Software Design Document
Version 2.0
February 14, 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. Implementation Overview 5
3. Architectural Overview 6
3.1. Data Importation 7
3.2. Correlation 8
3.3. Web Page Rendering 10
4. Module and Interface Description 11
4.1. Backend Module 11
4.2. Front End Module 18
4.2.1. Technician 18
4.2.2. Aircraft Pilot 21
5. Implementation Plan 23
6. Conclusion 24
7. References 25

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 and helped spark the advancement of the Aerospace and Defense industry. The Aerospace and Defense industry is comprised of the manufacturing, sale, service of aircraft, aerospace parts, space vehicles, and military defense systems.

Our project focuses on the manufacturing of engines, specifically turbofans. Turbofans are a type of jet engine also called a gas turbine which is primarily used for aircraft propulsion. These jet engines produce thrust through the burning of fuel which gets released as hot gas. This gas is forced through the turbine blades causing them to rotate. These turbines, however, require a sufficient amount of airflow between the blades before introducing fuel and starting combustion. If the turbine blades are not pushing enough airflow before this happens, a hot start occurs which causes the engine to overheat and results in damage.

Auxiliary Power Units (APUs) are smaller turbine engines that generate high-pressure exhaust which is used to kickstart the turbine blades to prevent a hot start. Our sponsor, Honeywell, is the largest producer of gas turbine APUs, with more than 100,000 produced and over 36,000 still in use today. Honeywell APUs are found on common commercial aircrafts such as the Boeing 747 and Boeing 777.

During a flight, the Engine Control Unit (ECU) saves trending and maintenance data which is reported to the mechanics and engineers via a hardware diagnostic report. Proper upkeep is important to prevent maintenance delays and keeps the engine functioning properly to ensure a safe flight. Currently, Honeywell engine operators are required to download and send engine diagnostic data reports once a month. This process requires:

1. a manual port connection using a USB device and a cable,
2. transferring a 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 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 allows engines to autonomously upload engine data wirelessly to a cloud. The CEDAS is hosted on an embedded computer located in the aircraft along with a WiFi antenna. 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

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

determine the status of the aircraft; whether it is grounded, inflight, or if there is a potential problem with the engine.

To solve our client's problem, our team has come up with the Connected Engine Upload Status System (CEUSS). Two primary goals of this system are: [1] provide airplane operators a way to know where to park their aircraft for the highest chance of upload success and [2] provide engineers a way to be notified and help visualize where and when potential problems occur. We describe our requirements in the form of user stories to 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 the 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.

Our requirements are further explained through the distinction between system and user requirements. System requirements are broken into three main components: Database, Backend Server, and GUI. The Database requirements are focused primarily on the creation and data population of three databases: Upload, Landing, and Access. The Backend Server requirements focus on the data correlation and identification mechanisms, and the GUI requirements focus on the speed and latency specifications of the system. The User requirements detail the system's capabilities from the ser's perspective.

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. Implementation Overview

The Connected Engine Upload Status System (CEUSS) is composed of two main components: the server-side backend software, primarily responsible for the data import and execution of database correlations, and the administrative front end web panel, primarily used as a graphical interface to display notifications and LON/LAT locations. CEUSS can be broken down into five high-level technical components including a Flight Database, Database Management, Command Line Application, Web Application (Front/Back End), and Graphical Illustration.

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.

3. Architectural 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 document, the server backend and correlation execution are considered separate. This component breakdown is outlined in Figure 1 shown below.

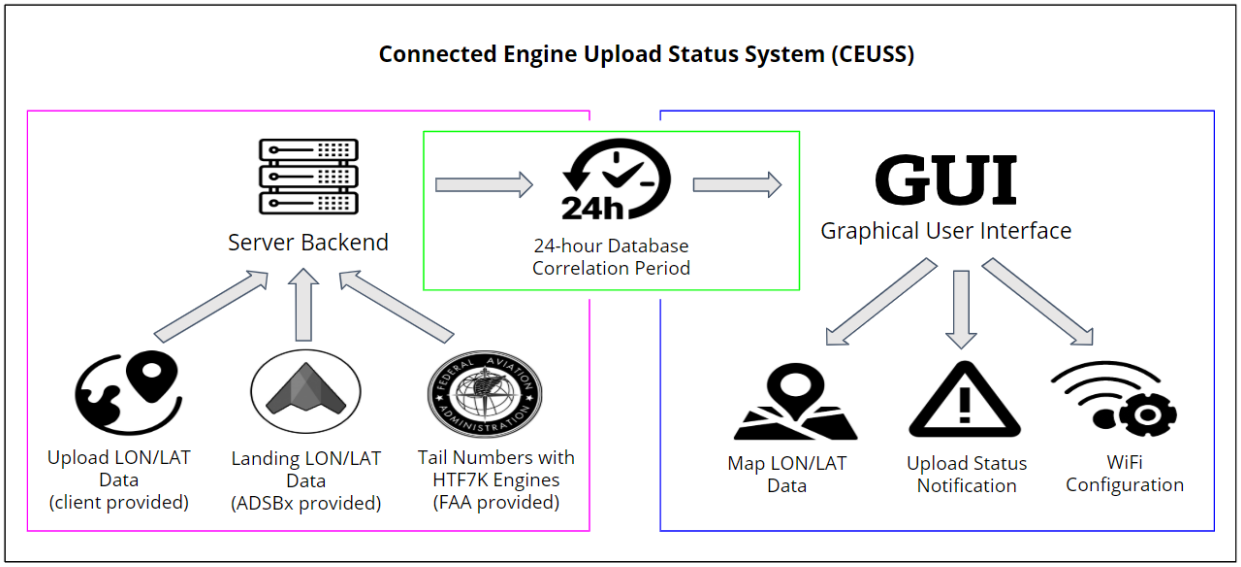

Figure 1: Overall Architecture of the Connected Engine Upload Status System

The black outline shows the overall system composed of the three parts, 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. This architecture is centered around object-oriented programming (OOP) concepts and incorporates several design patterns. Our system exploits the modularity of OOP through classes and objects and our project uses common open source practices.

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

3.1. Data Importation

CEUSS processes data from three sources to be used in the correlation step. Upload LON/LAT data will be 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. A more detailed description of the data importation process is described below in Figure 2.

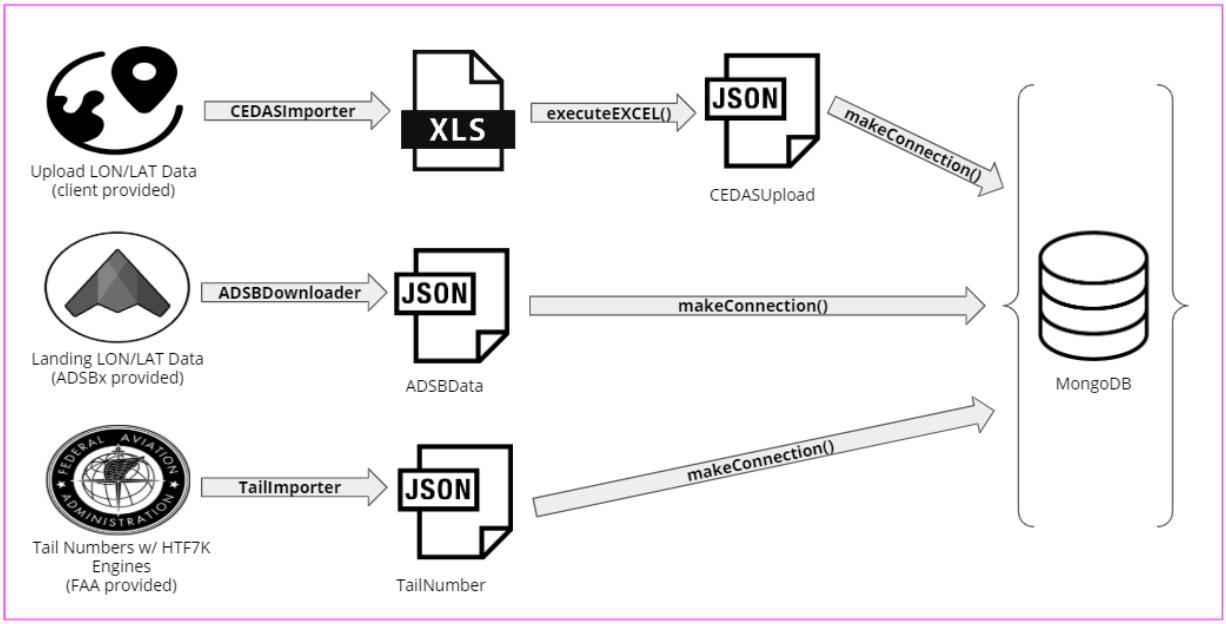

Figure 2: Data Flow of Data Importation Component of CUESS

The upload LON/LAT data, which contains longitude and latitude coordinates of previously recorded HTF7K engine diagnostic uploads, will be provided by the client in an Excel document format. The CEDASImporter will download this data as an XLS file and will be converted to a JSON file through executeEXECL() as CEDASUpload. The landing LON/LAT data from ADSBx includes longitude and latitude coordinates of landing sites of all planes that have been recorded by ADSBx's vast network of public receivers. This data is downloaded via ADSBDownloader as a JSON file type ADSBData. The tail numbers of aircrafts containing an HTF7K engines is downloaded as a JSON file via TailImporter. This file is type TailNumber. All three data JSON files must be added to the Mongo database for data correlation. This occurs via the Morphia MongoDB ORM library that we use. These connections are made from the ImportExecutor::makeConnection() method.

3.2. Correlation

The correlation step is the core of our system's functionality. This step correlates the ADSBData and CEDASUpload in the database to determine flight upload statistics like identifying when and where missing uploads occur and categorizing the status of each upload. This correlation via the Correlator results in a JSON file containing all the correlation results and statistics that will be rendered to the user. The correlation data flow is shown below in Figure 3.

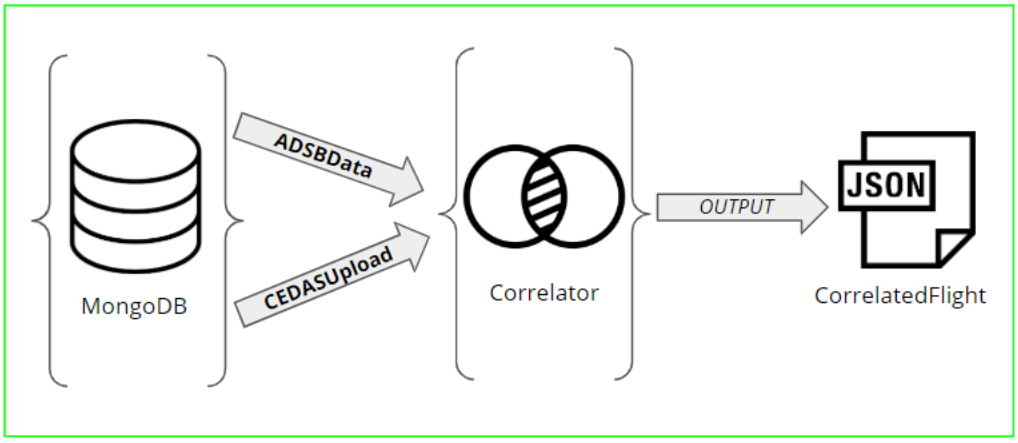

Figure 3: Data Flow of Correlation Component of CUESS

Figure 4 displays a screenshot of the CorrelatedFlight JSON file produced from the correlation step. 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 similar to the GeoJSON specification of these fields.


```
className: "edu.nau.enginair.models.CorrellatedFlight"
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 4: 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).

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

3.3. Web Page Rendering

The webpage is responsible for rendering the correlation statistics visually for the user to understand. The CorrelatedFlight JSON file produced by the Correlator contains LON/LAT coordinates of all correlated flights and its corresponding WiFi configuration. This will be plotted on a map using OpenStreetMap. The CorrelatedFlight JSON file also contains 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 to be searched via the GUI.

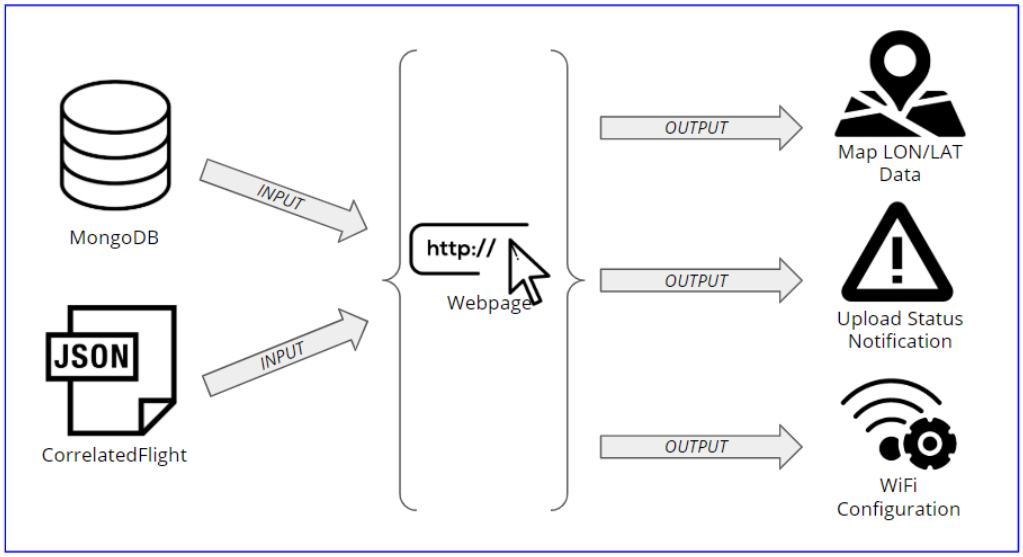

Figure 5: Data Flow of Web Page Rendering Component of CUESS

4. Module and Interface Description

The system contains two modules: the backend server module and the front-end GUI module. The backend module can be expressed using a UML diagram since the architecture is influenced by OOP principles such as classes and objects. The front-end GUI is illustrated using a type of page breakdown diagram which include subpages and page options.

4.1. Backend Module

This section describes the backend server module and explains each component in depth specifying each method and field attribute. Figure 6, shown on the next page, illustrates the UML structure of the backend and is followed by an explanation of the backend execution flow overview.

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

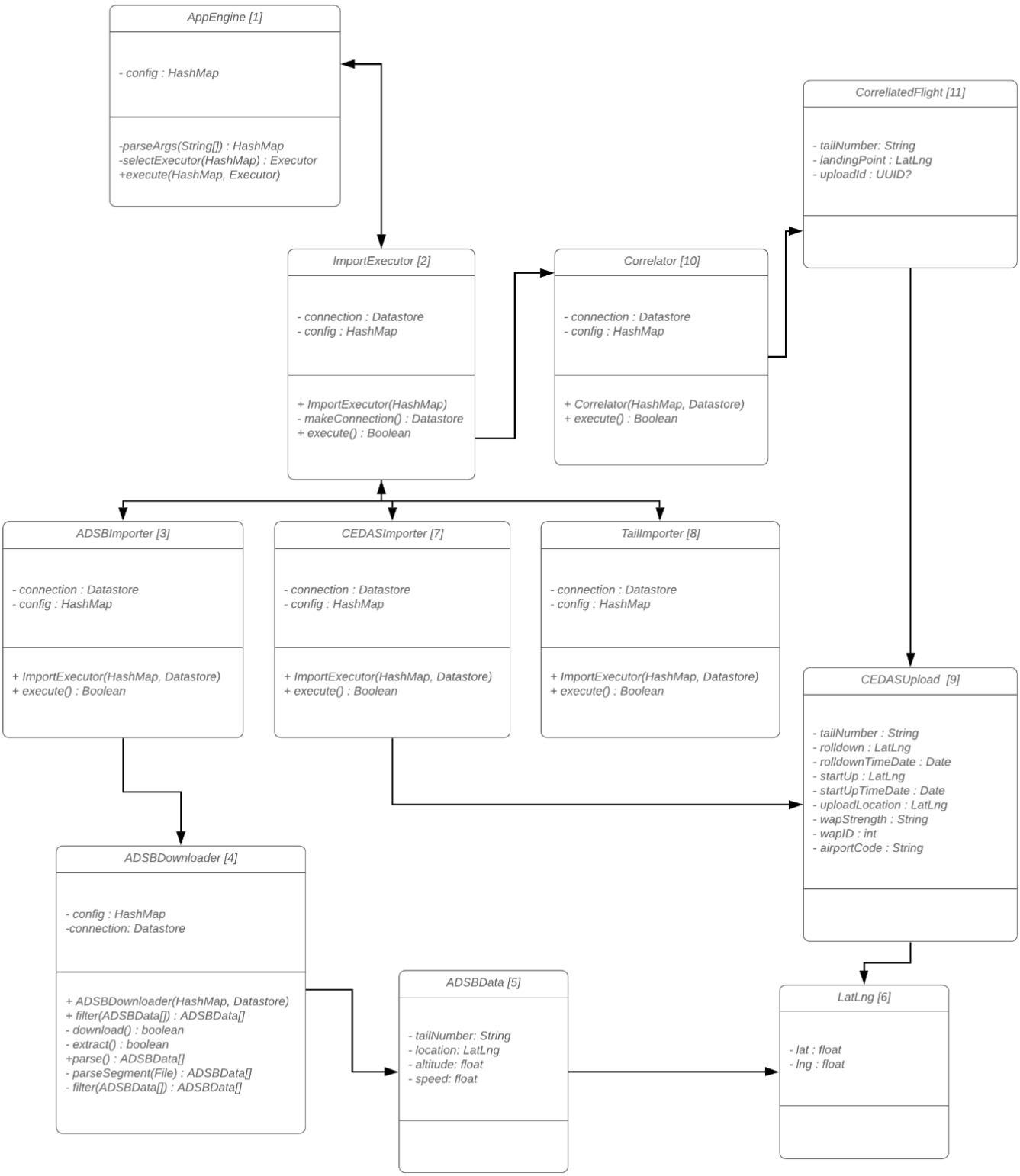

Figure 6: UML Diagram of Backend Architecture

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 to 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 below.

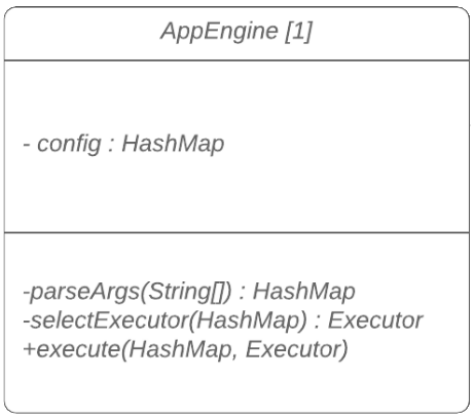

Figure 7: UML of AppEngine Component

App Engine

AppEngine is the start of the program execution and will execute a path according to the command-line arguments passed in. Figure 7 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 instances execute() function which begins the database import based on the given run-configuration.

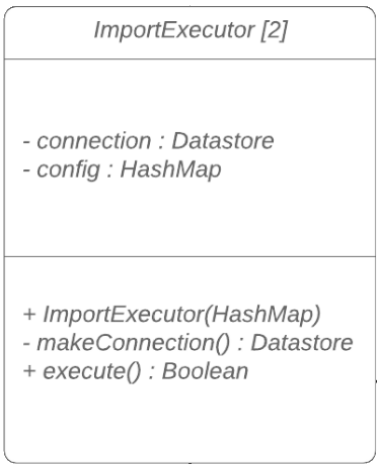

Figure 8: UML of Import Executor Component

ImportExecutor

The ImportExecutor is the template for ADSBImporter, CEDASImporter, and TailImporter. Figure 8 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. 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.

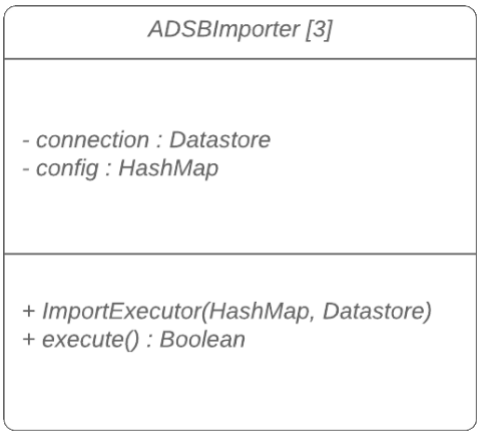

Figure 9: UML of ADSBImporter Component

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. Figure 9 organizes the methods and data fields associated with ADSBImporter, which is mirrored from ImportExecutor.

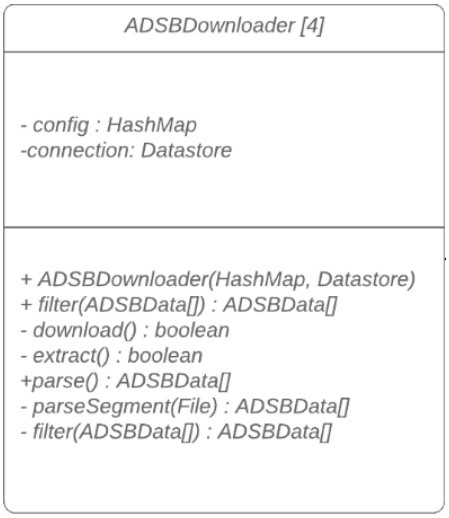

Figure 10: UML of ADSBDownloader Component

ADSBDownloader

ADSBDownloader is the helper class invoked from ADSBImporter which is outlined in Figure 10. 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.

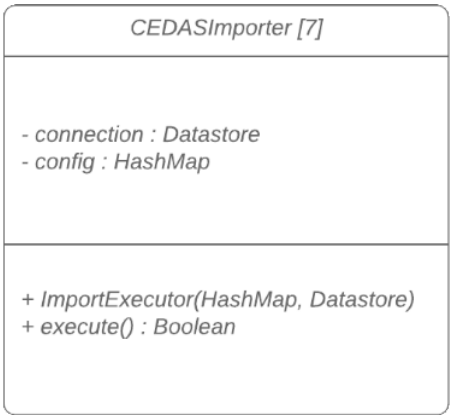

Figure 11: UML of CEDASImporter Component

CEDASImporter

The CEDASImporter is an object created from the Importer interface that imports data from client provided Excel files. Like ADSBImporter, Figure 11 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.

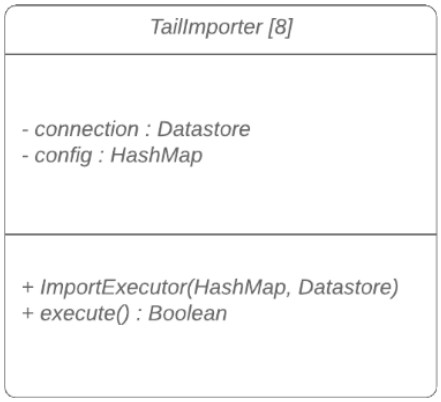

Figure 12: UML of TailImporter Component

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. Figure 12 displays TailImporter as a UML component similar to CEDASImporter and ADSBImporter.

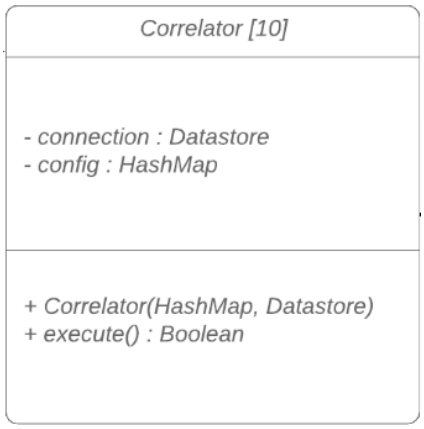

Figure 13: UML of Correlator Component

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 13 outlines the fields and methods of Correlator as a UML component.

4.2. Front End Module

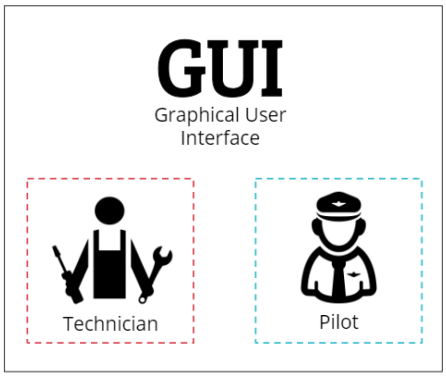

Figure 14: Overview of GUI User Modes

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 as outlined in Figure 14. In order to differentiate the two user modes, the webpage will have a toggle button to switch between the users and will display the appropriate information. 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 breakdown of each web page mode is described below.

4.2.1. Technician

The engine technician front end web page architecture consists of four main pages not including Home: Flights, Flight Upload Simulator, Database, and Mapping. These four pages provide the GUI functionality for all technician user stories. The web page outline is illustrated in Figure 15 and each page is explained is 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].
- simulate various locations and their corresponding WiFi configuration [2].
- know the status of each landing/upload entry [3].
- run a report to determine the cause of a failed upload [4].
- visualize the status of each upload entry [5].
- view all aircraft landing locations, every 24 hours [6].

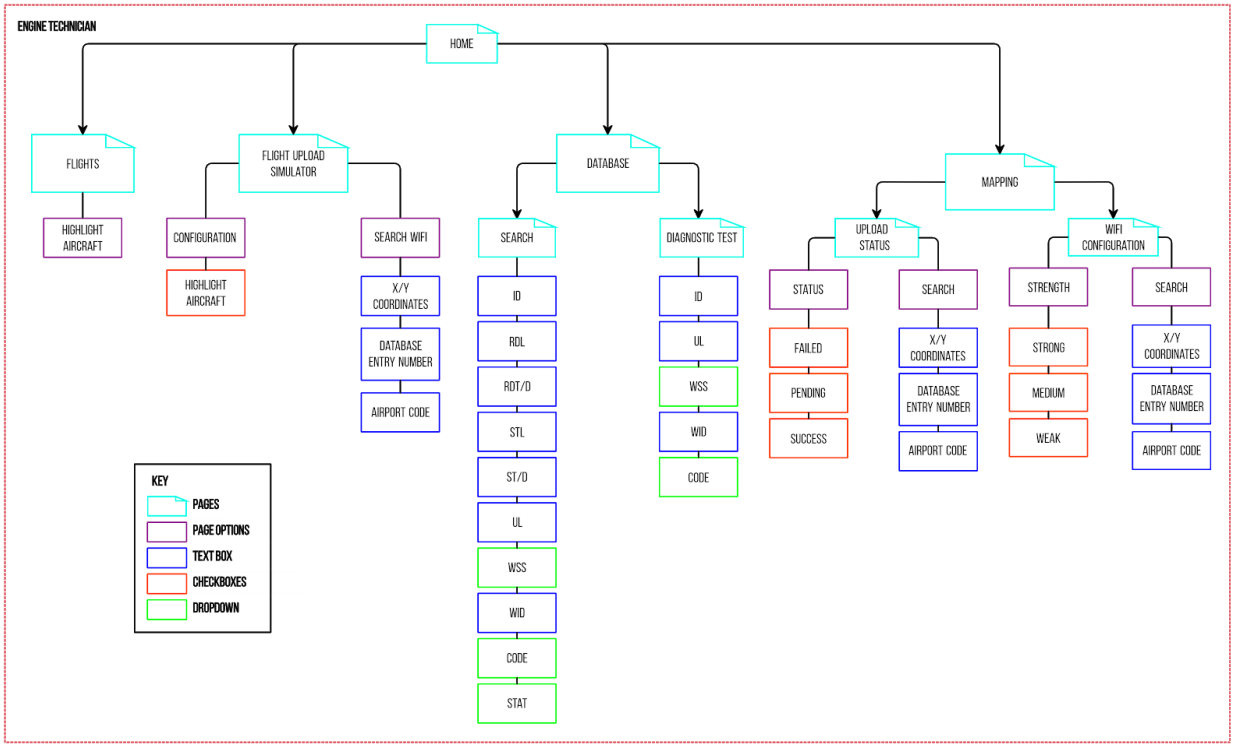

Figure 15: Overview of Web Diagram of the Technician User Mode

Flights

The Fights page is responsible for mapping current flights in progress [1]. This data comes from ADSBx flight feeders collecting which will be integrating with the OpenStreetMap API. This map is contained for flights within the U.S. and each flight on the map will be indicated using an icon. Each individual flight on the map can be highlighted via a checkbox on the page which will display flight information like tail number, engine number, and if available, longitude and latitude coordinates as well as altitude. This map is updatable every 24 hours as our data collection only occurs every 24 hours.

Flight Upload Simulator

The Flight Upload Simulator is responsible for displaying WiFi configurations at a location(s) to simulate and predict upload success [2]. 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 has the option to indicate a specific X/Y coordinate to pinpoint a potential upload location and the system will correlate those coordinates to find the nearest WiFi configuration. The system will then display the result of the potential upload as “Success”, “Pending”, or “Failed” given the location and the WiFi strength. The engine technician also has the option of searching a specific upload entry from the database using the entry number. This will display an icon indicating the aircraft location at the time of upload and will also

display the heat map of the nearest WiFi configuration to visualize the upload result in relation to location. Lastly, the technician can easily view an airport's WiFi configurations and simulate landing locations by searching the three-letter airport code. For increased icon visibility when navigating the map, the page has a configuration checkbox to highlight the aircraft.

Database

The Database page has a subpage Search which is primarily used to visualize each database upload entry [3]. Using the search 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
ST/D: Engine STart up Time and Date
UL: Upload GPS Location
WSS: WAP Signal Strength (1-5)
WID: WAP ID number
CODE: Airport code
STAT: Status of upload entry (Success, Pending, Failed)

The Database page also contains the Diagnostic Test page which displays all Failed upload status entries and allows the technician to run a test to determine the cause of failure [4]. This page also has filtering features which allows the technician to single out types of failures or failures at a location. The diagnostic test page has filters ID, UL, WSS, WID, and CODE. After running the diagnostic test, the system will display a results page containing a test ID and a summary of findings according to predefined conditions listed in the Requirements Document. The test results will also contain a time and date stamp of execution.

Mapping

The Mapping page contains an Upload Status subpage which displays upload status entries as colored dots on a map correlating to the status of the upload at a location [5]. 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 "Status" or can filter by specific LON/LAT coordinates. Other filters include airport code 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 database entry number, X and Y coordinates, airport code, upload status, and WAP ID number [5]. This mapping feature inherently indicates landing locations since an upload entry occurs on landing and corresponds to its location [6].

The Mapping page also contains a WiFi Configuration subpage which displays heatmaps correlating to each WiFi reference point. The user can select the WiFi strength of interest using

the checkboxes which will render a map indicating WiFi locations with this type of strength. The user can narrow the search by inputting the three-letter airport code to view all WiFi configurations at that airport or can enter an X/Y coordinate location to view a specific configuration.

4.2.2. 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 user story and is illustrated in the web page outline shown in Figure 16. The user stories are listed below and numbered for reference.

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

- simulate locations and their WiFi strength [7].

visualize locations on where to park the aircraft for an upload success [8].
**Note:* these user stories are extremely similar but are different approaches to the same result.

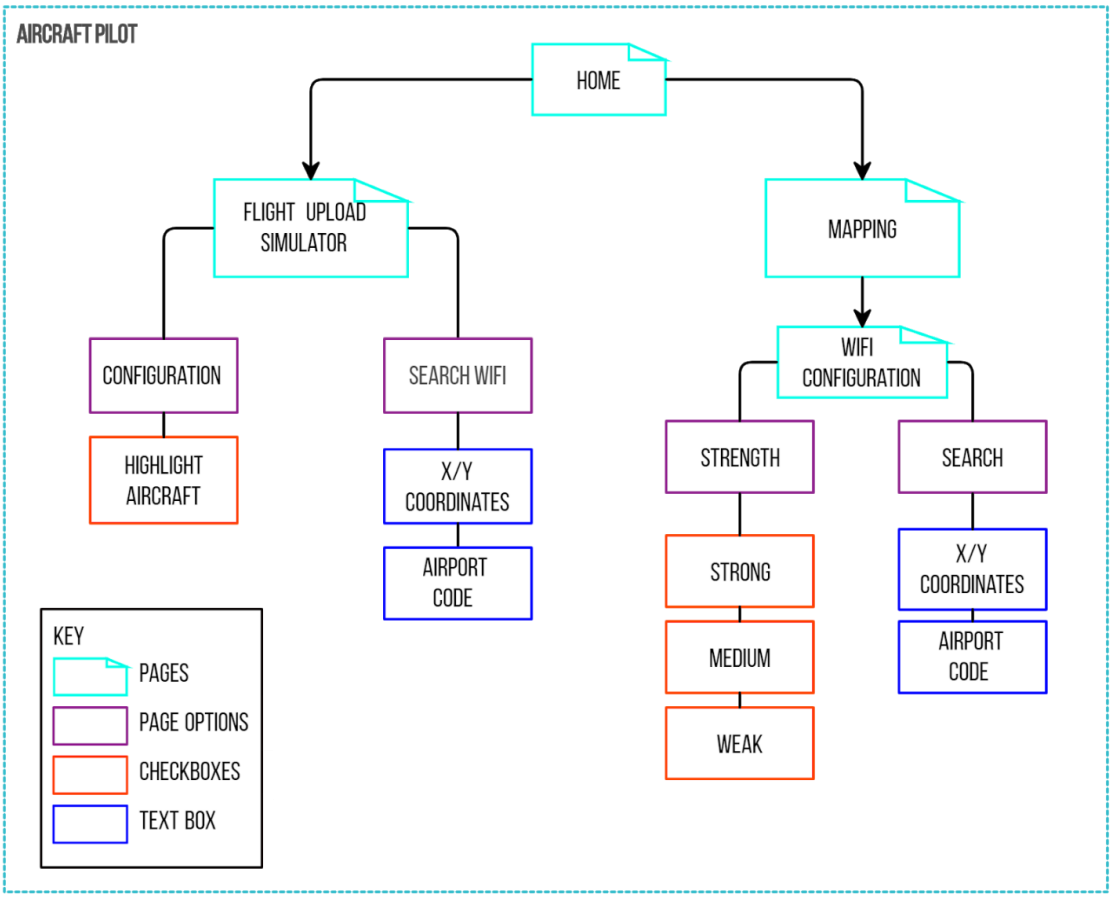

Figure 16: Overview of Web Diagram of the Pilot User Mode

Flight Upload Simulator

The Flight Upload Simulator is responsible for allowing the user to simulate parking locations to predict upload status based on WiFi connectivity in the area [7]. Using the Search WiFi options, the user can input X/Y coordinates to pinpoint a landing location and which the system will correlate the nearest WiFi heat map. Based on the heatmap and the potential landing location of the aircraft, the system will determine whether the upload will be successful. Users can then determine what longitude and latitude coordinates to park the aircraft for a successful engine upload. The users can simulate landing locations based on airport code rather than X/Y coordinates. This will render the map to show WiFi heat maps in the airport to help pilots determine the best upload location. The map can also be configured to highlight the "test aircraft" indicating the designated test location.

Mapping

The Mapping subpage provides similar features as the Simulator subpage but does not provide the upload status prediction. The mapping subpage is specifically designed for visualization of WiFi configurations [8]. Users can use this subpage but will have to determine the success of an upload on their own. Users can view WiFi strength heatmaps using the checkbox strength option that displays configurations across the US. The user can narrow the search using an airport code or specific X/Y coordinates.

5. Implementation Plan

Figure 17, illustrated below, outlines the entire spring 2020 semester schedule which includes individual project development as well as capstone course documentation and presentations. This schedule is broken up into weekly intervals starting on Sundays during our internal team meetings. The red font in the weekly headers indicate the current week and the coloration of the boxes indicate levels of completion. Green indicates "completed", yellow indicates "in-progress", and red indicates "not complete". Each task is also complimented with a "Person(s) Responsible" column. EnginAir refers to the entire team and is relevant for team assessments like Design Reviews, Prototype Demos, UGRADS Presentation and supplemental items and reports relating to the entirety of the project.

EnginAir Schedule Spring 2020

Task	Person(s) Responsible	Jan 19 - 25	Jan 26 - Feb 1	Feb 2-8	Feb 9-15	Feb 16-22	Feb 23-29	March 1-7	March 8-14	March 15-21	March 22-28	Mar 29 - Apr 4	April 5-11	April 12-18	April 19-25	Apr 26- May 2	May 3-9
Backend Architecture UML	Ian, Gennaro																
Backend Importer Development	Gennaro, Dylan																
Backend Correlator Development	Ian, Chloe																
Software Design Document Draft/Final	Megan			Feb 7	Feb 14												
Front End Development	Chloe, Ian, Gennaro, Dylan																
Front/Backend Integration	Chloe, Ian, Gennaro, Dylan																
Design Review II	EnginAir						Feb 28										
Full Prototype Tech Demo	EnginAir							buffer	Mar 9								
Testing/Refinement	Chloe, Gennaro, Ian, Dylan																
Software Testing Plan	Megan											Apr 3					
Design Review III	EnginAir											Apr 3					
Team Website	EnginAir														Apr 24		
Capstone Poster	EnginAir														Apr 24		
UGRADS Conference	EnginAir														Apr 24		
Final Acceptance Demo	EnginAir														buffer	Apr 27	
Team Reflection Document	EnginAir																May 7
User Manual	EnginAir																May 7
Final Checkoff Sheet	EnginAir															Apr 30	May 7
Final Project Report	EnginAir																May 7

Figure 17: Schedule of Spring 2020 Semester

A big milestone for our project is the full prototype demo which will display the functionality of the entire project. Currently, we are working on the backend component and will work on the front-end web page and backend integration during that time. We project our full prototype to be complete a week before the demo is due and we project our completed product, after testing and refinement, to be done a week before the acceptance demo. We anticipate problems to occur and so having this time buffer allows us to be more flexible on programming deadlines if needed.

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

6. Conclusion

In conclusion, the US Aerospace industry, composed of manufacturing, sale, service of aircraft, aerospace parts, space vehicles, and military defense systems, is highly profitable and supplies millions of jobs. Our sponsor, Honeywell is the largest producer of gas turbine APUs and ranks 13 out of the top 50 aerospace companies. 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 helps predict when and where an upload should occur and helps predict why an upload failed. This document outlines the major software design decisions implemented for this project. It includes a review of the key requirements which we have detailed through user stories. It describes the technologies we have decided to use and illustrates the overall architecture of the backend server and the front-end web page as well. The document details an implementation plan which details milestones and due dates of programming/project specific deadlines and course required presentations/documents. We are team EnginAir and we are confident that our project plan is on track to provide our client a software product that helps better the maintenance process to keep engines working properly.

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

7. References

1. Arbor, Ann. “Global Aerospace Industry Worth $838 Billion, According to AeroDynamic Advisory and Teal Group Corporation.” Home - Teal Group. Teal Group Corporation, July 12, 2018. https://www.tealgroup.com/index.php/pages/press-releases/53-global-aerospace-industry-worth-838-billion-according-to-aerodynamic-advisory-and-teal-group-corporation.
2. Quora. “How Do Jet Airplanes Start Their Engines?” Forbes. Forbes Magazine, April 14, 2017. https://www.forbes.com/sites/quora/2017/04/14/how-do-jet-airplanes-start-their-engines/#11309fa9ab4b.
3. “Top Aerospace Companies: Top 50 Lists.” AviationOutlook. AviationOutlook.com, October 10, 2019. https://aviationoutlook.com/top-aerospace-companies/#kcmenu.
4. “2019 STATE OF THE AMERICAN AEROSPACE AND DEFENSE INDUSTRY.” AIA Aerospace Industries Association. www.AIA-AEROSPACE.rg . Accessed November 10, 2019. https://www.aia-aerospace.org/wp-content/uploads/2019/06/AIA-2019-Facts-and-Figures.pdf.

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