



Technological Feasibility Analysis
November 4, 2019
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. Technological Challenges.....	4
3. Technology Analysis.....	5
3.1 Flight Database.....	6
3.1.1 Alternatives	6
3.1.2 Chosen Public Flight Database: ADSB Exchange.....	7
3.1.3 Proving Feasibility	7
3.2 Command Line (Server)	8
3.2.1 Alternatives	8
3.2.2 Chosen Command Line Application Language: Java	8
3.2.3 Proving feasibility.....	9
3.3 Database Management.....	9
3.3.1 Alternatives	9
3.3.2 Chosen Database: MongoDB	10
3.3.3 Proving feasibility:.....	10
3.4 Web App (Front End)	10
3.4.1 Alternatives	11
3.4.2 Chosen Front End Web App Application: NGINX.....	11
3.4.3 Proving Feasibility	11
3.5 Web App (Back End).....	12
3.5.1 Alternatives	12
3.5.2 Chosen Back End Web App Application: Node.js	12
3.5.3 Proving feasibility	13
3.6 Graphical Illustration	13
3.6.1 Alternatives	14
3.6.2 Chosen Graphical Illustration Application: Google Maps API.....	15
3.6.3 Proving feasibility	15
4. Technology Integration	16
5. Conclusion.....	19

1. Introduction

In 1903, the Wright Brothers made history building and flying the first successful powered airplane. Since then, airplanes have become the most commonly used mode of transportation in the US. Aircraft engines require periodic maintenance and are 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.

Our sponsor, Honeywell, is the largest producer of gas turbine auxiliary power units (APUs), with more than 100,000 produced and over 36,000 still in use today. 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 allows for a collection of a small data set of basic maintenance information. Honeywell has developed a connected engine product called Connected Engine Data Access System (CEDAS) which 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. The problem is that, if the WiFi connection is spotty or nonexistent at a certain location, the diagnostics may not send. When this happens, 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.

Our team, EnginAir, has been tasked with creating three essential applications: Landing Status Monitor (LSM); Wi-Fi Configuration Manager; and Customer Connection Information module. This project aims to make the diagnostic report process more efficient by ensuring WiFi access points for operators to upload their reports to a cloud database. EnginAir is a Computer Science senior capstone group formed under the School of Informatics, Computing and Cyber Security (SICCS) at Northern Arizona University (NAU). Team members include Chloe Bates, Megan Mikami (Team Lead), Gennaro Napolitano, Ian Otto, and Dylan Schreiner. The team mentor is Scooter Nowak. EnginAir works closely with Harlan Mitchell who is a Systems Technical Manager for the HTF7K Controls System Integration Unit at Honeywell.

The purpose of this technological feasibility analysis document is to provide a guideline for our project's design rationale. This analysis begins with an outline of our project's high-level requirements which are the major challenges we expect to face in the development of this project. This is followed by the Technological Analysis section which introduces the metrics that will be used to analyze potential technology solutions. Each subsequent section details the advantages and disadvantages of each alternative and explains our decision for choosing one over another. We then explore how each chosen technology integrates with each other and provides an overview of the flow and structure of our system. This document is concluded with a summary of each challenge, the chosen technology, and the confidence level in which we believe will solve the technical problems of our project.

2. Technological Challenges

Our project is comprised of three major components: [1] Landing Status Monitor, [2] WiFi Configuration Manager, [3] Customer Connection Module.

The Landing Status Monitor collects the date, time, and longitude and latitude (LON/LAT) coordinates identifying a landing event where an individual engine startup or shutdown occurs. This change in engine status is when a data diagnostic report is uploaded to the cloud. After 24 hours, the landing status monitor will analyze both databases and find discrepancies between actual uploads and the predicted uploads. If an inconsistency is identified, the system will notify the user about a possible reason (i.e. if an airplane checked in at 11:56 AM on June 14 at longitude 35.135504 and latitude -111.677332, and an upload was not found at 11:56 AM June 15, there could be a missing WiFi router near the landing point.)

The WiFi Configuration Manager then maps specific LON/LAT points on a map corresponding to a specific aircraft upload location. This information corresponds to the cloud upload database which is populated with diagnostic reports from the ECU. The ECU also provides WiFi configuration data which helps create an overall view of the airport WiFi configuration/connection.

The Customer Connection Module is a GUI that illustrates various metrics including WiFi strength and location of WiFi access points.

In order to complete the three components listed above, we have outlined five high-level technical requirements described below:

- We need to determine which public flight database to mine [1][2]
- We need to create a command-line server-based program to take in information from a flight database [1][2]
- We need to build a database to store flight information [1][2]
- We need to build a web application for the client [3]
- We need a way to graphically illustrate the WiFi connections/strength [3]

The numbers in brackets correspond to the task that the technical requirement most closely satisfies.

3. Technology Analysis

In this section, we breakdown the technical challenges further into five technical components and we describe the metrics that will be used to ensure that the project challenges are completed with the most feasible technology. The major components for this project include:

- Flight Database
- Command Line (Server)
- Database Management
- Web Application (Front End/Back End)
- Graphical Illustration

For each of these, we will give a brief explanation of what the challenge entails, how it plays a larger role in the project, and rationalize a chosen technology based on intensive research and analysis.

The technology will be selected and measured using the following metrics:

- Accessibility (i.e. How easy is the application or data access? Do you need an account/authentication?)
- Capability (i.e. How well does the application complete the tasks? Does it do everything we expect it to do?) will be ranked according to the following scale:
 - 1 = None, does not complete any required tasks
 - 3 = Some, completes some required tasks
 - 7 = Most, completes most required tasks
 - 10 = All, completes all required tasks
- Compatibility (i.e. What are the required applications/languages? How well does it work with other applications/languages?)
 - 1 = None, does not work with any of the other applications
 - 3 = Some, works with some of the other applications
 - 7 = Most, works with most of the other applications
 - 10 = All, works with all the other applications
- Cost (i.e. What are the costs associated with the application/technology?)
- Difficulty (i.e. How easy is the application/language to use? Does it require experience?) will be ranked according to the following scale:
 - 1 = novice, requires little/no knowledge/experience
 - 3 = intermediate, requires some basic knowledge
 - 5 = expert, requires extensive knowledge/experience

3.1 Flight Database

The flight database is the baseline for our data collection. It will be primarily used for the Landing Status Monitor and the WiFi Configuration Manager. This public flight database will need to download a data file containing flight tail numbers and an approximate landing time to estimate when an engine diagnostic report should be uploaded to the cloud.

3.1.1 Alternatives

Two alternatives that we are investigating is FlightAware and ADS-B Exchange:

FlightAware is a digital aviation company that operates the world's largest flight tracking and data platform. FlightAware receives its data from air traffic control centers, a network of Automatic Dependent Surveillance Broadcast (ADS-B) stations, Aireon global space-based ADS-B, and datalink (satellite/VHF) via major airline providers. FlightAware data reports are downloaded as TSV or CSV formats that can be easily opened with Microsoft Excel, Microsoft Access, or similar spreadsheet/database applications.

FlightAware has two APIs (FlightXML and FlightAwareFirehose) available to download and access their data, both of which require an account for user and FlightXML API key authentication. FlightXML is a query-based API used to obtain current/historical flight data. FlightXML is compatible with any application that supports SOAP/WSDL or REST/JSON and can be used with Ruby, Java, Perl, Tcl, ASP, and other languages. FlightAwareFirehose is a TCP-based, SSL streaming data feed that uses machine learning models and algorithms that aggregate 10,000 flights/second.

ADS-B Exchange (ADSBx) is a co-operation of different surveillance and navigations feeders from around the world and is the world's largest source of unfiltered flight data. ADS-Bx differs from other typical flight tracking sites because all data is from the community and made available to the community. ADSBx does not anonymize any flights and collects data on military and private aircrafts. ADS-B information is broadcasted unencrypted over the air and thus ADSBx does not expose any information that would not be available via other methods. ADSBx data reports are downloaded as a JSON file containing an ICAO identifier, the aircraft registration number, the aircraft longitude and latitude, and information if the flight is grounded.

ADSBx had two different types of data access: Live Position Data and Historical Data. Both data access options require a user to host a feeder that sends data to ADSBx. It also requires that the user to be a non-commercial personal or an enthusiast user. The live position data that is accessed through a REST API also requires an authentication key provided by ADSBx. The historical data that is categorized via data and time requires a donation of about 0.15 USD per GB downloaded to cover bandwidth and storage costs.

3.1.2 Chosen Public Flight Database: ADSB Exchange

We decided to choose ADS-B Exchange as our technology for the flight database. Table 1 below summarizes scores associated with the metrics used to determine which flight database to use. For the metric accessibility, we decided that ADSBx was more feasible because we could download sample data without having an account. Our client mentioned that he will provide data for us to work with but in the meantime, ADSBx has data from the 1st of each month available publicly for evaluation purposes. We will use this data as a training set on our system until we can verify it with our client's provided dataset. ADSBx data access also requires the user to be a host for a feeder, a device that listens to radio data, decodes it, and sends it to the ADSBx database. We have been in contact with an ADSBx representative to host a receiver at Northern Arizona University in trade for data access. For these reasons, Accessibility was given a score of 10 since this database contains an easy way to access information.

Because ADSBx's data download is a JSON file, it is easy to use a data format compatible with different applications, especially those that we choose to work with for our other technical challenges. Lastly, the capability of ADSBx is greater than the capability of FlightAware primarily because ADSBx does not anonymize any flights. This means that ADSBx data contains military and private aircraft. Because of this, ADSBx's dataset is more populated and thus we have a better overview of all Honeywell engine operators. For this reason, Compatibility was also given a score of 10 since the data file is simple to work with and the data is unfiltered and can provide a more accurate overview of the flights.


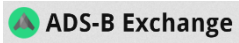
	Accessibility	Capability	Compatibility	Cost	Difficulty
	Account, API key	7	10	Dependent on flight data download	3
	Authentication key OR Monetary donation	10	10	About \$1.50-\$3/day	3

Table 1: Summarization of FlightAware and ADS-B Exchange flight databases

3.1.3 Proving Feasibility

The demo required to prove the feasibility of integrating the flight database execute the following:

- Finalize feeder hosting in trade for data access
- Download JSON file from ADSBx
- Ensure that the command line server application and database management can access and parse data

3.2 Command Line (Server)

In addition to the flight database, a server-side tool is needed for the Landing Status Monitor and the WiFi Configuration Manager. This tool will extract data from customer provided excel files. These files will contain the information exported from the cloud drive which contains the diagnostic reports uploaded from the ECU. Typical data fields include upload connection status, longitude, and latitude coordinates. Once our software reads the data, it will perform specific calculations to help determine and generate feedback to the user on whether an aircraft had a successful upload. This component will work alongside the flight database to correlate information about the aircraft uploads.

3.2.1 Alternatives

The two languages we are investigating for this technology are Java and Python.

Java is a general-purpose object-oriented programming language managed by Oracle. The Java development community is very large and includes an abundant amount of open source libraries and articles making it a very versatile programming language. Java is also most widely used for mobile application development, being the primary programming language for Google's Android OS and server applications.

Python is a general-purpose dynamically typed object-oriented programming language managed by the Python Software Foundation. Similar to Java, it has a large open-source user base with a wide variety of libraries. Python is widely used in the scientific space for subjects like Machine Learning, Artificial Intelligence, Data Science, and Bioinformatics. One of Python's most prominent users happens to be Instagram, which runs the world's largest deployment of Django.

3.2.2 Chosen Command Line Application Language: Java

There are a few reasons our group decided to use Java as the primary programming language for the server-side client. The first is our experience. Our academic experience at Northern Arizona University is heavily based on Java, making it the most familiar language in our option pool. The second is familiarity. We plan on using the Apache POI, which is an open-source library for handling excel formatted documents and is one of the most widely used libraries of its kind. Because Java has a more mature collection of libraries for this type of development, it can help us easily integrate some of the core functionality of the backend client. Due to our team's familiarity with Java and the openly available libraries, our team rated the difficulty a 6.



	Accessibility	Capability	Compatibility	Cost	Difficulty
	Downloadable	9	7	Free	6
	Downloadable	7	7	Free	6

Table 2: Summarization of Java and Python for the command line server

3.2.3 Proving feasibility

For future testing purposes, we would like to develop a demo implementing/exercising some of the core features expected from our server's client app. We expect our client to execute the following tasks:

- Read and write Excel and CSV data from the command line into the software to be used for calculations and manipulations using Apache's POI excel library
- Create HTTP requests to our API of choice, using Apache's http-client Java library
- Read and write data from our database management solution to effectively analyze previous and incoming information

3.3 Database Management

The database is an essential part of the Landing Status Monitor and the WiFi Configuration Manager as it will store all information received from the diagnostic reporters and be the primary comparative measure between the proposed uploads and actual upload datasets. The database will take in the data through the command-line application which will be stored to use on our web application. Because the database will need to interact with multiple programs with varying functionality, it must be fast, reliable, and secure in doing so.

3.3.1 Alternatives

Two different database options were selected to investigate: MongoDB and PostgreSQL.

MongoDB is a document database which stores data in a JSON format. It is a non-relational database meaning that it does not have tables and rows like many other databases. This also means that SQL commands cannot be used. Instead, MongoDB offers "drivers", which are APIs that can be imported into many popular languages, some of which include Java, C, Node.js, and Python. These drivers allow programs to interact with the database. Because MongoDB uses a JSON format, importing data from this file is simple using a method called *mongoimport*.

PostgreSQL is an open-source relational database. PostgreSQL is more widely known and has been around for many years. Unlike MongoDB, this database is object-oriented instead of

document oriented. While PostgreSQL still works with JSON files, it mainly uses static JSON files.

3.3.2 Chosen Database: MongoDB

The main reason that our team selected MongoDB as our database, was because of its compatibility with JSON files. With MongoDB, JSON files do not need to be static to be used; they can be readily changed when needed. Along with this benefit, MongoDB has “drivers” that can be used by different languages. Two of the compatible languages are those that we are planning to use for other portions of this project. The Java and Node.js sections will be able to easily add the driver to their code to add accessibility to the database. Because of compatibility with the data file and the use of drivers, MongoDB was given the compatibility rating of 10 and the reason why MongoDB was the selected database for our project.



	Accessibility	Capability	Compatibility	Cost	Difficulty
	Download	7	10	Free	7
	Download	9	8	Free	4

Table 3: Summarization of MongoDB and PostgreSQL for the database manager

3.3.3 Proving feasibility:

To test feasibility, a demo that will work with the Node.js and Java sections will be created. This demo is expected to have the following features:

- Creation of a MongoDB database
- Addition of drivers to the Node.js and Java applications to test connectivity with other areas of the project
- Testing how data insertion and deletion from the database

3.4 Web App (Front End)

Our front-end web application will support the Customer Connection Module. It will be responsible for relaying information to the user about the data uploads and will be loosely connected to the graphical illustration component. This front-end application will be serving primarily static files, Javascript and static HTML. For this webpage to contain the necessary information, it will work in cohesion with the server which will need to have high reliability and good response times.

3.4.1 Alternatives

We have selected a couple of options to conduct our analysis: Apache and NGINX.

Apache is the number one most used web server in production environments around the world, such as PayPal and Apple.com. Because of this, it is a very mature software and touts many features, such as URL rewriting and *mod_security* which is an input sanitizing security module. It is also increasingly stable, meaning that it's feature-set is well documented, and implementations of features are unlikely to change.

NGINX is a very popular caching and general-purpose webserver. It tends to serve static files significantly faster any because of its secondary purpose as a caching server, it also has a robust feature set implemented to support proxying to backend services, as well as caching of backend responses.

3.4.2 Chosen Front End Web App Application: NGINX

For this web app front end, we chose NGINX for its superior performance and ability to load balance our backend. It has an incredibly low time-to-first-byte and response overhead, in addition to the additional caching capabilities built-in to the server. Because we expect to receive lots of traffic for API calls and expect these calls to return large quantities of data, a load balancer group would allow us to split these requests amongst multiple servers. We can also use these load balancer groups to ensure stability during server updates. Because NGINX has load balancing and low overhead, our team rated it's capability a 10.



	Accessibility	Capability	Compatibility	Cost	Difficulty
	Package Install (yum, apt, etc)	7	10	Free	3
	Package Install (yum, apt, etc)	10	10	Free	4

Table 4: Summarization of Apache and NGINX for the web app front end

3.4.3 Proving Feasibility

To ensure that NGINX is the best solution, we will create a demo with the following tests:

- Test the various solutions against artificial load simulations
- Test each software's load balancing capability against a test backend while also maintaining prompt responses
- Test a reverse proxy using the same artificial load simulations

3.5 Web App (Back End)

In addition to our front-end application, the back end of the web app will also be supporting the Customer Connection Module. This application is concerned with the access of information through the database rather than the rendering of information via a web page. This causes a lot of stipulations in terms of performance and security. Our backend must be able to support frequent and repeated interactions with the database solution in order to be considered effective. It must also be able to set up easy interactions with the database to support the development process and must be scalable to support multiple concurrent requests.

3.5.1 Alternatives

We researched and analyzed the following technologies: Node.js, Django, and Flask.

Node.js is a commonly used variant of JavaScript designed to run in a server environment. The standard server framework for Node.js called Express allows for asynchronous catching of HTTP requests. A benefit of this is that multiple requests can be handled by one thread at the same time while maintaining a low time-to-first-byte (TTFB). TTFB is a measurement that is used as an indication of the responsiveness of a web server. Express will also work with many database frameworks due to its vast package repository and simple object design.

Flask is another Python-based web framework with significant efforts placed in ease of use. Its API is easy to get into, by simply marking functions as “routes” the server will automatically select the most appropriate route for an incoming request. This would help in our development tasks, as well as fixing potential bugs. One issue with Flask is its inability to perform multithreaded operations or handle multiple requests. To resolve this would take custom server setups and would add more complexity to our system. It also has no built-in database access approach, so we would have to use a tertiary library to accomplish this. This, however, may be trivial due to Python’s vast package repository.

Django is a web framework designed for Python environments. It has a vast feature set, such as conditional request routing, built-in load balancing, and model/view/controller support due to the project’s maturity. It also supports the common web development methodology, known as Model-View-Controller. This would support our need for an easy and reliable database access method.

3.5.2 Chosen Back End Web App Application: Node.js

Considering the data, we chose Node.js for a variety of reasons. For one, it will interact easily with our backend database, given that it natively supports JSON as its object notation. It also will be easy to make our system able to handle multiple requests on different threads, due to the availability of PM2, a common multithreading library. Node.js’s ExpressJS library also offers a simple and easy to use routing framework, which will result in reduced development

timelines overall. This is because all it takes to define a new subset of our API would be to create a new JavaScript file and define its routes inside of this file. For these reasons, we assigned Node.js a compatibility score of 8 and due to our relative unfamiliarity with the language, we assigned it a difficulty score of 6.




	Accessibility	Capability	Compatibility	Cost	Difficulty
	Package Install (yum, apt, etc)	10	8	Free	6
	<i>pip</i> install	9	4	Free	5
	<i>pip</i> install	9	8	Free	8

Table 5: Summarization of Node.js, Flask, and Django for the web app back end

3.5.3 Proving feasibility

To test the feasibility, we will develop a demo that executes the following tasks:

- Configure a test Node.js server with the ability to pull an arbitrary record from our backend database
- Obtain an average TTFB (time-to-first-byte) directly from the server to determine its unloaded performance
- Simulate high load requests and determine server responsiveness with 4 threads on a 4-core server, by ensuring that the TTFB remains under 500ms with 400 database requests per minute

3.6 Graphical Illustration

This graphical illustration component is an essential part of the Customer Connection Module. This component is designed to work in conjunction with the front-end web app to renders the LAT/LON coordinates from the upload database into a map representation. The goal of this module is to make it simpler for pilots to determine where they should park or stage their planes in order to get a WiFi signal that is adequate for a complete CEDAS data download. With this graphical illustration, we must be able to:

- Load LAT/LON coordinates from a database into a graphical representation
- Display ideal locations where to stage the aircraft within range of adequate WiFi
- Overlay various WiFi access points and their strengths on the graphic

3.6.1 Alternatives

We have researched four graphical APIs and are analyzed below: Google Maps API, Bing Maps API, Mapbox and OpenLayers.

Google Maps API is by far the most popular and widely used API today. With its extensive libraries and tools, this API delivers a fully interactive experience for the user. Google Maps JavaScript API capabilities provide an abundant amount of styling options, map interactions, and object layering. All of these are helpful in creating a custom and unique website, tailored to its functionalities. The familiar Google Maps navigation interface is another reason why this API is popular; many users have experience and feel comfortable using it. Accuracy is another strong point for the Google Maps API. With the out-of-the-box solution, you can count on constant, daily updates provided by Google Satellite and other sources. Machine learning algorithms are also used to increase speed and automate the mapping process. This maintains accurate, real-time location data.

This API is able to work alongside any platform that we chose. Google also provides an Android and iOS SDK to automatically handle access to Google Map servers, map rendering, and user interaction (such as clicks and drags) responses. To use data from a database, it can be imported into a map from MySQL and PHP or of the GeoJSON format.

Bing Maps API is another widely used mapping API provider. There are many tools and services that allow Bing's mapping API to ease the process of designing, developing, and implementing applications. The Interactive SDK, for example, provides a way to explore various Bing Maps features through usable and editable JavaScript code samples. This API, however, is not as fast or accurate as Google and cannot compete with Google's widespread data coverage. Bing uses the open platform HERE to provide location data which infrequently updates its location and map files.

MapBox is an open-source JavaScript library that is among the top Geographic Information System products. This API maps clean and concise data and is known for its stunning data visualization and map presentation. This API is known to have a steep learning curve and many users feel they do not utilize the API features to their full potential due to tricky coding standards. Because the MapBox system works best when used with large and complicated sets of data, it would not work well with the simple LAT/LON coordinates we would be feeding into the mapping system.

OpenLayers is an open-source JavaScript library that provides a solid foundation for mapping. An attractive feature of this API is its flexibility. Unlike Google, it is not tied to a certain map provider or technology. Also, with OpenLayers, you have the option to combine maps from multiple different sources which includes vector data (KML files, etc.), Google Maps background, etc.

3.6.2 Chosen Graphical Illustration Application: Google Maps API

The Google Maps API was chosen because it met all of the criteria that we were looking for in a graphical illustration. After creating an account, accessing the interface, and generating an API key, development using this API may begin. Both the extensive documentation provided by Google and the large user community aid in a quick learning curve and low difficulty in development, giving it a difficulty score of 3. Google Maps API provides a vast library that allows custom styling to provide a tailored experience for the user. The broad libraries can also be used to support the needs for diverse graphing features. This will play a huge role in displaying WiFi strengths and aircraft coordinates. The JavaScript-compatible-API provides multiple techniques to render database coordinates onto the map. Because of these libraries and JavaScript-compatible-API, we gave the Google Maps API a compatibility score of 5. Its wide range of mapping features and emphasis on usability were unmatched and thus we concluded that this API is more stable and polished.





	Accessibility	Capability	Compatibility	Cost	Difficulty
 Google Maps Platform	Account, API Key	5	5	Free	3
 Microsoft Bing Maps	Account, API Key	4	5	Free	3
 mapbox	Account, API Token	3	4	Free	5
 OpenLayers	No account	4	5	Free	3

Table 6: Summarization of Google Maps API, Bing Maps API, MapBox, OpenLayers for the graphical illustration

3.6.3 Proving feasibility

To prove the feasibility of using Google Maps API as our graphical illustration, we will need to prove that we can:

- Display LAT/LON location coordinates from a database onto a map using an XML file
- Render a specific airport's WiFi configuration data on click using an event listener to the Maps JavaScript API code
- Overlay data locations with various WiFi strengths using styling options such as a legend, symbols, and other customizations

4. Technology Integration

Our system consists of two broad execution patterns: data mining from a public flight database and data display in the form of a notification to the user about upload errors and a graphical illustration of WiFi configurations.

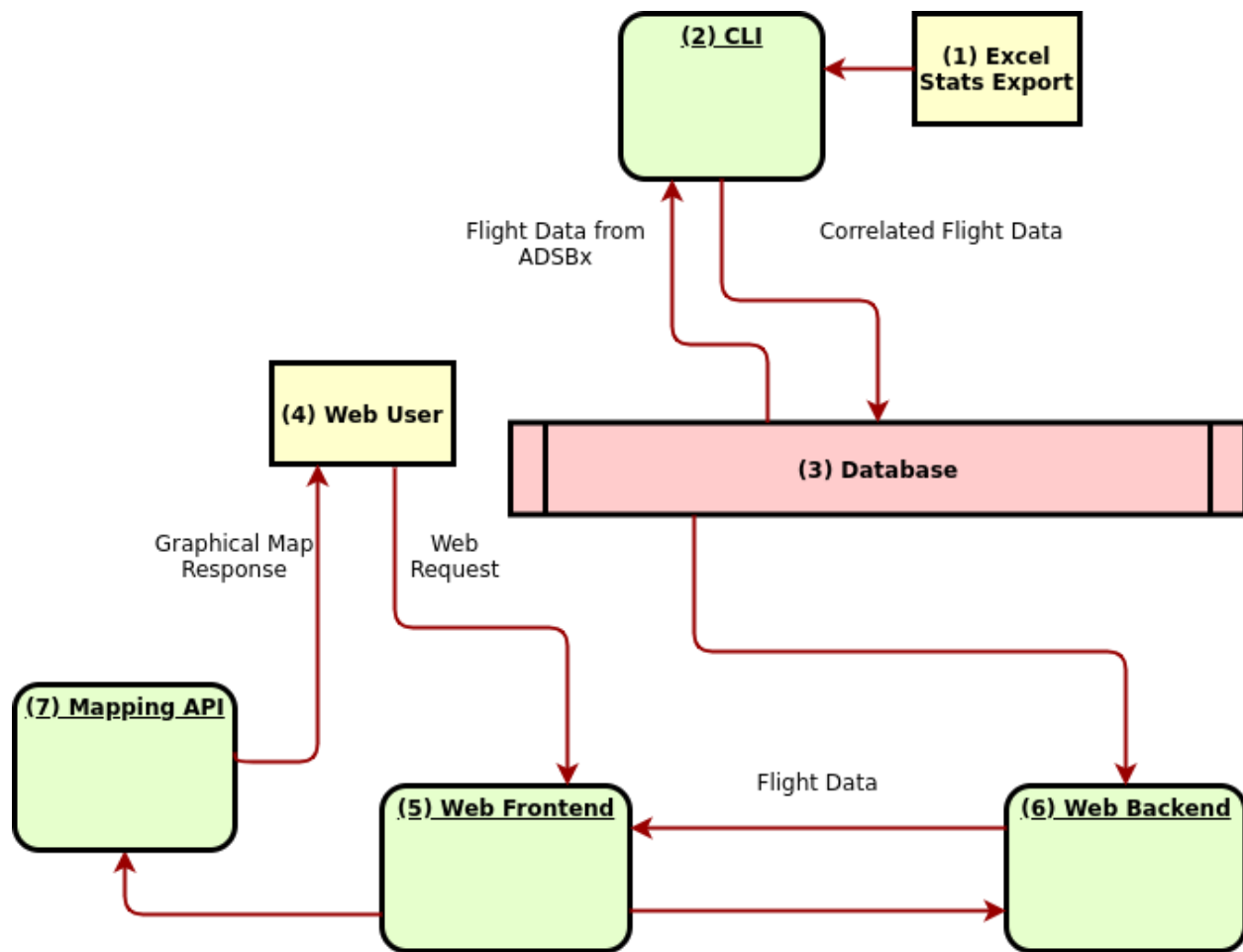


Figure 1: Graphical illustration of the program execution flow

The first part is concerned with ingesting data into our system, starting with the execution of (2) our command-line interface (CLI). The CLI will retrieve the JSON file downloaded from the flight database which will include tail numbers, date, time, and LON/LAT coordinates of where a plane checked in and should have uploaded a report. It will also take in an (1) Excel document containing information about unique, successful uploads which will also include tail numbers, date, time, and LON/LAT locations. This Excel file will be converted into a JSON file so that both datasets can be compared to each other in order to determine upload discrepancies. This dataset comparison will occur through the (3) database API which allows queries to search

between JSON files. The CLI will request every flight found in the flight database and attempt to locate its matching entry in the successful uploads. If there is no match, either a missing entry in the public flight JSON or the successful uploads JSON, then we assume the upload was unsuccessful. If there is a matching entry in both files, the CLI will add an entry into another database which will be used in the data display execution stages.

There are several pitfalls with the data input section. Integrating the command-line app with the public flight database could be difficult due to the potential for incomplete flight data. For example, if a flight takes off, but the flight data cuts out midflight (loss of GPS signal, travels overseas where ASDbX feeders to do connect, etc.), and is not able to locate a landing position, our software will see a flight took place, but won't know if the upload was successful at the landing airport. In instances where there is missing data, we will aggregate flights for a 24-hour period corresponding to the tail number. To account for missing data, we will analyze and classify the flight data according to the following conditions:

- If there are an equal number of flights and uploads documented, we will assume that each upload was a success and use the upload location to set the ending flight airport code.
- If there are multiple flights and only one upload, we will assume that the upload closest to the loss of service, within 12 hours, was the matching flight. The other flights will have no matching upload and will be marked missing. Any flights that have complete data, within airport range ~20 miles, will be correlated to their corresponding airport.

Another obstacle we must account for is the API downtime for the flight data service we use. If the service is down, we will be unable to correlate data correctly. We will first check to see if the service is responding and then check the validity of the data received to see if it is reasonable. If either of these conditions are false, we will put the data in a retest queue to be processed later. This will be noted to the user on the web application.

The second part is about displaying the information we collect to the user in an easy to understand way. This is composed of two parts: the WiFi Configuration Manager and the (5) Customer Connection Module. The WiFi Configuration will use the information loaded to the (6) CLI from the uploaded data Excel file and will display each successful landing point on a (7) map. On each point, the user will be able to see the specific WiFi configuration that was used to upload the diagnostic data to the cloud. This provides an overview of an airport's WiFi configuration throughout the terminal, tarmac, and parking areas. This page will integrate with our (3) backend database to gather this information about specific tail numbers, like those with failed uploads, and aggregate the most successful airport codes, best parking locations for good uploads, and other metrics that may be important to airplane operators. The Customer Connection Module will also utilize the backend to gather this information to graphically illustrate the WiFi configurations and strength of WiFi connections. This module heavily depends on the (7) graphical illustration API that we chose based on our analysis.

A potential drawback for this portion of our project is with our database. During an upload process from the command-line app, our dataset in the database will be incomplete and

illustrating or relaying incomplete information would be unacceptable. We intend to resolve this by marking data that is currently being ingested as “in-progress”. The web app will not pull any “in-progress” data and instead, will only pull data that has been marked “completed”. This strategy is useful in preventing the use of “incomplete data” which includes failed uploads.

5. Conclusion

Since 1903 the first powered airplane has evolved into the connected aircrafts of today, with capabilities to link major cities and small towns in the world, 24 hours a day. These airplanes, however, are constantly grounded due to unplanned mechanical issues. These issues can be avoided with efficient maintenance operations. Our team has been tasked to create a diagnostic application to help engineers and mechanics predict maintenance and mitigate unanticipated errors. The goal of this document is to show both the challenges of this project and describe the planned solutions our team has come up with. The table below lists the technical components and our selected choice of technology. It also includes an indicator which explains our confidence level in the solution on how well it will be able to solve the technical challenge. We have our strongest confidence in our flight database, front end web app, and graphical illustration, each with a score of 5. The command line application, back end web app, and database management have a slightly lower confidence level at 4.5. Throughout this technological feasibility analysis, we have researched many solutions and from those selected the best for our project. We are confident that, with this document, team EnginAir will produce a product that will exceed the expectations of the client.

Challenge	Proposed Solution	Confidence Level (1-5) 1=low confidence, 5=high confidence
Flight Database	ADS-B Exchange	5
Command Line	Java	4.5
Web App Front End	NGINX	5
Web App Back End	Node.js	4.5
Database Management	MongoDB	4
Graphical Illustration	Google Maps JavaScript API	5

Table 6: Summarization of all technical challenges, proposed solutions, and confidence levels