creating an interactive map with LON/LAT points is simple.

5. Architecture Overview

Our system, Connected Engine Upload Status System (CEUSS), consists of 3 components: the server backend, correlation period, and the front-end GUI web page. The correlation period is normally considered part of the server backend since it is associated with data import and data analysis but for purposes of this section, the data import and correlation execution will be considered separate. This component breakdown is outlined in Table 2 shown below and maps the technology chosen above.







server-side-app/			web-app/		
Data Import		Correlation	GUI Rendering		
Public Flight Database	Command Line	Database	Web App Back End	Web App Front End	Graphical Illustration
					
ADSB Exchange	Java	MongoDB	NGINX	Node.js	OpenStreetMap

Table 2: Technologies Used in CEUSS

Figure 1 illustrates the 3 components of the overall system, CEUSS, and highlights the sections which utilize the chosen technologies. The pink outline focuses on the data importation from three sources, the green outline indicates the correlation step, and the blue outline concentrates on the GUI web page rendering. The data importation, correlation, and web page rendering data flows are explained in the subsequent sections.

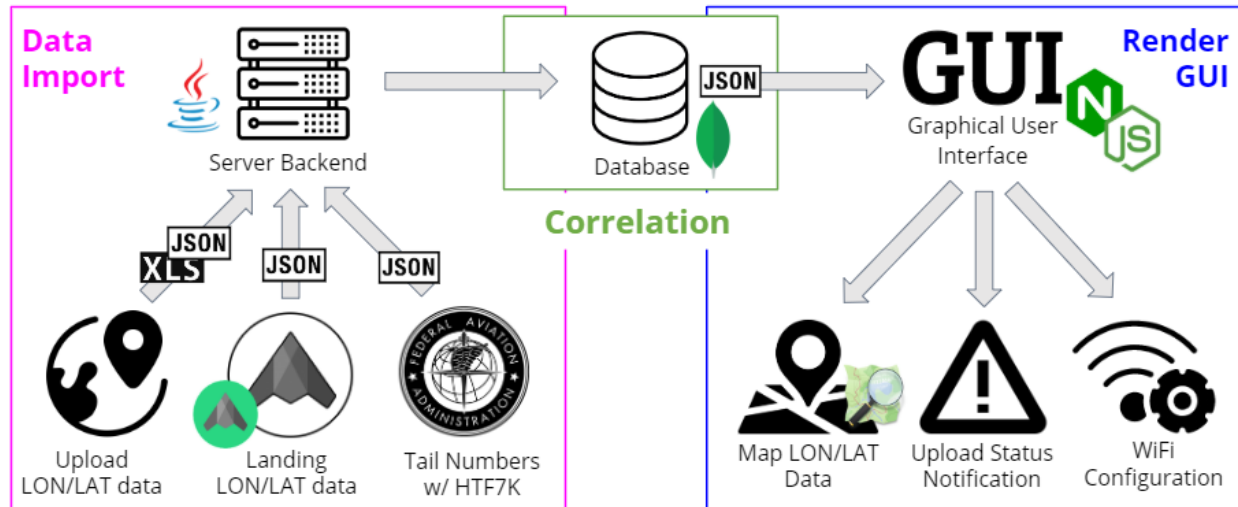


Figure 1: Overall Architecture of the Connected Engine Upload Status System

Data Import

CEUSS processes data from three sources to be used in the correlation step. Upload LON/LAT data is provided by the client as a Excel file, the Landing LON/LAT data is downloaded from ADSBx as a JSON file, and the tail numbers of aircrafts that contain an HTF7K engine are downloaded from the Federal Aviation Administration (FAA) as a JSON file. Due to security reasons, our client was not able to provide us actual CEDAS upload data containing tail numbers of the aircrafts, but he did provide the data fields which we were able to create a mock file that imitated this type of data. Because we could not access all the real-time ADSBx data, we were only able to extract data from the first day of every month pulled from their free archived database.

Correlation

The correlation step is the core of our system's functionality. This step correlates the ADSBx and CEDAS upload data located in the MongoDB database to determine flight upload statistics like identifying when and where missing uploads occur and categorizing the status of each upload. This correlation results in a JSON file containing all the correlation results and statistics that will be rendered to the user by the GUI.

Render GUI

The webpage is responsible for rendering the correlation statistics visually for the user to understand. The JSON file produced by the correlation step contains LON/LAT coordinates of all correlated flights and its corresponding WiFi configuration. This will be plotted onto a map using OpenStreetMap. The JSON file also contains a reference to missing or mismatched flight entries in the database which can be illustrated on the web page as a notification to the user. The webpage also renders the database in an easy searchable way.

6. Module and Interface Description

This section outlines the details of the two modules, the backend server module and the front-end GUI module. The backend module is composed of the data import and correlation components which are expressed using a UML diagram since this module architecture is influenced by object-oriented programming (OOP) principles such as classes and objects. The front-end GUI is illustrated using a page type breakdown diagram that includes subpages and page options.

6.1. Backend Module

This section describes the backend server module and explains each component in depth specifying each method and field attribute. Figure 2, shown on the next page, illustrates the UML structure of the backend and is followed by a detailed explanation of each UML component and its role in the backend execution flow.

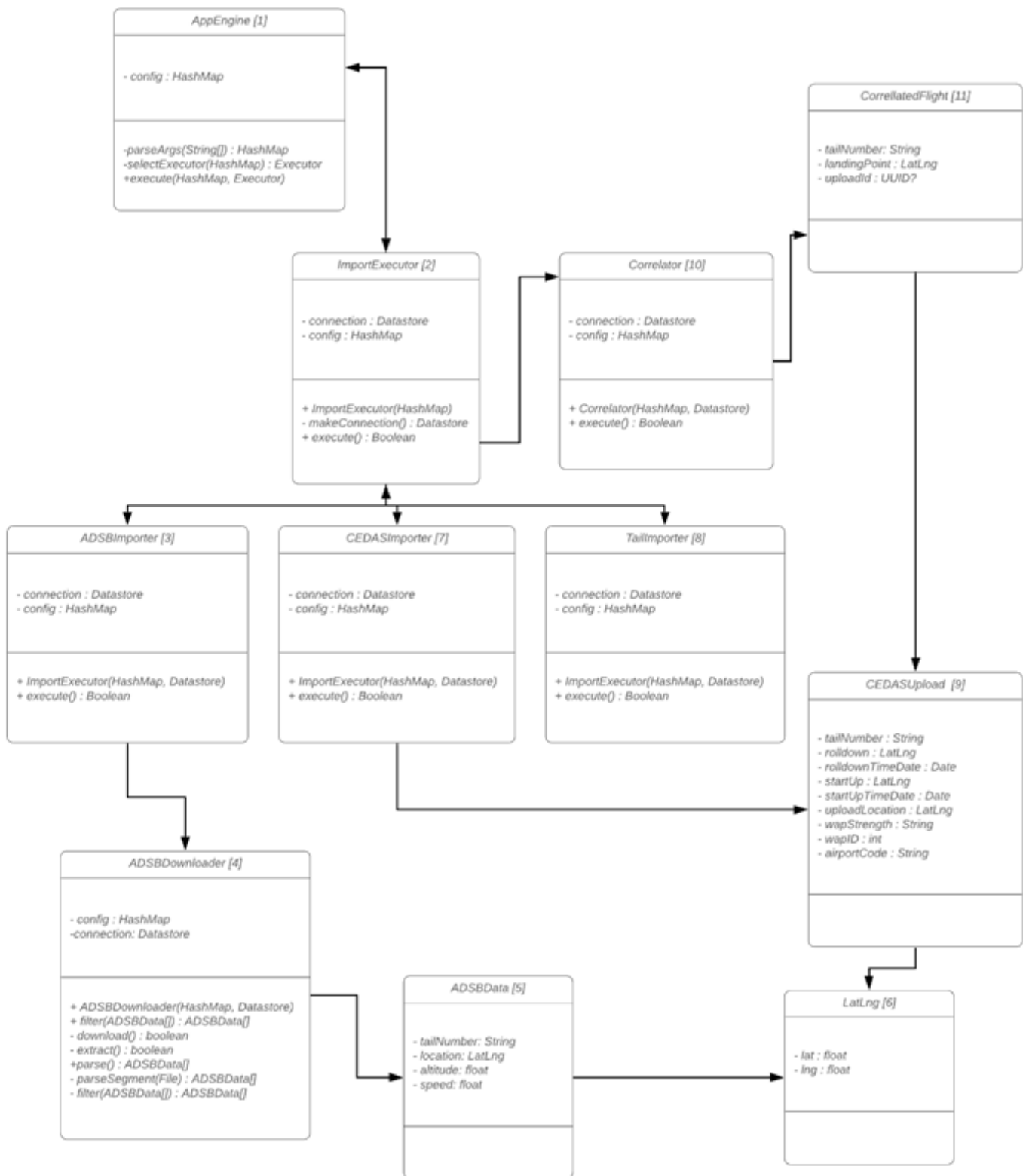


Figure 2: UML Diagram of Backend Architecture

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

Our system backend architecture begins at AppEngine [1], which determines which command-line options were passed into the program. AppEngine then creates a new ImportExecutor [2] object, which is responsible for connecting with MongoDB to allow data correlation by the Correlator [10]. The Correlator is responsible for most of the system's computation and is used to pre-compute data that the front end will later use. The Correlator sorts through the tables populated with client and ADSBx data to find non-matching entries and indicate a potential problem. This correlation result is returned and posted to the database as a CorrelatedFlight [11] object.

The Importers and their corresponding data objects are described below:

The ADSBImporter [3] is one of three importers that the ImportExecutor could invoke. The ADSBImporter is responsible for downloading a days' worth of data from the ADSBx website via the ADSBDownloader [4]. Once ADSBDownloader gets called, it downloads data from ADSBx as a .zip file, extracts and filters the information as ADSBData [5], and returns the data back to ADSBImporter.

The CEDASImporter [7] is the second of three importers that the Executor could invoke. The CEDASImporter is responsible for extracting client provided data in the form of Excel or JSON files and adding this information to a database. These data are encapsulated in CEDASUpload [9] objects.

The TailImporter [8] is the third Importer the Executor could invoke. The TailImporter is responsible for filtering aircraft tail numbers, containing Honeywell HTF7K engines, from a user provided JSON list. This importer, however, is different from the other importers because it does not interact with the Correlator. Instead the TailImporter adds relevant tail numbers to a database to be used as a reference database for the ADSBImporter and CEDASImporter to identify and extract information from the individual aircrafts we are interested in.

A more detailed description of the components AppEngine, ImportExecutor, ADSBImporter, ADSBDownloader, CEDASImporter, TailImporter, and Correlator are explained in the next sections.

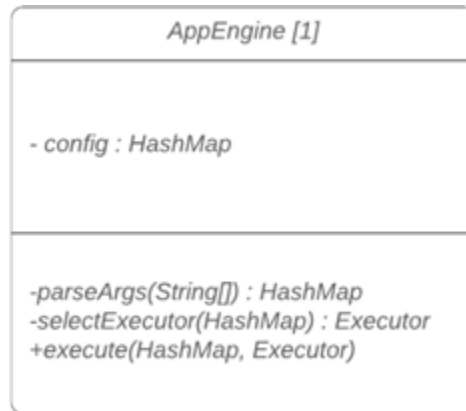


Figure 3: UML of AppEngine Component

App Engine

AppEngine is the start of the program execution and will execute a path according to the command-line arguments passed in. Figure 3 outlines the methods associated with AppEngine including parameters and return values. AppEngine contains *main()* which has a parameter of type *String[]*. These argument parameters get passed in to *parseArgs()* and returns a *HashMap* of the parsed run-configuration. The keys represent the flag that was passed to the command line, and the value represents the content of that flag. When the application starts, it creates a new instance of *ImportExecutor* which takes in the parsed run-configuration. During the *execute()* method, we use this *ImportExecutor*'s *execute()* function which begins the database import based on the given run-configuration.



Figure 4: UML of Import Executor Component

ImportExecutor

The *ImportExecutor* is the template for *ADSBImporter*, *CEDASImporter*, and *TaillImporter*. Figure 4 outlines the template methods and data fields for *ImportExecutor*. The fields include a connection to MongoDB for data correlation and a *HashMap* configuration determining what type of data is being accessed. Each *Importer* has a constructor which initializes the *config* field using the *HashMap* passed as an argument. The *makeConnection()* function results in a MongoDB connection and this is what pre-creates the database

connection that will be used later in the importers and Correlator. This database connection is important in allowing the data JSON files downloaded from the Importers to be added to MongoDB and for the Correlator to extract information to correlate. The `execute()` function calls the correct Importer depending on the `config` member's options, then, upon success, calls the Correlator to correlate the newly imported data. It returns a Boolean indicating whether the path it took was successful.

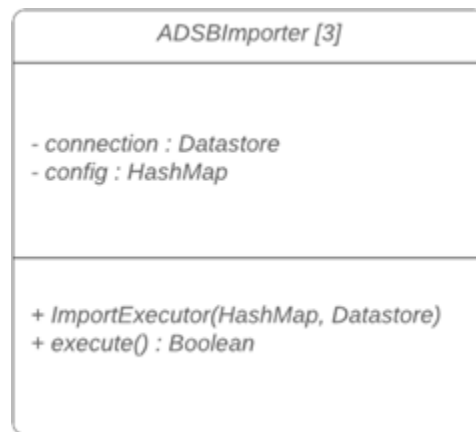


Figure 5: UML of ADSBImporter Component

ADSBImporter

The ADSBImporter is an object created from the ImportExecutor template. It contains the same attributes and methods as the template and uses ADSBDownloader to download and extract the ADSBx data. ADSBx data includes longitude and latitude coordinates of landing sites of all planes that have been recorded by ADSBx's vast network of public receivers. Figure 5 organizes the methods and data fields associated with ADSBImporter, which is mirrored from ImportExecutor.

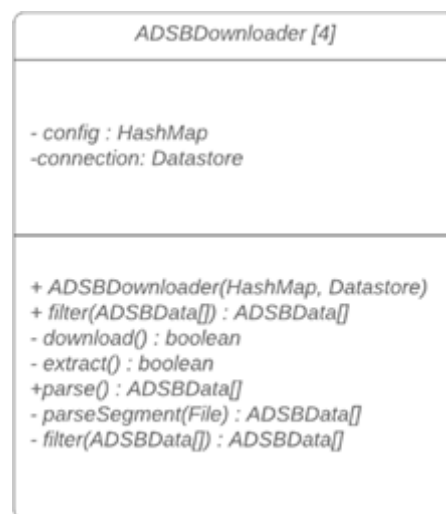


Figure 6: UML of ADSBDownloader Component

ADSBDownloader

ADSBDownloader is the helper class invoked from ADSBImporter which is outlined in Figure 6. ADSBDownloader downloads a .zip file from ADSBx via download() and returns a boolean value of download success. The extract() function, unzips the file and extracts the information and returns a boolean value of success. The parse() method reads all files in the extracted directory created by the extract() function, and passes them to parseSegment() which collects the result from all files and returns it. The parseSegment() function returns an array of ADSBData parsed from the file it was passed after being filtered by the filter() function. The filter() function takes all of the data of a file and removes data for aircrafts that are not in the tail number database, then returns the filtered data.

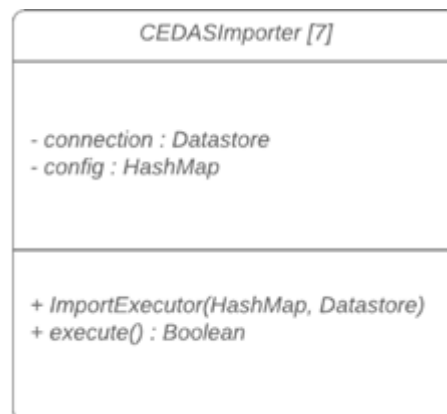


Figure 7: UML of CEDASImporter Component

CEDASImporter

The CEDASImporter is an object created from the Importer interface that imports the upload LON/LAT data, which contains longitude and latitude coordinates of previously recorded HTF7K engine diagnostic uploads, was imitated and provided in an Excel document format. The CEDASImporter will download this data as an XLS file that will be converted to a JSON file through the function executeEXECL() as a CEDASUpload object. Like ADSBImporter, Figure 7 shows the same attributes and methods as the Importer template. Functionally, the CEDASImporter class is similar to the ADSBImporter parse() and filter() functions, except they accept and return CEDASData.

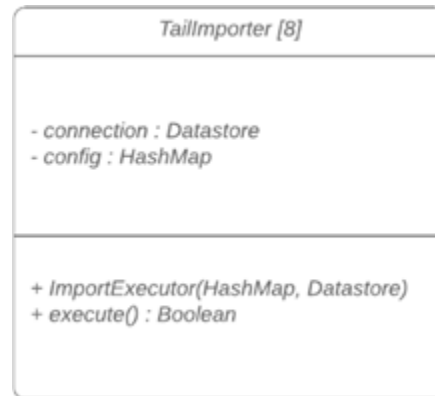


Figure 8: UML of TailImporter Component

TailImporter

The TailImporter is another object created from the Importer interface. It contains the same attributes and methods as the ImportExecutor and imports data from user-provided tail number files. This tail number file, type TailNumber, contains the tail numbers of aircrafts containing an HTF7K engine and is downloaded as a JSON file. Figure 8 displays TailImporter as a UML component similar to CEDASImporter and ADSBImporter.

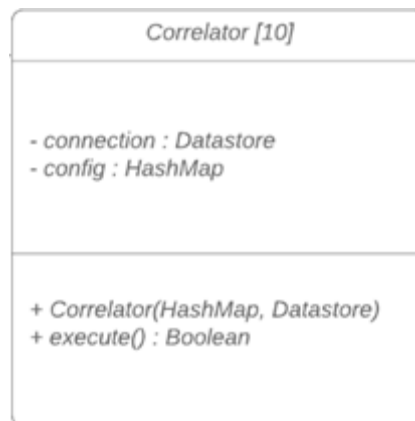


Figure 9: UML of Correlator Component

Correlator

The Correlator is responsible for correlating data in the database and must have a connection between MongoDB. It takes in the connection that was made from the ImportExecutor. Upon running execute(), it begins to correlate new data in the database, which results in the creation of new CorrelatedFlight objects, and adds them to the database. Figure 9 outlines the fields and methods of Correlator as a UML component.

Figure 10 shows a screenshot of the CorrelatedFlight object created from the Correlator. It includes a tailnumber, landingPoint, takeoffPoint, landingDate, and outcome of the correlation. The landingPoint and takeoffPoint are JSON Objects that are of type LatLong and have two fields: "type" which is always set to "Point" and "geometry" which is an array of length two containing floats that represents the latitude and longitude coordinates of the point. This array representation of LON/LAT coordinates is like the GeoJSON specification of these fields.

```
  className: "edu.nau.enginair.models.CorrelatedFlight"
  tailNumber: "N672WM"
  ✓ landingPoint: Object
    type: "Point"
    ✓ geometry: Array
      0: -95.29151153564453
      1: 29.991790771484375
  > takeoffPoint: Object
    landingDate: 2016-08-01T02:23:51.619+00:00
    outcome: "FAIL_NO_WIFI_AIRCRAFT"
```

Figure 10: CorrelatedFlight JSON File Outline

The outcome field contains the upload status correlation result which is may be one of the following:

- SUCCESS_UPLOAD: Indicates a successful upload with correct WAP credentials and confirmed landing location.
- FAIL_NO_LANDING: Indicates that the aircraft took off but did not record a landing within 24 hours of takeoff (i.e. aircraft may have flown internationally or landed in an area not canvased by ADSBx receivers).
- WARN_IN_PROGRESS: Indicates that the flight was in-progress during the ADSBx download procedure; Flight must be reverified after the next ADSBx download.
- FAIL_NO_WIFI_AIRPORT: Indicates a failed upload due to missing WiFi connection at an airport (i.e. airport has not previously recorded any successful uploads and/or no aircraft has the airport's WiFi configuration).
- FAIL_NO_WIFI_AIRCRAFT: Indicates a failed upload due to missing WiFi connection configuration on aircraft (i.e. current aircraft has failed to upload at an airport location where previous uploads from other aircrafts have occurred).
- FAIL_DEAD_EDG100: Indicates a failed upload due to a broken EDG100 upload mechanism onboard the aircraft (i.e. aircraft failed to upload three times consecutively at a location where aircraft has had previous successful uploads).
- FAIL_WAP_CHANGED: Indicates a failed upload due to changed or unconfigured airport WAP credentials (i.e. aircraft has had successful uploads at a specified airport previously but experienced a current failed upload and has been able to successfully upload at other airports).

6.2. Front end Module

The GUI interface is responsible for presenting information to the user in an easily understandable way. Our system's end users are engine technicians and aircraft pilots, each with their own set of webpages, as outlined in Figure 11.

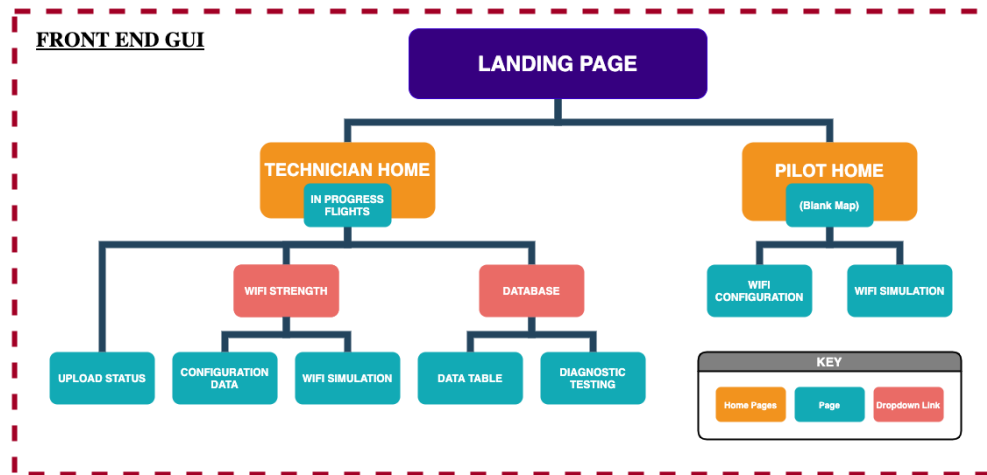


Figure 11: UML Diagram of the Frontend GUI

The GUI is designed to be easy to use and complete the requirements described as user stories depending on what each user would need to know from the system. The subpages of each user will be described in more detail in the following sections.

Landing Page

The landing page, shared by both users, is the initial page loaded when CEUSS is launched. A screenshot of this page is shown in Figure 12. The landing page lists the generic functionality under each user which, when clicked, will redirect to the corresponding pages.

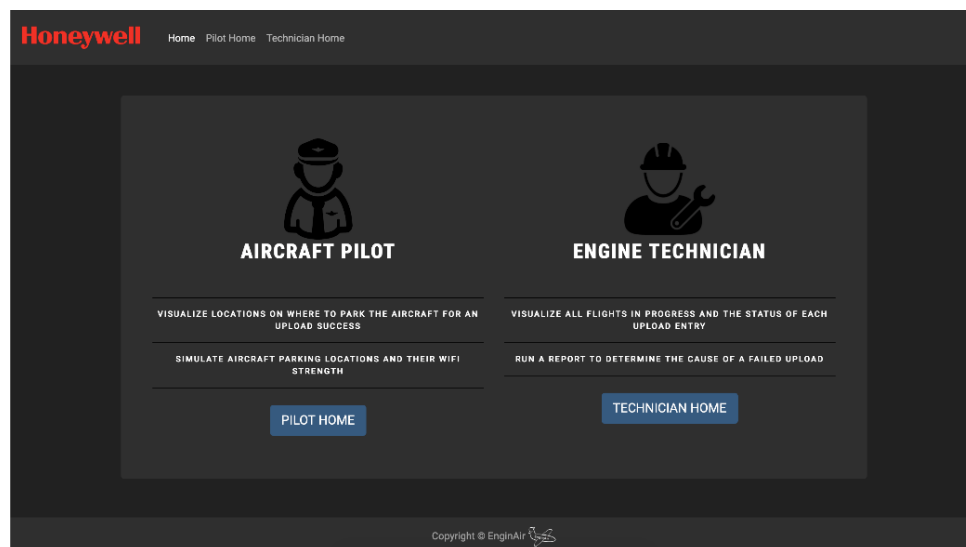


Figure 12: Screenshot of CEUSS Landing Page

6.2.1. Technician

The engine technician front end web page architecture consists of six pages: Home, Upload Status, WiFi Configuration, WiFi Simulation, Database Table, and Database Diagnostic Testing. These six pages provide the GUI functionality for all technician user stories. The web page outline is illustrated in Figure 13 and each page is explained in accordance to the user story it completes. The user stories are listed below and numbered for reference.

As an engine technician, I want to be able to:

- visualize all flights that are currently in progress [1].
- visualize the status of each upload entry [2].
- view all aircraft landing locations, every 24 hours [3].
- simulate various locations and their corresponding WiFi configuration [4].
- know the status of each landing/upload entry [5].
- run a report to determine the cause of a failed upload [6].

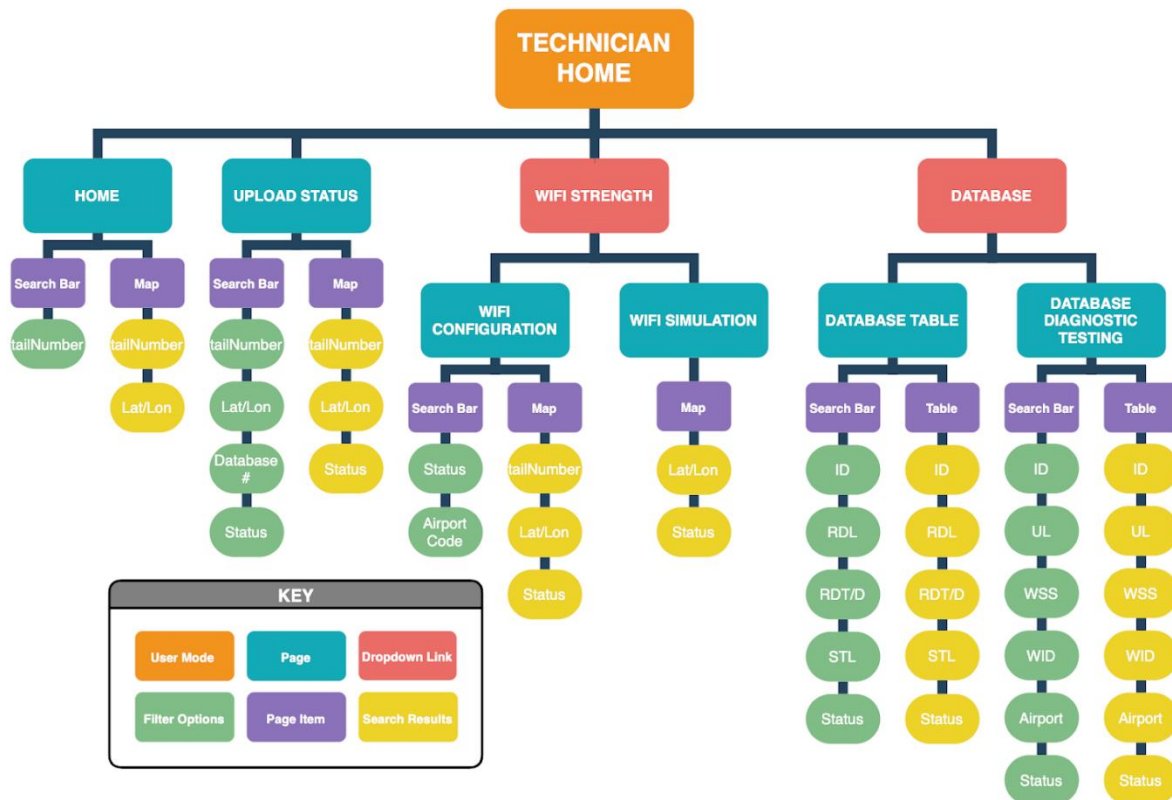


Figure 13: Webpage Diagram of the Technician Web Pages

Home

The Home page displays a mapping of current flights in progress [1]. This data comes from ADSBx flight feeders which will be integrated with the OpenStreetMap API. This mapping is contained to flights within the U.S. and each flight on the map is indicated using a white dot. Each individual flight on the map can be searched via its tail number or clicked which will display flight information like tail number and current longitude and latitude coordinates. This map is updatable every 24 hours as our data collection only occurs every 24 hours. Figure 14 shows a screenshot of this page and figure 15 illustrates the result of a search by tail number.

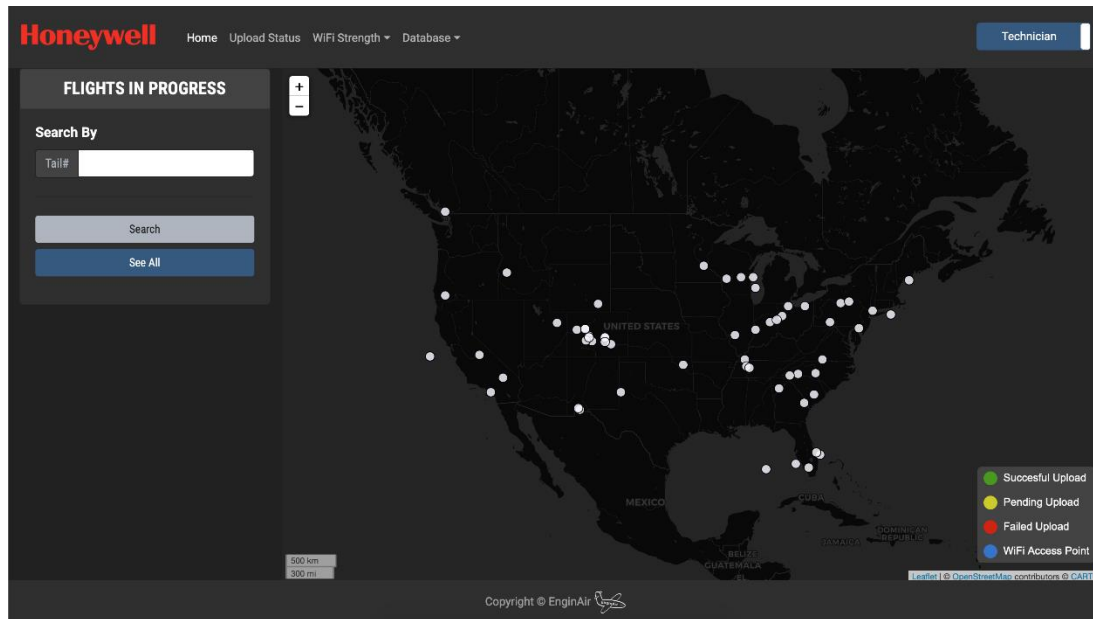


Figure 14: Screenshot of Technician Home Page

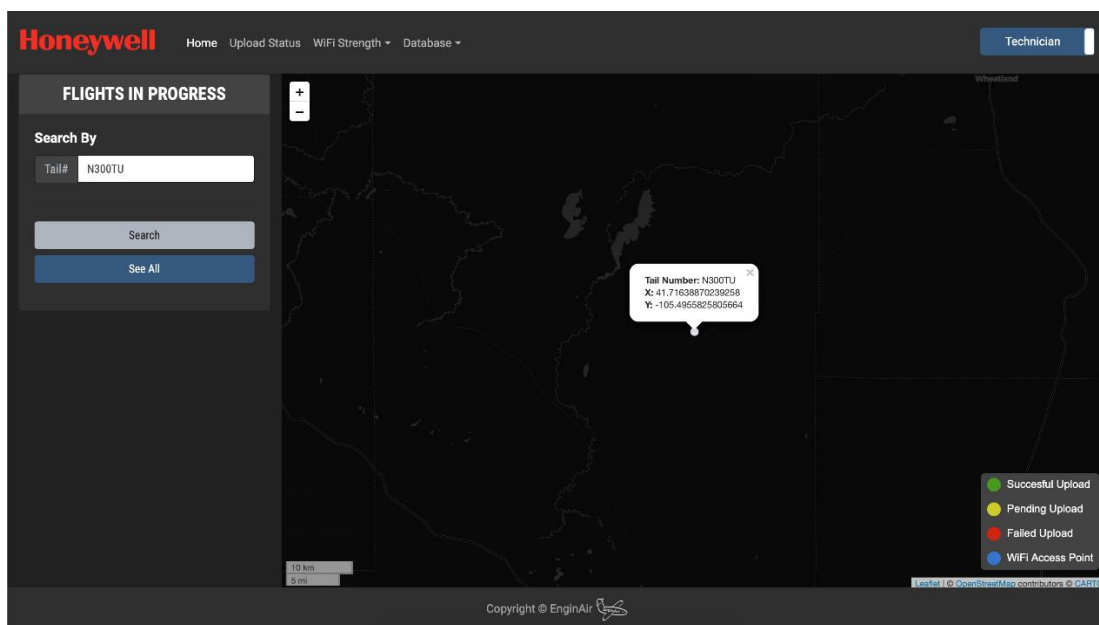


Figure 15: Screenshot of Searching a Tail Number on the Home Page

Upload Status

The Upload Status page displays upload entries as colored dots on a map correlating to the status of the upload at a location [2]. A green dot indicates “Success”, yellow indicates “Pending”, and red indicates “Failed”. The technician can filter the results based on status using the checkbox field or by specifying LON/LAT coordinates. Other filters include tail number and database entry number. Once a specific entry is located on the map and selected, the system will display information regarding the upload entry such as X and Y coordinates, tail number, upload status. This mapping feature inherently indicates landing locations with successful and failed upload statuses since an upload entry occurs on landing and corresponds to its location [3]. Figure 16 illustrates a screenshot of the Upload Status page.

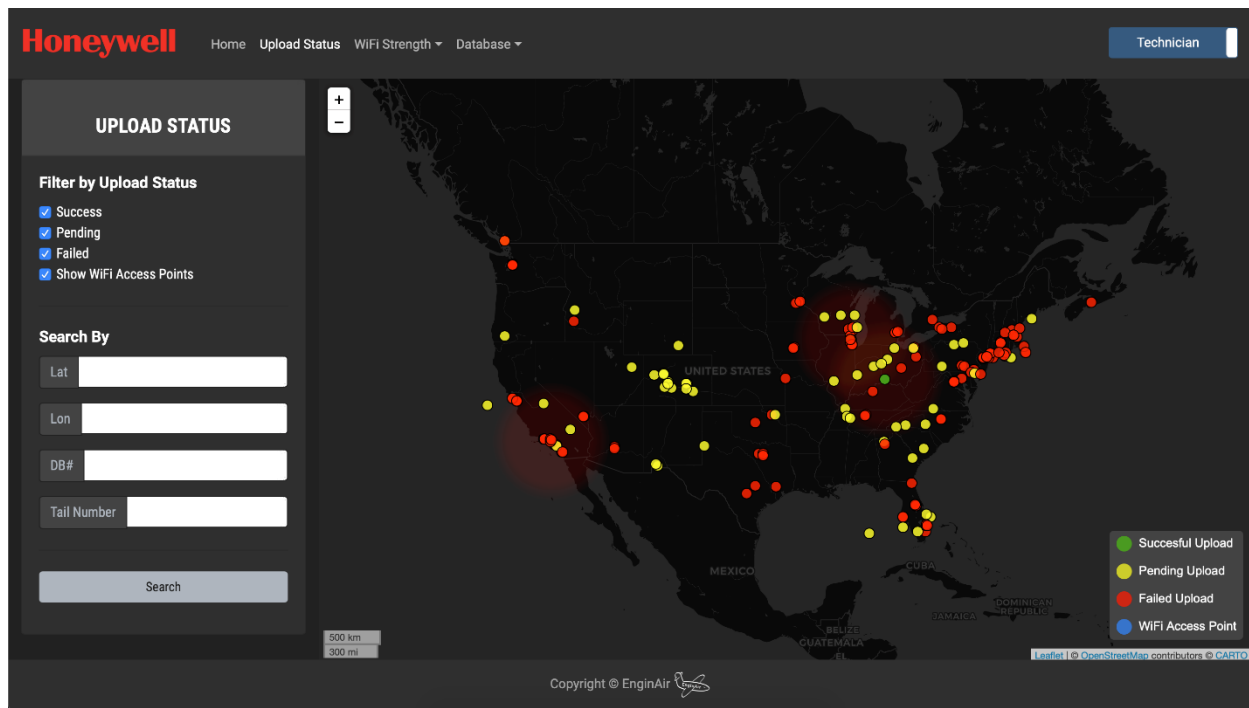


Figure 16: Screenshot of Upload Status Page

WiFi Configuration

The WiFi Configuration page displays all documented WiFi configurations and upload statuses. This page allows technicians to visualize upload statuses in proximity to known WiFi hotspots. These hotspots are illustrated as heat maps indicating the WiFi range of connection and the uploads are classified by color depending on whether the upload was a success, is pending, or failed. Figure 17 shows this page.

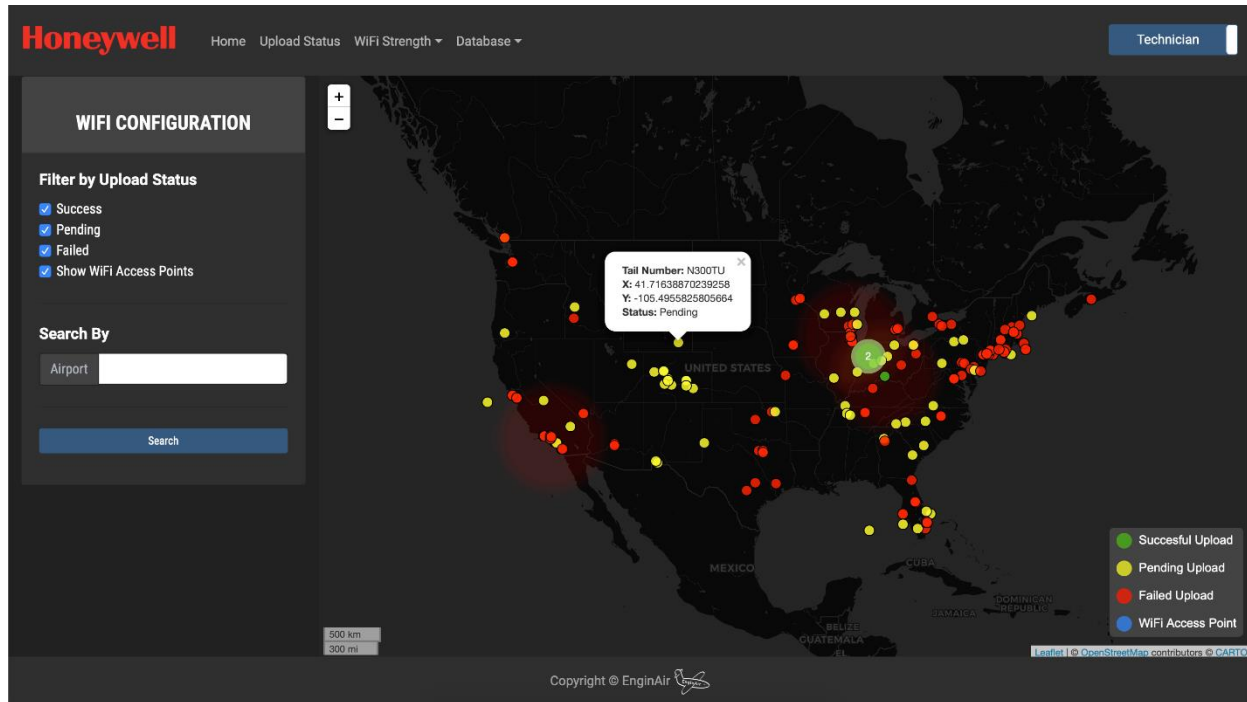


Figure 17: Screenshot of WiFi Configuration Page

The user has the option to sort by upload status using checkboxes on the left of the screen. This option will filter the colored dots by the corresponding. When an individual status is clicked, the system will show the tail number, the coordinates, and the status of the upload. The user also has the option to filter by airport code which will show all WiFi configurations and uploads that occurred at the specified airport. In figure 17, there are only 3 WiFi configurations in the database for this project's scope but in theory the map would be covered with heatmaps indicating all the WiFi configurations located at the airports in which these uploads occur.

WiFi Simulation

The WiFi Simulation page is responsible for predicting a CEDAS upload success according to a selected location [4]. As an engine technician, this is important in determining locations that lack or have insufficient WiFi connections in order to be aware of where uploads may not occur. It also may be helpful in determining where WiFi connections should be established to obtain a wider range of coverage or increase strength of WiFi connection. The user will select a point on the map to indicate a specific XY coordinate of a potential upload location and the system will correlate those coordinates to the nearest WiFi configuration. The system will then display the result of the potential upload as “Success” or “Failed” given the location and the WiFi strength. If the prediction is “Success”, the system will display the upload location coordinates, the three-letter airport code which the upload would occur at, and the SSID password of the WiFi heat map that would process this upload. If the prediction is “Failed”, the system will display the coordinates at which this upload would fail. This page also displays the located WiFi configurations to visually see the potential upload location compared to the WiFi heat map. Figure 18 shows a screenshot of a failed upload prediction.

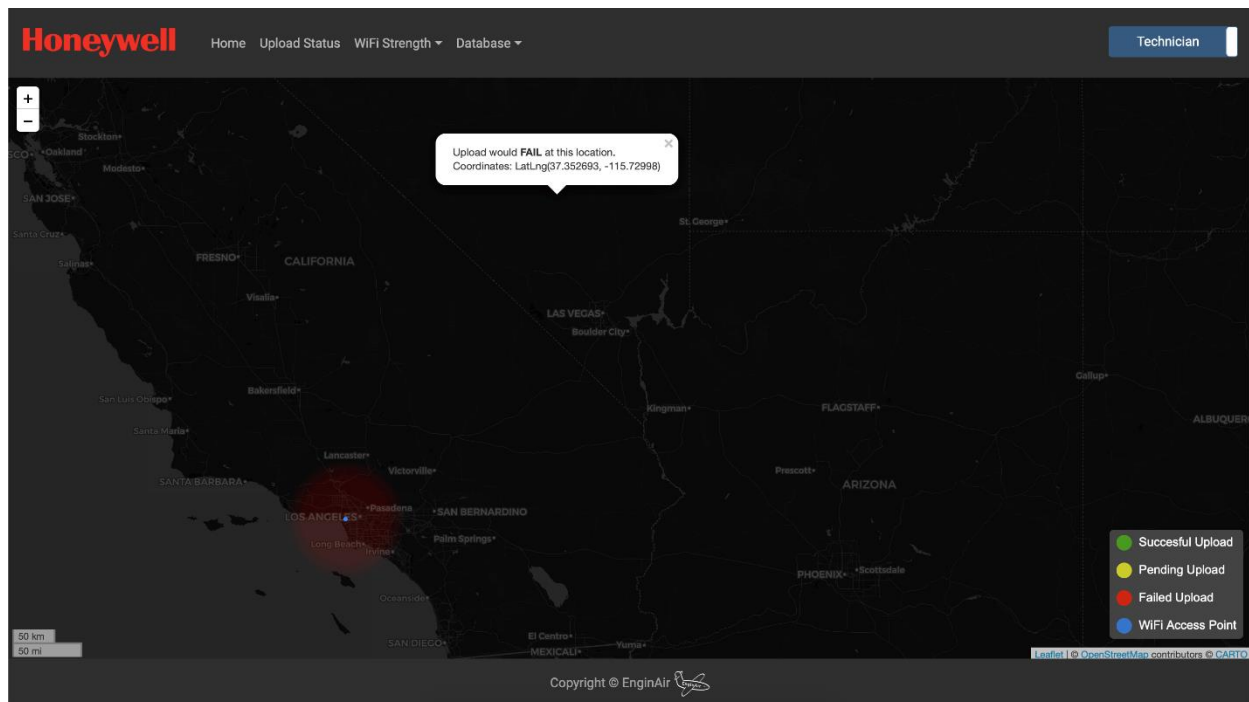


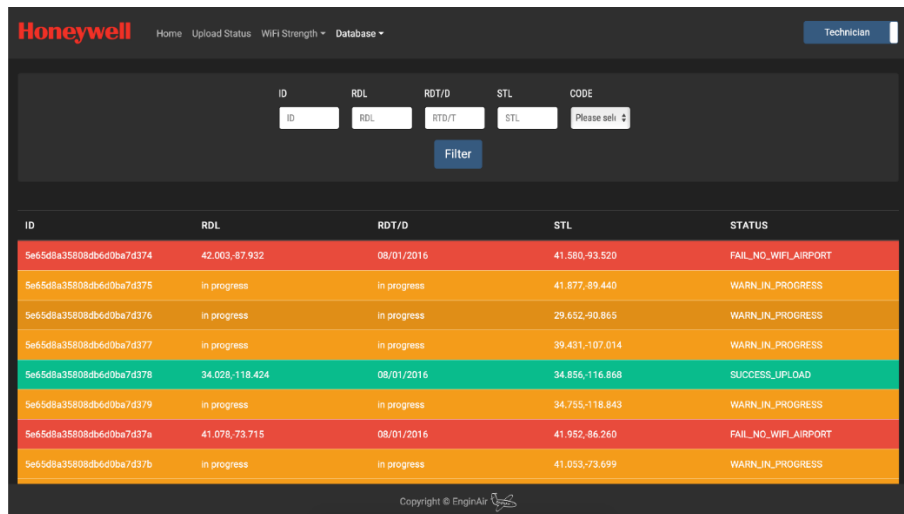
Figure 18: Screenshot of a Failed Upload Prediction

Database Table

The Database Table page is primarily used to visualize each database upload entry [5]. Using the dropdown boxes, the technician can filter out specific entries to view all information associated with that entry. The filter fields are described below:

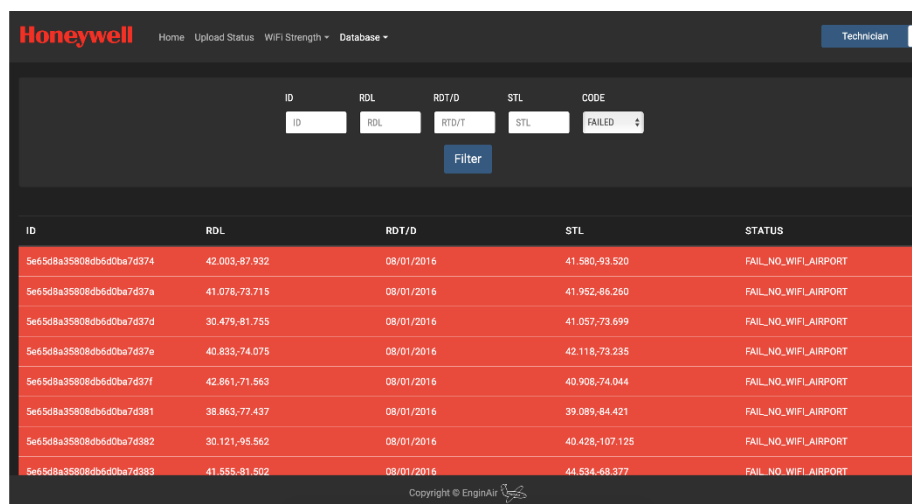
- ID: Unique identification number for each database entry
- RDL: Engine Roll-Down GPS Location
- RDT/D: Engine Roll-Down Time and Date
- STL: Engine Start up GPS Location
- CODE: Status of upload entry (Success, Pending, Failed)

Figure 19 illustrates a screenshot of this database page. The CODE status for each upload entry is color-coded using green to indicate successful, orange to indicate pending, and red indicating failure. Figure 20 illustrates the database filtered by failed upload statuses. These failed statuses can be either FAIL_NO_WIFI_AIRPORT, FAIL_NO_WIFI_AIRCRAFT, FAIL_DEAD_EDG100, and FAIL_WAP_CHANGED.



ID	RDL	RDT/D	STL	STATUS
5e65d8a35808db6d0ba7d374	42.003,-87.932	08/01/2016	41.580,-93.520	FAIL_NO_WIFI_AIRPORT
5e65d8a35808db6d0ba7d375	in progress	in progress	41.877,-89.440	WARN_IN_PROGRESS
5e65d8a35808db6d0ba7d376	in progress	in progress	29.652,-90.865	WARN_IN_PROGRESS
5e65d8a35808db6d0ba7d377	in progress	in progress	39.431,-107.014	WARN_IN_PROGRESS
5e65d8a35808db6d0ba7d378	34.028, 118.424	08/01/2016	34.856, 116.868	SUCCESS_UPLOAD
5e65d8a35808db6d0ba7d379	in progress	in progress	34.755,-118.843	WARN_IN_PROGRESS
5e65d8a35808db6d0ba7d37a	41.078,-73.715	08/01/2016	41.952,-86.260	FAIL_NO_WIFI_AIRPORT
5e65d8a35808db6d0ba7d37b	in progress	in progress	41.053,-73.699	WARN_IN_PROGRESS

Figure 19: Screenshot of Database Table Page



ID	RDL	RDT/D	STL	STATUS
5e65d8a35808db6d0ba7d374	42.003,-87.932	08/01/2016	41.580,-93.520	FAIL_NO_WIFI_AIRPORT
5e65d8a35808db6d0ba7d37a	41.078,-73.715	08/01/2016	41.952,-86.260	FAIL_NO_WIFI_AIRPORT
5e65d8a35808db6d0ba7d37d	30.479,-81.755	08/01/2016	41.057,-73.699	FAIL_NO_WIFI_AIRPORT
5e65d8a35808db6d0ba7d37e	40.833,-74.075	08/01/2016	42.118,-73.235	FAIL_NO_WIFI_AIRPORT
5e65d8a35808db6d0ba7d37f	42.861,-71.563	08/01/2016	40.908,-74.044	FAIL_NO_WIFI_AIRPORT
5e65d8a35808db6d0ba7d381	38.863,-77.437	08/01/2016	39.089,-84.421	FAIL_NO_WIFI_AIRPORT
5e65d8a35808db6d0ba7d382	30.121,-95.562	08/01/2016	40.428,-107.125	FAIL_NO_WIFI_AIRPORT
5e65d8a35808db6d0ba7d383	41.555,-81.502	08/01/2016	44.534,-68.377	FAIL_NO_WIFI_AIRPORT

Figure 20: Screenshot of Database Page Filtered by Failed Status

Database Diagnostic Testing

The Database Diagnostic Testing page allows the technician to run a “test” on the failed upload entries to determine the cause of failure [6]. This test produces additional information regarding each of the 4 types of failure statuses. Figure 21 shows this diagnostic test page and the 4 types of statuses to search from. After clicking on the specified failure status, the system may prompt the technician for specifics relating to the status such as airport code. The system will then display a results page containing a summary of findings according to predefined conditions listed in the Requirements Document.

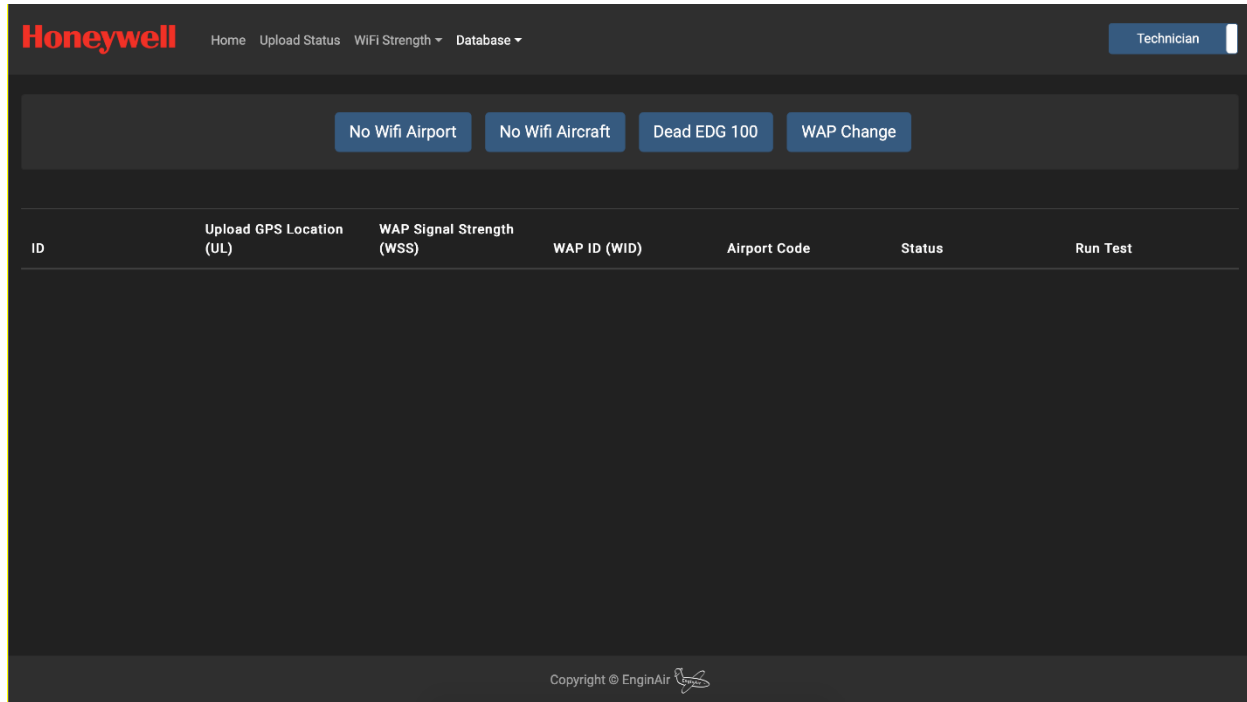


Figure 21: Screenshot of Database Diagnostic Testing Page

6.2.2 Aircraft Pilot

The aircraft pilot front end web page only consists of two main pages, Flight Upload Simulator and Mapping, as opposed to four. These two pages contain the GUI functionality for the pilot user stories. These pages are broken up into functionality according to the user story and is illustrated in the web page outline shown in Figure 22. The user stories are listed below and numbered for reference.

As an aircraft pilot/operator, I want to be able to:

- visualize locations on where to park the aircraft for an upload success [7].
- simulate locations and their WiFi strength [8].

*Note: these user stories are extremely similar but are different approaches to the same result.

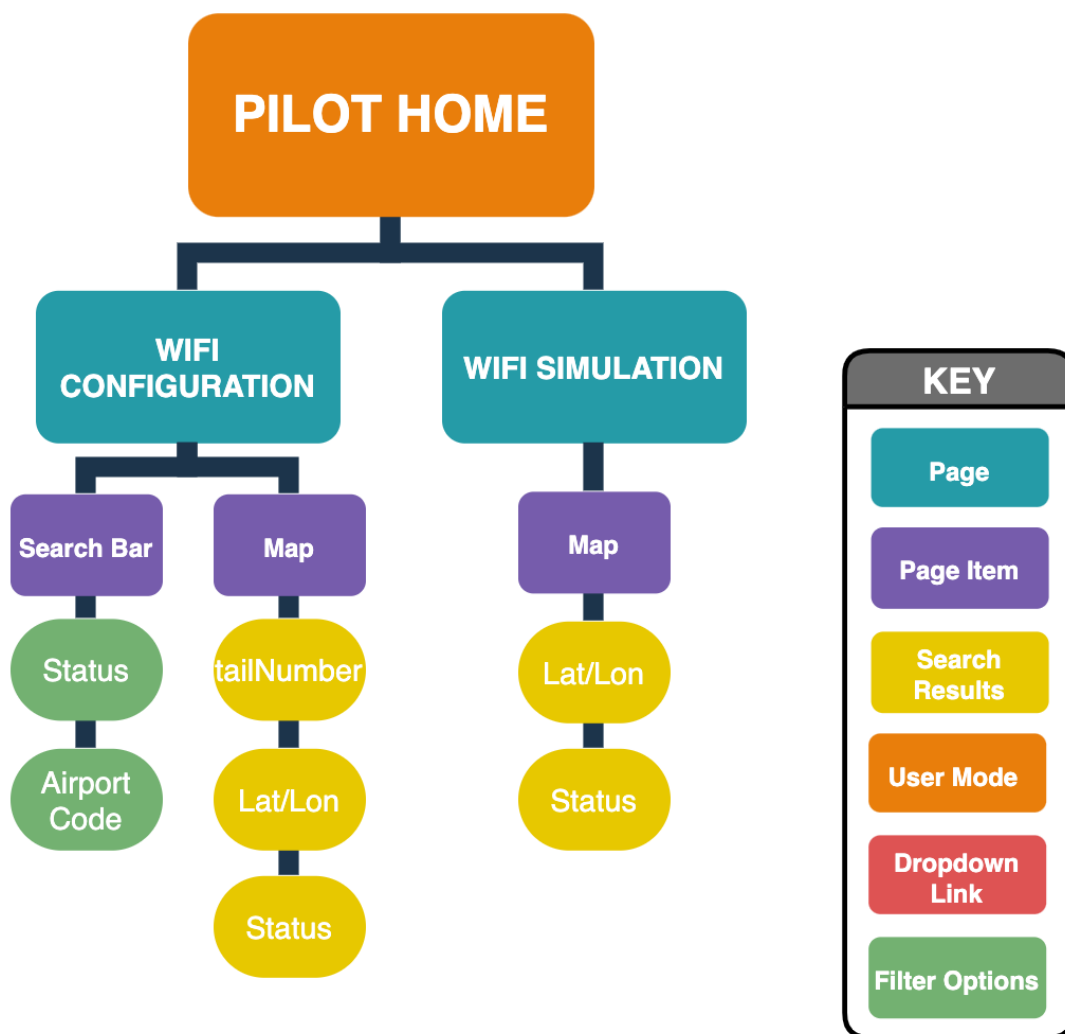
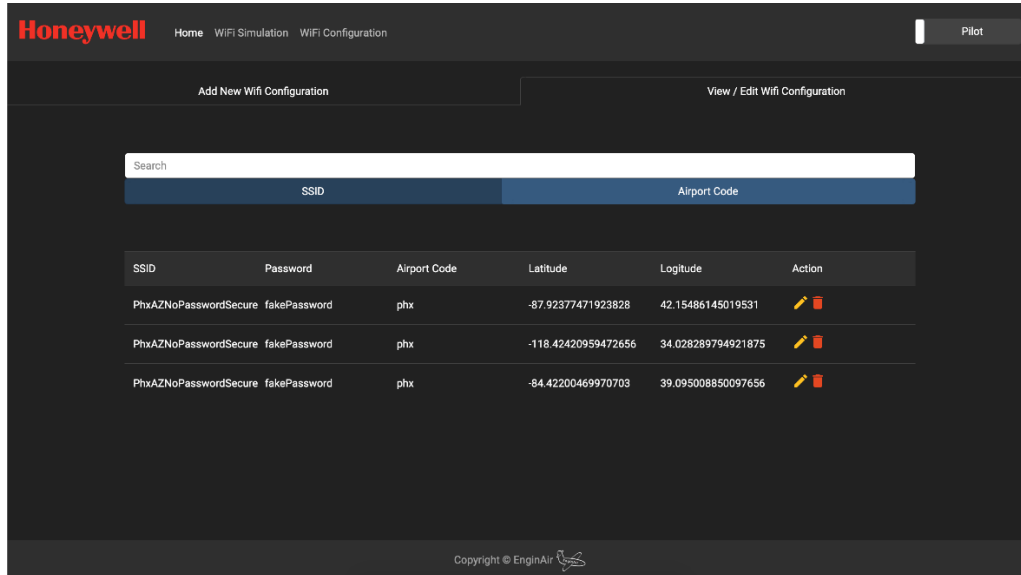








Figure 22: Webpage Diagram of the Technician Web Pages

Home

This Pilot Home page contains WiFi password configurations. Pilots can identify these WiFi hotspots via location coordinates and can connect using the password to upload their CEDAS reports. Pilots can search WiFi configurations based on airport code or specific SSID numbers. Figure 23 illustrates this feature of the Home page.



The screenshot shows the Honeywell Pilot Home page. At the top, there are navigation links: Home, WiFi Simulation, WiFi Configuration, and a Pilot button. Below these, there are two tabs: 'Add New Wifi Configuration' and 'View / Edit Wifi Configuration'. The 'View / Edit Wifi Configuration' tab is active, showing a search bar and a table of WiFi configurations. The table has columns for SSID, Password, Airport Code, Latitude, Longitude, and Action. The data is filtered by the airport code 'phx'.

SSID	Password	Airport Code	Latitude	Longitude	Action
PhxAZNoPasswordSecure	fakePassword	phx	-87.92377471923828	42.15486145019531	 
PhxAZNoPasswordSecure	fakePassword	phx	-118.42420959472656	34.028289794921875	 
PhxAZNoPasswordSecure	fakePassword	phx	-84.42200469970703	39.095008850097656	 

Copyright © EnginAir

Figure 23: Screenshot of Pilot Home Page Filtered by Airport Code

WiFi Configuration

The WiFi Configuration page is specifically designed for visualization of WiFi configurations [8]. Pilots can view WiFi heatmaps in proximity to upload statuses across the US. The user can narrow the search using an airport code or filter by upload status checkboxes. Figure 24 shows a screenshot of this page.

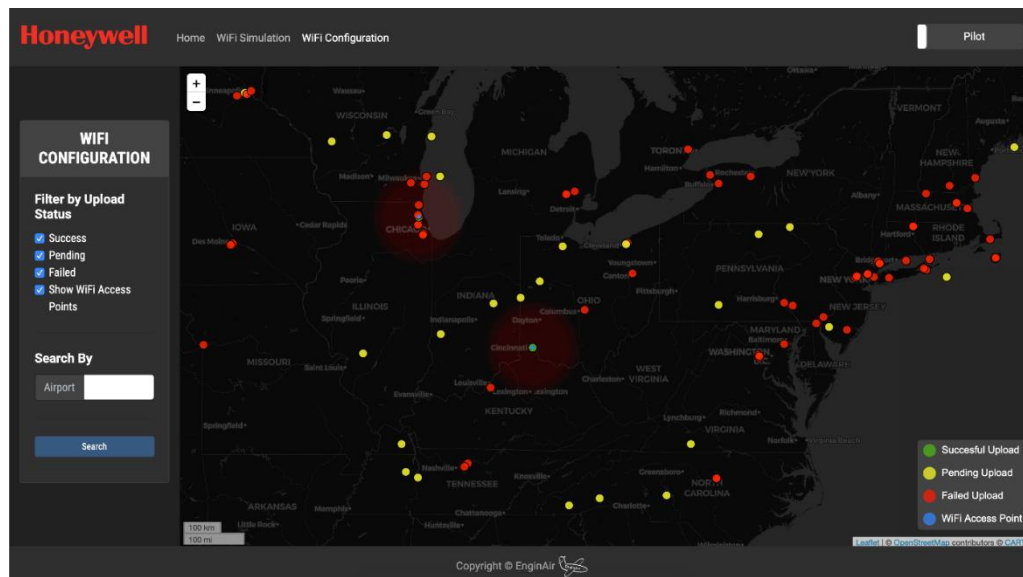


Figure 24: Screenshot of WiFi Configuration Page

WiFi Simulation

The WiFi Simulation is responsible for allowing the user to simulate parking locations to predict upload status based on WiFi connectivity in the area [8]. This page's functionality is identical to the Technician's Simulation page. By clicking on a location on the map, the system will determine whether the upload will be successful. The system will display exact longitude and latitude coordinates of the failed or successful upload prediction which pilots can then use to park the aircraft. Figure 25 shows a screenshot of the WiFi Simulation page and figure 26 displays the results of a successful prediction.

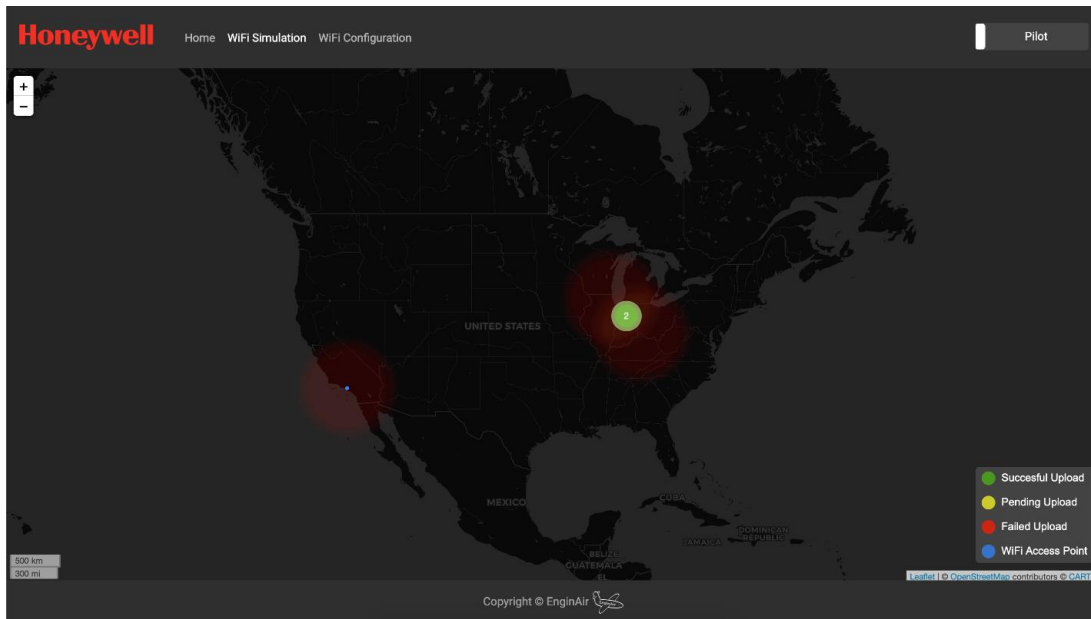


Figure 25: Screenshot of Pilot Simulation Page

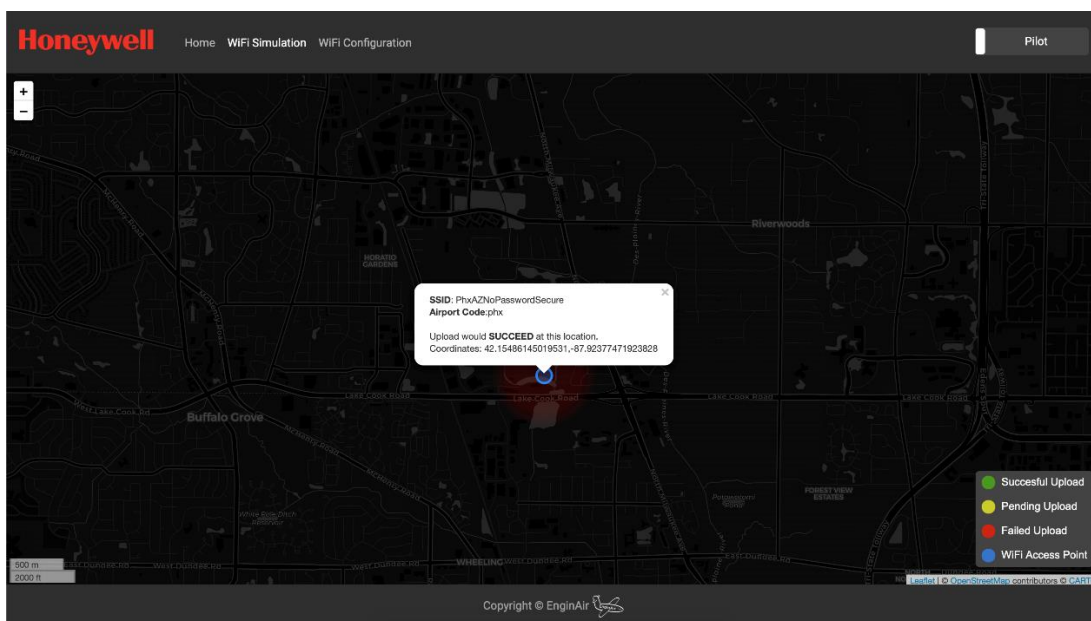


Figure 26: Screenshot of a Successful Upload Prediction from the Simulation

7. As-Built vs As-Planned

Overall, our “as-planned” product is essentially our “as-built” product in terms of functionality. This functionality was planned from last semester and did not go through any extensive changes. Some aesthetic and GUI web page layout changes were shown in the previous front end module section. The original web page designs can be found in the Software Design Document.

8. Testing

Like any software system, CEUSS underwent 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. The following sections outline the testing procedures for unit, integration, and usability testing for CEUSS.

8.1. Unit Testing

To assist in our execution of unit testing, we utilized 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 measured test coverage using CodeCov and JUnit. CodeCov is an open source automatic coverage report generator that supports Java and JavaScript. JUnit includes a built-in code coverage functionality which helped us reach 100 percent code coverage tests.

Because we utilized object-oriented programming techniques, methods are the smallest units we tested. Table 3 lists out all the classes whose methods will have a corresponding unit test. 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. The Software Test Plan document outlines a detailed test description and expected test results for each method in these classes.

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

Table 3: Data Import and Correlation Classes Subject to Unit Testing

8.2. Integration Testing

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 of CEUSS focuses on the interactions between the database component and the `web-app/`. This testing simulated a web browser and verified that web API calls correctly accessed the server backend and returned accurate results. Because the Data Import and Correlation components were written and executed using external libraries and platforms, we refrained from testing those 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. Figure 27 below shows this module integration.

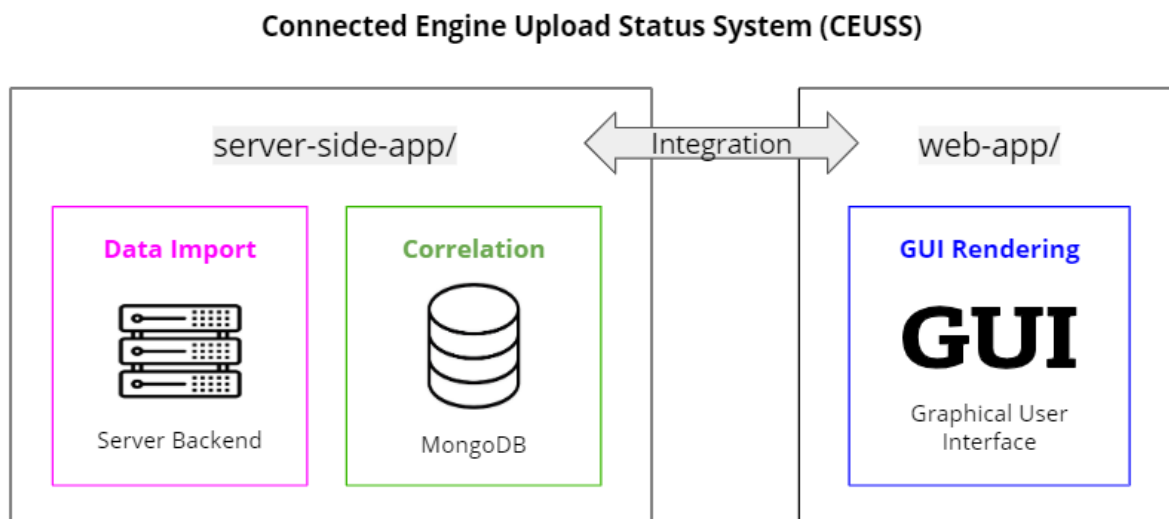


Figure 27: CEUSS Module Integration

To automate this process, we used the Postman testing utility, a collaboration platform for API development, used to simulate API requests to debug and inspect the responses. These API requests were in the form of a database search query. The API web client helped to verify and validate that the API returned responses were what was expected. Figure 28 illustrates the following circumstances of an expected API response:

- If an API call returns an empty response, the result should be an empty JSON array.
- 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 with a matching a predefined JSON format as specified in the `server-side-app/`

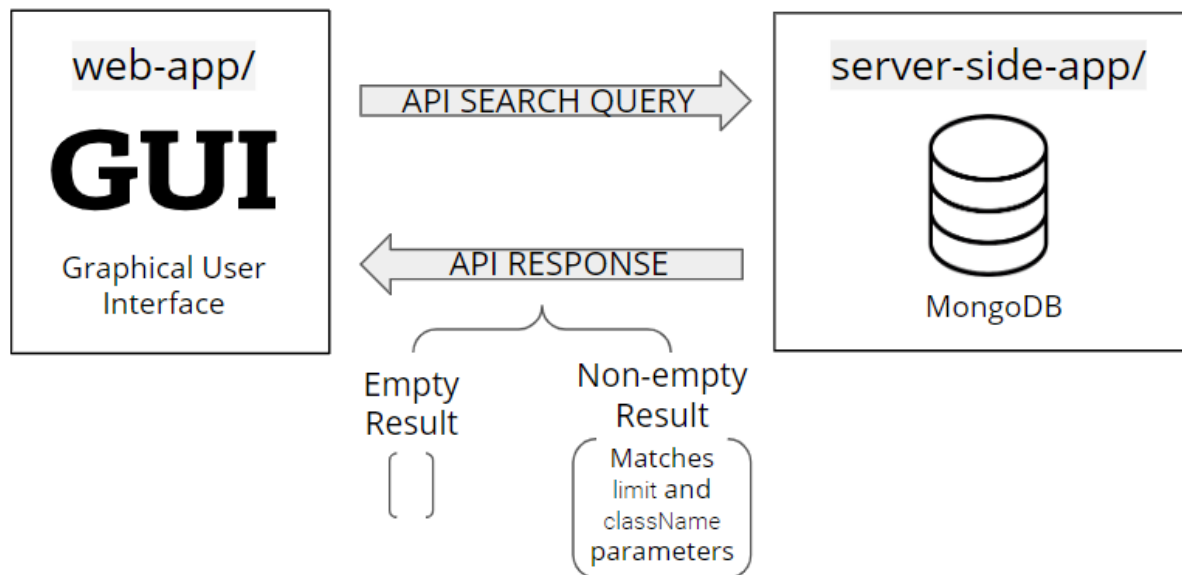


Figure 28: Detailed Integration via API Calls

8.3. Usability Testing

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 used 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 was executed on each webpage URL which generated individual reports. These reports contained 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 were made, the audit was executed again and repeated until all audits were passing.

After our client's first view at the alpha prototype, he asked if we would be able to utilize more data to make the system feel more "real". There were emails exchanged about our client providing us real CEDAS files but after viewing the upload files and estimating the time investment, we decided to add this type of data integration into a future works, version 2.0 project. There were also emails exchanged between a representative from ADSBx and us regarding installing a receiver on NAU campus in exchange for access to the full ADSBx database. Like the CEDAS upload data, given the time left in the semester, we decided that a full ADSBx integration would be part of a version 2.0 project.

9. Project Timeline

Our project commenced in the Fall 2019 semester starting with preliminary documentation and team introductions. This included the team standards document which outlined team member roles, expectations, and development tools. Requirements acquisition and feasibility testing followed initial client communication and meetings were conducted. After the requirements document was accepted and signed by our client, development and integration began in the Spring 2020 semester. The backend software was developed first, followed by the front-end GUI development and integration. The alpha prototype was demonstrated to our client mid-semester which led to testing and GUI refinement thereafter. The full beta prototype and final documentation, along with required UGRADS presentations and posters, concluded the semester and the project. Figure 29 illustrates a detailed weekly schedule of the individual tasks completed over the semester.

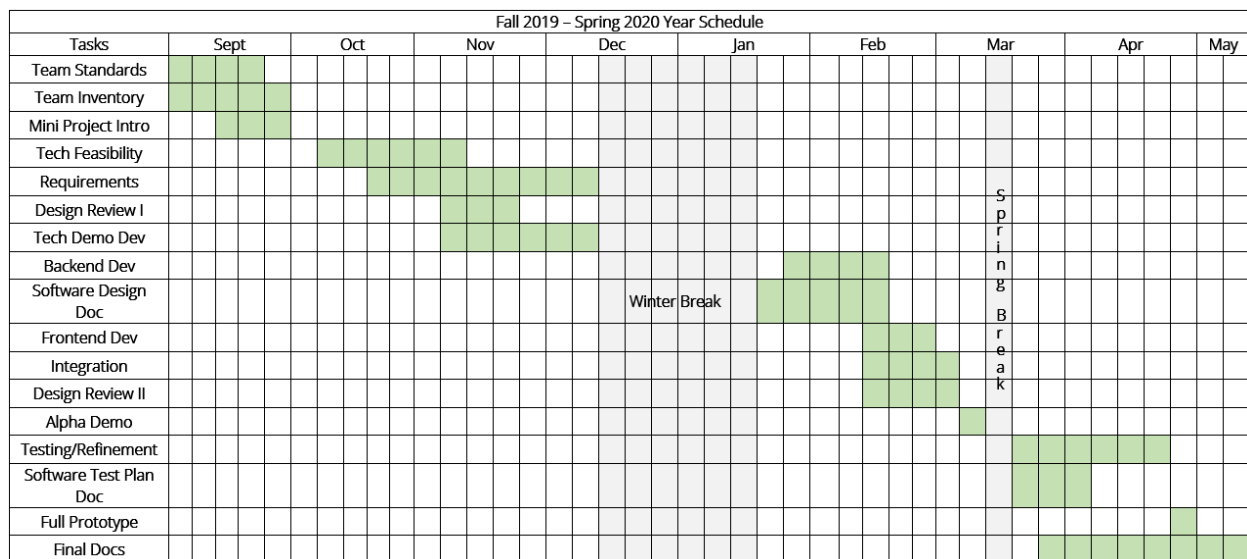


Figure 29: Timeline of Project

In terms of managing workload, Megan spearheaded all things pertaining to documentation and presentations. This included the requirements, software design, and software test plan documents as well as design review presentations. Ian, our scrum master, oversaw all things pertaining to code development ranging from managing the GitHub repository pushes and delegating workload. Chloe, Gennaro, and Dylan helped with coding and documentation as needed and where assigned.

We are currently on schedule to complete all required assignments for this project and are projected to complete the project signoff on Tuesday, April 28th.

10. Future Work

Although we had completed all the requirements for this capstone project, we brainstormed ideas that a potential CEUSS version 2.0 could benefit from having. These features are outlined in figure 30.



 CEUSS v. 1.0	CEUSS v. 2.0 
<ul style="list-style-type: none"><input type="checkbox"/> No Authentication<input type="checkbox"/> "Fake" CEDAS Data<input type="checkbox"/> Limited ADSBx Data<input type="checkbox"/> Stand Alone Application	<ul style="list-style-type: none"><input type="checkbox"/> Technician and Pilot credentials<input type="checkbox"/> "Real" CEDAS data<input type="checkbox"/> All ADSBx Data<input type="checkbox"/> Integration with production environment

Figure 30: Version 1.0 and 2.0 Comparisons

CEUSS version 1.0 currently has no authentication when logging into the technician or pilot pages. Version 2.0 would include some type of credential authentication to create a more secure application. Currently, CEUSS version 1.0 uses only a day's worth of data from ADSBx and uses "fake" data we created to correlate and demonstrate CEUSS's functionality. As previously mentioned, version 2.0 would likely integrate real CEDAS data and be able to access all ADSBx data in real-time so that correlations would more closely reflect the client's typical use for the program. Lastly, version 2.0 would be focused on integrating with the client's current production environment, in the form of a widget or a popup on their current system, versus a stand-alone application as it currently is.

11. 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, CEUSS, helps predict when and where an upload should occur and helps predict why an upload failed. Some of the main features of CEUSS include a flight tracking map, an upload diagnostic test, and an upload location simulation. Not only will CEUSS help Honeywell's maintenance process, it will also help ensure safe flights for years to come and prevent aircraft accidents involving engine failures. All in all, this capstone project was a success as our team, EnginAir, worked productively together throughout the year to learn real-world project experience in a simulated environment. We look forward to enriching our post-graduation careers using the techniques and skills learning during this capstone year.

Appendix A: Development Environment and Toolchain

The following sections outline the development hardware environments, product toolchains, product setup, and production cycle.

Hardware

There are no specific hardware requirements for the development of CEUSS but table 4 lists our computer hardware specifications we used. This table includes the operating system, processor, memory, and number of cores.

Development Environment			
Operating System	Processor	Memory	Cores
Arch Linux	i9	32GB	12
MacOS	i7	16GB	8
Ubuntu Linux	i7	32GB	8
Windows	i7	64GB	12

Table 4: Development Environments

Toolchain

This section outlines tools used in the development of CEUSS starting with figure 5 outlining the development IDEs and additional plug-ins.

Development IDEs	Purpose	Plug-Ins	Purpose
Intellij	Environment used for the Java Backend Development using the Maven Package Management architecture	Lombok	A library designed to make the Java programming language easier; Makes code concise and reduces a chance for a bug
WebStorm	Environment that specializes in web development and specifically used for its compatibility with JavaScript	N/A	N/A

Table 5: Development IDEs

Development setup for this system is straight forward, we have two repositories, the front end and the back end. The backend repository is a Java project using the Maven Package Management architecture. To start developing a Java program, open the project in a Java IDE of your choice and import the Maven project. We decided to use IntelliJ because of the team's familiarity with the software and the availability of the plug-in Lombok which helped automatically generate getter and setter functions. Instructions of this being executed in the IntelliJ IDE can be found here:

<https://www.lagomframework.com/documentation/1.6.x/java/IntelliJ Maven.html>. The frontend repository is composed of JavaScript and HTML code. Like the backend project, the choice of IDE is arbitrary. We decided to use WebStorm due to the team's familiarity and its specified capability with JavaScript.

Table 6 outlines the supporting packages and their descriptions used in the development of CEUSS.

Supporting Packages	Description
PM2	ExpressJS multiprocessing manager module that allows multiple concurrent web requests
see-es-vee	A high efficiency CSV parsing library for Java 8 and above

Table 6: Supporting Packages and their Description

Because Node.js, the technology used for our frontend web app, by default only runs on one thread, this causes scalability issues with multiple clients. **PM2** was used to allow service to and be able to manage requests from multiple instances of the web app.

The package **see-es-vee** was used to help parse airport information from the airport parser. This package helped reduce the lines of code needed to complete this task and was ultimately used for parsing convenience.

Setup

Since we use Docker as a virtualization layer on top of the host operating system, theoretically any operating system that Docker supports can be used. However, for the purposes of keeping this document clear and succinct, we will be using Ubuntu 18.04 as our example on setting up and installing Docker. Instructions for other OSes are available here:

<https://docs.docker.com/get-docker/>.

Prerequisites

The important prerequisites for this system are as follows:

- Docker
- *docker-compose*

- Python (installed automatically through apt/yum, needs manual installation on Windows)

These systems help create and maintain a standard environment that is not susceptible to configuration changes on the host system nor host system updates.

We also recommend running the CEUSS system on a single machine, with **at least** 8 cores and 16GB of RAM. The server-side-app processes approximately 50GB of raw data every time it correlates which can result in high memory and CPU usage.

To install these on an Ubuntu 18.04 system, type the following as the root user:

```
apt install docker.io docker-compose
```

Please note that a reboot may be required in order to ensure that Docker's system service is loaded properly.

Initializing the Environment

In order to prepare the database and web application environments, we will use Docker Compose to set up a private networking bridge to securely allow communication between MongoDB, the server-side-app, and the web-app. It will also configure the web application and MongoDB when the system starts.

To start the "service" on Ubuntu 18.04, type the following as root or a non-privileged user in the docker group:

```
git clone https://github.com/enginair/server-side-app
cd server-side-app
docker-compose up -d
```

Docker will now automatically restart the MongoDB database and the web application. It also creates a virtual network adapter to allow the apps to securely communicate with each other without interrupting the host system's network operations. The web application becomes available at <host system IP>:3000. When this system is first started, the database will be empty and uninitialized. We will use the server-side-app to initialize this service.

To do this, perform the following steps as the root user or in a non-privileged account in the docker group:

```
cd <folder containing tails.json>

docker run --network="root_default" -e START_RAM=1G -e MAX_RAM=10G -e
MONGO_HOST=mongo -v $(pwd):/data duper51/enginair-server-side-app
```

```
./bootstrap.sh -d honeywellDatabase -A  
  
docker run --network="root_default" -e START_RAM=1G -e MAX_RAM=10G -e  
MONGO_HOST=mongo -v $(pwd):/mega duper51/enginair-server-side-app  
./bootstrap.sh -d honeywellDatabase -t tails.json
```

The web-app and MongoDB instance are now prepared and ready for correlation.

Production Cycle

The following steps walk through the process of editing the source code, compiling, and deploying CEUSS. This process is like committing a change to a repository although it is run in a docker image and would require a rerun of the image to automatically compile and build the new version.

Step 1: Login to docker image

```
docker login <docker hub username>
```

Step 2: Make edits to source code

Step 3: Verify changes are correct in IDE

Step 4: Commit changes to GitHub repository

Documentation on using Git and GitHub can be found at <https://dont-be-afraid-to-commit.readthedocs.io/en/latest/git/commandlinegit.html>

Step 5: Build and run the docker image as described in the “Initializing the Environment” portion of this document