TLDWorkerBee

Team: Austen Christensen, Morgan Lovato, Wei Song
Sponsor: Harlan Mitchell - Honeywell
Mentor: Austin Sanders

Final Report
May 9, 2019

Table of Contents

1. Introduction

Planes are the most popular way to travel quickly to anywhere around the world. For instance, a traveler can get from Phoenix to Los Angeles in a little over an hour by plane when it takes 8+ hours to drive. In order to travel such long distances, these machines must maintain a cruising altitude of 33,000 to 42,000 feet. Every day, there are over one hundred-thousand flights scheduled across the globe with up to one million people in the air at any given point in time. This makes it vital for every plane's engine to constantly be working properly since there is nothing stopping a plane from falling out of the sky other than it's own momentum.

Our team is TLD Worker Bee, and we are working on the project Prototype Time Limited Dispatch (TLD) Application for our sponsor, Harlan Mitchell from Honeywell Aerospace. Honeywell Aerospace is a leading manufacturer of all sorts of aircraft engines ranging from helicopters to commercial airliners. They are the leading manufacturer in engine control systems for a variety of private and commercial jets. These engines and their connected systems generate data every flight that is important for the functionality of their product. While in flight, an engine is constantly reading sensor data and storing it on the onboard computer called the Engine Control Unit(ECU). The computer will read the data and create a time-limited dispatch; time-limited dispatch allows the degraded redundancy dispatch of aircraft. Aircraft can be dispatched with certain control system faults and fault combinations for specified periods of time if the failure rates from those configurations meet certification requirements. The various system faults and fault combinations are assigned to dispatch categories according to these failure rates. This gives the dispatch criteria for the system.

Currently, to gather this data, a technician will physically download the data from the ECU through a wire connection. They do not check it after every flight, but will connect periodically to the ECU to retrieve the data. The cumbersome process of physically connecting to a computer and downloading this data on location greatly limits the amount of flight data to collect.

Our prototype is a webapp that uses an internet connection to connect to the data stored in the cloud for a completely wireless experience. It verifies data integrity before showing the user any data to avoid reading false data. This ensures the mechanic knows exactly what maintenance to perform on the engine from anywhere in the world.

2. Process Overview

2.1 Team Members Roles

Our team consisted of three members: Austen Christensen, Wei Song, and Morgan Lovato. The roles of the team were broken up as follows:

- Austen Christensen
  - Team Lead
  - Client Relations
  - Software Developer
- Morgan Lovato
  - Recorder
  - Document Editor
  - Software Developer
- Wei Song
  - Architect
  - Release Manager
  - Website Manager
  - Software Developer

2.2 Expectations

Our weekly team meetings started with a 5 minute scrum style discussion about what each team member did since the previous meeting. Each meeting lasted about an hour. The decision making process for this project was conducted on a majority rules basis, anytime a decision needed to be made, it was brought up to the team and we voted on the outcome.

2.3 Tools and Documents Standards

- Version Control - Github or email if any difficulties
- Issue Tracking - Trello, unless something better is found
- Word Processing and Presentation - Google Docs and Presentation, Microsoft Word and PowerPoint
- Composition and Review - All progress done in a Google Doc will be reviewed by editor before submission. Print manager responsible for hard copy turn in and team lead responsible for soft copy (email) turn in.

3. Requirements

For our requirements acquisition process, we began by brainstorming different use cases for our product to see how a typical user might interact with our system. We were able to come up with a few requirements of our own, but we wanted our sponsor's opinion on what requirements he felt our system should have as well. We spoke to our sponsor, Harlan Mitchell, about what he would like to see in our final product and how he would like the final product to perform. We explained to him our proposed solution and what we had in mind to fix his problem, and he explained how our proposed solution needs to perform. From this discussion, we gathered a handful of functional and non-functional requirements.

Our functional requirements are as follows:

- [F-SYS1] The web viewer tool shall download the raw data file from the cloud to the user's computer upon user's request.
- [F-SYS2] The web viewer tool shall display the data stored in the database.
- [F-GUI1] The user shall navigate to data by plane tail number.
- [F-GUI2] The GUI shall be adaptive for PCs and tablets using Google Chrome. Note: Adaptive means the user will see the data without needing to side scroll.

These functional requirements describe how the system is expected to function; they cover both system requirements and GUI requirements.

Our non-functional or performance requirements are as follows:

- [P-GUI3] The web viewer tool shall display data onto the web page after receiving it from the cloud.
- [P-DT1] The database shall reject SQL injection 100% of the time.
- [P-SYS3] The web viewer tool shall explicitly validate the data after receiving it from the cloud before displaying it on the web browser. This validation will be done by comparing local and cloud MD5 hash values for the data.

These non-functional requirements describe benchmark goals for the system; they cover system, GUI, and database requirements.

4. Architecture and Implementation

The solution our team has in mind to build for our sponsor is a prototype web application that will serve as a viewing tool for data that is stored in a cloud. The web application will be able to download the data files a user is requesting from the cloud and display them in the web browser of choice. The web viewing tool we build will be usable on Google Chrome and Apple Safari.

We have three main components that will each serve a purpose in our solution: cloud storage, parsing and verification services, and a web application. The cloud storage contains the databases for this prototype: one to store data for processing and one to store data that is already processed. These databases communicate with each other using Python to perform operations on the pre-processed data. We will be using Amazon S3 cloud storage to hold the database containing the pre-processed data and Amazon RDS to hold the database containing the processed data. The RDS database is the one that will be accessed when a user makes a data request in the web application. Before the data reaches the web application, it will be sent through a parsing and a verification tool. These tools/services will ensure data integrity throughout the data flow process. For the web application itself, we will be using Django and Python to create a web page to display data that a user requests from the database. This flow of data can be seen below in figure 1.

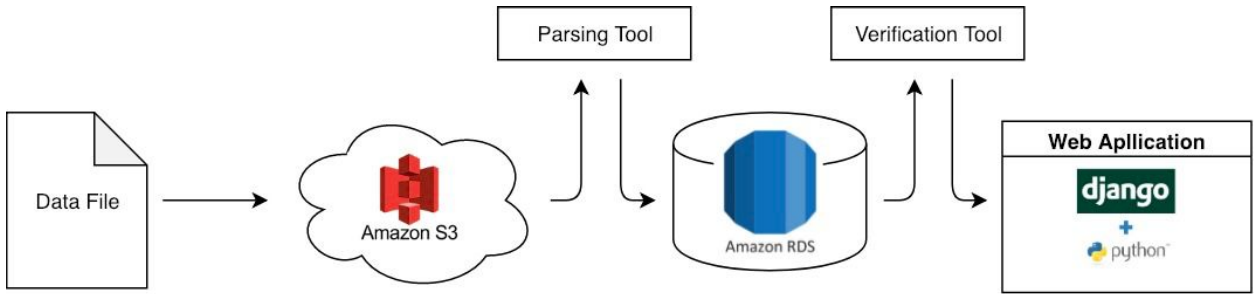

Figure 1. Data Flow Diagram

The data starts by being collected from the ECU and is then sent to the Amazon S3 cloud database. From there, the data is sent through a parsing tool for data processing and is then sent to the Amazon RDS database. Once the user makes a request for data, it is passed through a verification tool before it is displayed to the user in the web application.

4.1 Architectural Overview

The components discussed above will be used to create software that adheres to the Model View Presenter (MVP) model, a key part of which is that data is handled and represented in three separate layers. These layers are as follows:
Database Layer: where the data used by the software is stored(Model)
Presentation Layer: where the data used by the software is displayed(View)
Service Layer: where the data used by the software is parsed and verified.(Presenter)
This separation of responsibilities surrounding the data into separate layers ensures a level of security with data parsing and makes sure that the data that is displayed is accurate. This configuration is shown in figure 2.

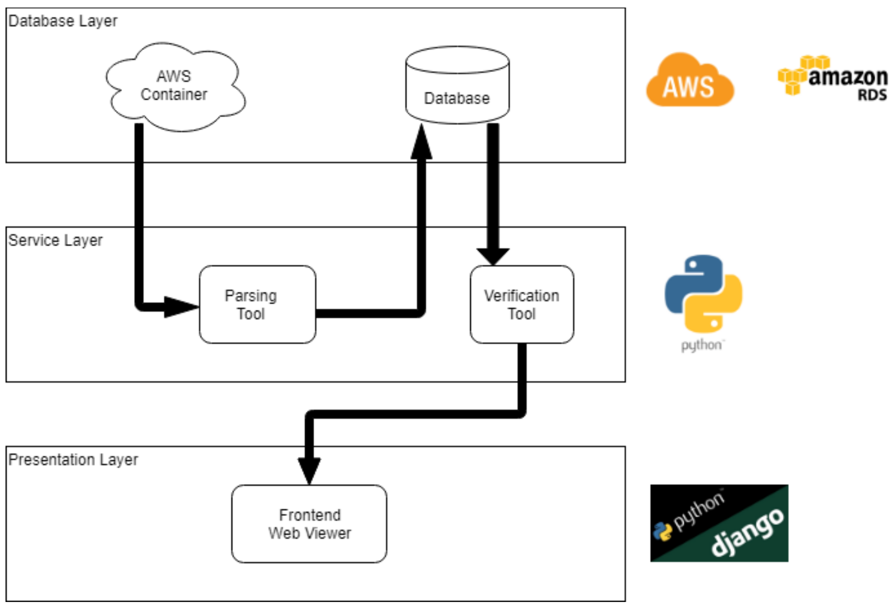

Figure 2. High-Level Architecture Diagram

The Database Layer consists of two major components: Amazon RDS and S3 cloud storage. The S3 storage is where the data is sent from the ECU and is stored for processing. The Amazon RDS database will be used to store SQL tables which will house the processed data.
These two components communicate with the service layer, which will be the core backend of this project, consisting of several modules written in Python that perform operations on the data. These modules will primarily be responsible for parsing through the data in the cloud storage to store in the database and validating data integrity.

After the data is stored in the database, it will wait for the user to request to view. Once the user makes a request, the data will be passed into a verification tool that will verify data integrity before displaying it to the user. In the Presentation Layer, the data will be displayed in a way that is easy for an experienced user to understand.

While this arrangement of layers may seem unnecessary or redundant, it is necessary to ensure data integrity throughout the process. This is due to the importance of an engine to be maintained properly.

4.2 Module Implementation Overview

In this section we will illustrate the finer points of our system’s architecture. For each module in the design, we will provide a brief description of its purpose, create a diagram that outlines how this module fits into the larger whole, and describe exactly how and when this module is interfaced. These technical descriptions will serve as the blueprint to the TLD Application and establish how such a system should be created.

4.2.1 Database Layer

For the database layer, we will be using Amazon RDS to store the processed data for our solution. The database is based on MySQL. As for the structure of the database, we created four tables that are related to each other: The TLD_data table store all the TLD data. The TLD_metadata table contains the metadata from the config file. The TLD_plane table records the affiliations between aircraft and users. The ER diagram of this database structure is shown in Figure 3. Since these data are related to each other, we think utilizing a relational database like MySQL is the best option.

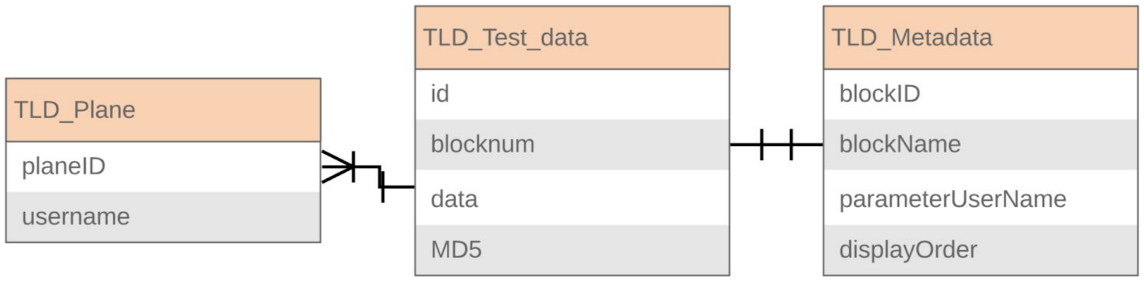

Figure 3. ER Diagram of Database Structure

4.2.2 Services Layer

The service layer will be responsible for two major components: maintaining data integrity and parsing the raw data into a database. When the data file gets stored in the cloud, our parsing tool will begin parsing through the data and passing it into the database along with an MD5 hash code. Once the user makes a request for the data, the data will then get passed into our data integrity checker which will cross check the stored hash with the one created on the user's machine upon receiving. If the MD5s do not match, then the request is made again. If the hashes don't match three times in a row, the data stored in the cloud (if it still exists) will be reparsed into the database. If the file does not exist, an error message will be sent to the Presentation Layer. Figure 4 illustrates this flow of data.

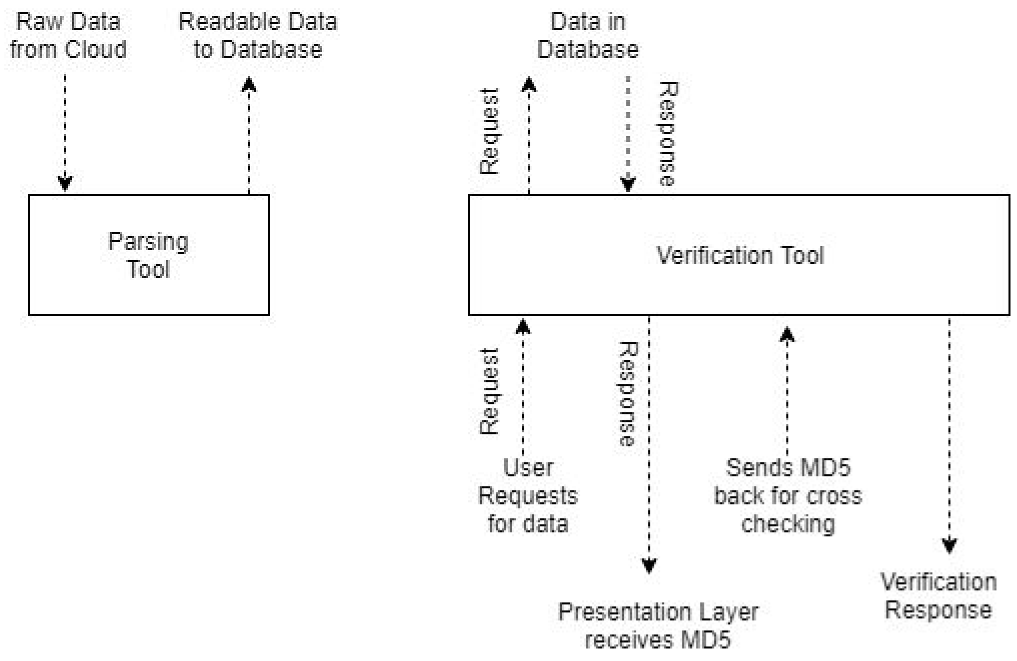

Figure 4. Overview of Service Layer Dataflow

As mentioned above, the Service Layer is composed of two parts. The first part is the parsing tool. The parsing tool is a major factor in the system because it creates the MD5 hashes for each entry in the table so the data can be verified by the verification tool. The data will be passed into the parsing tool as a raw data file which the parsing tool will then break the data up into its correct format based on the corresponding config file. The data will then be passed into the database to be stored along with its MD5 hash. Figure 5 displays this flow and manipulation of data.

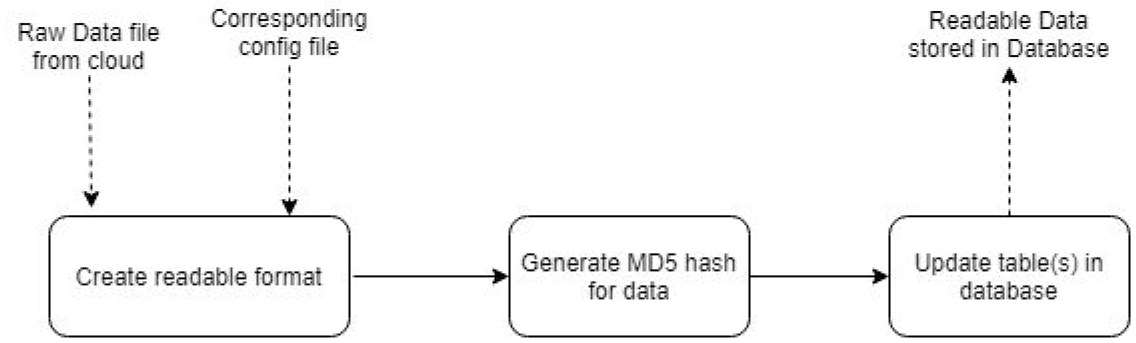
Raw Data file from cloud
Corresponding config file
Readable Data stored in Database
Create readable format
Generate MD5 hash for data
Update table(s) in database

Figure 5. Parsing Tool Dataflow

The second part of the Service Layer is the verification tool. The verification tool is the key component of the system. Since it is so vital that these airplanes are serviced properly, the data needs to maintain its integrity 100% of the time throughout the system. If at any point in time the data changes or loses its integrity, the system needs to know right away and fix the problem. This happens by verifying the MD5 hash before the user sees the data. If the hash doesn't match, then it will try to retrieve the data again. If the data keeps failing, the parsing tool will try to reparse the datafile if it still exists in the cloud. This process is outlined in figure 6.

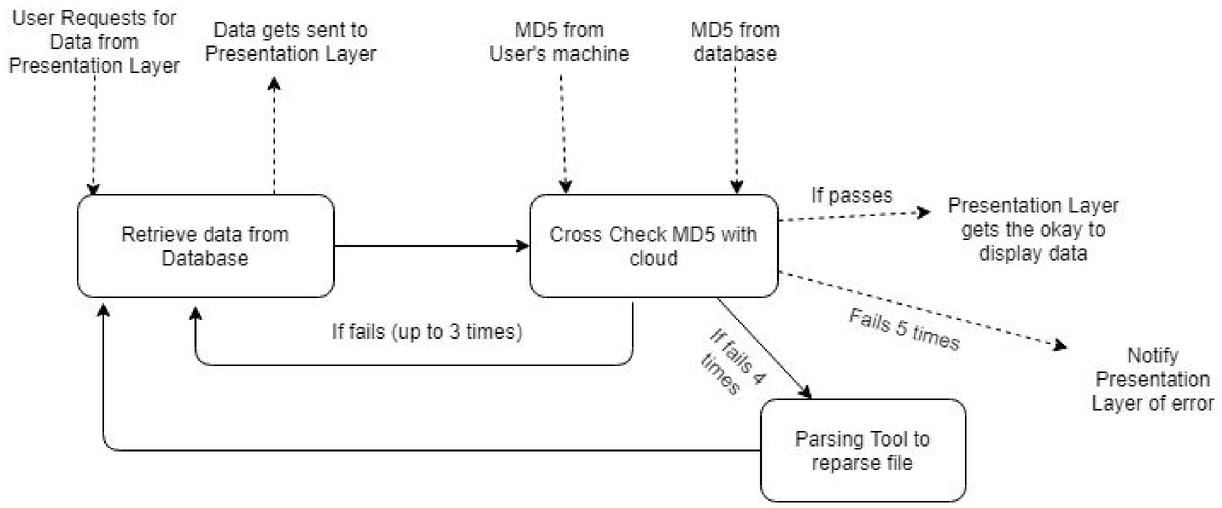
User Requests for Data from Presentation Layer
Data gets sent to Presentation Layer
MD5 from User's machine
MD5 from database
Retrieve data from Database
Cross Check MD5 with cloud
If passes
Presentation Layer gets the okay to display data
If fails (up to 3 times)
If fails 4 times
Fails 5 times
Notify Presentation Layer of error
Parsing Tool to reparse file

Figure 6. Verification Tool Dataflow

4.2.3 Presentation Layer

The presentation layer module is based on Django and consists of several components in order to present the validated TLD data to the clients and users. After being downloaded from the cloud module and processed by the service module, we have to make sure the integrity of TLD data and is ready to present them to the users. Before presenting the data, we still need to compute and compare their MD5 hash value in order to guarantee that the data being displayed is accurate.

In order to help the user locate the data they want to access more conveniently, we designed two view mode: the charts view and the table view. The table view mode is able to provide an organized way to arrange and display the TLD data by using easy-to-read table and grid structure. The chart view mode allows the user to plot the specific TLD data, and generate the line chart of that data to help the user analyze the properties and tendency of the specific TLD data.

We also built an authentication system through Django. This will provide security of the data in the presentation layer. Only the user with correct credential can have permission to access the specific data. And through Django Administration Panel, we developed an access right system that allows different groups of users to have a different level of access permissions (add/delete users, modify/delete TLD data, etc).

We developed most of the components as planned. We didn’t implement the search tool and filter tool in our original plan with three reasons: 1. After consulting with our client, the requirement that the application can “search” some kind of specific data is not necessary. 2. All the TLD data has some kind of relation within each other, and it’s not appropriate to just display individual one. 3. We believe that the chart view mode is a better solution for data visualization than search bar or filter tools.

5. Testing

To test our product, we conducted unit testing, integration testing, and usability testing. Unit testing is a software testing method that tests the individual units or components, which are the smallest part of the software, in order to validate that each unit of the software performs as designed. It is the first level of software testing during the whole implementation cycle, and it is performed prior to integration testing or usability testing. Integration testing is the testing of how individual parts of a program work together; the components tested in the unit testing are tested to see how they perform as a group. Integration testing is performed after the completion of unit testing and before the start of usability testing. The goal of integration testing is to find any problems with how units interact when they are integrated as a whole. Usability testing is the evaluation of a product by testing it with the individuals who will be using the product. Typical usability testing includes asking participants to complete typical tasks while the observers watch and see how they perform. Usability testing is performed after the completion of both unit testing and integration testing. The goal of usability testing it to find any problems with the product’s interface or usability and see how satisfied the users are with the product.

For unit testing, our team used Pytest since our code is written in Python. The Pytest framework is a full-featured Python testing tool, which supports both simple unit testing and complex functional testing for applications and libraries. It also supports parameterization to manage the test cases. There were 8 test units and 47 test cases in total.

For integration testing, our team used Travis CI as it works well with Python, Travis CI is a continuous integration framework that provides continuous integration and a testing environment for our product. We use this framework to test all interactions between the different modules or layers of our product.

For usability testing, our team asked a group of certified aircraft technicians to use our product and evaluated how easily they were able to navigate our system. We had two separate parts to our usability testing: categorical acceptance and live usability. For categorical acceptance, a user was given flashcards with two different categories written on them and they were asked to match them according to what they found most logical. For live usability, we recorded users interacting with our application given a testing script and the user was asked to talk out loud about how they think as they were interacting with the application.

6. Project Timeline

For the first two months of Capstone, September and October, our team worked on setting up the standards for the rest of Capstone and getting our team organized. After meeting with our sponsor, we began our Technological Feasibility Analysis where the team gained an understanding of the possible challenges and solutions for our project. By November we had begun our requirements acquisition and given our first Design Review Presentation. We also presented a technical prototype demo to our mentor. By December, which was the end of the first semester of Capstone, we had a completed Requirements Document outlining all the requirements for our project.

In January, at the start of the new semester, we began implementation. We started by implementing the database layer and finished up in February with the implementation of our service layer and presentation layer. In February we also finished our Software Design document and presented our second Design Review. In March we presented our Full Prototype Demo to our mentor and began planning our Software Testing. In

April, the team conducted unit and integration testing before presenting out third and final Design Review. Following this was the UGRADS Capstone Conference where we gave our final presentation and presented our team's poster. We finished the month with a Final Product Demo. In May, the last month of Capstone, the team conducted usability testing, finished the User Manual for our product, and presented our final product to our sponsor. Below is an Overall Project Timeline (list of milestones) for the entirety of Capstone:

- September 2018
  - Team Standards
- October 2018
  - Team Inventory
- November 2018
  - Technological Feasibility Analysis
  - Requirements Acquisition
  - Design Review 1
  - Technical Prototype Demo
- December 2018
  - Requirements Document
- January 2019
  - Implementation Begins
  - Implement Database Layer
- February 2019
  - Implement Service Layer
  - Implement Presentation Layer
  - Software Design Document Finished
  - Design Review 2
- March 2019
  - Full Prototype Demo
  - Software Testing Planning
- April 2019
  - Unit Testing
  - Integration Testing
  - Design Review 3
  - UGRADS Capstone Conference
  - Final Product Demo
- May 2019
  - Usability Testing
  - User Manual
  - Final Product Delivery

7. Future Work

As far as future work is concerned, our team would like to see a few things implemented in the next iteration of the project:

1. The parsing tool converting the raw binary file, right now our project only reads the already converted .txt file.
2. A more precise way of graphing the designated fields. We would like to clean up the chart view so non-necessary graphs aren't displayed (i.e. The fields with only binary values).
3. A way to store the ECU generated CRC values with the data in the database to make it easier to compare with the raw binary file.
4. Fix the table view to correctly line up the data with their headers. This can be fixed with the converting of the binary file. As it stands the headers in the table view are dynamic and can change easily with the config file.
5. An automatic refresh system where the verification tool automatically parses the data from the raw binary file into the database when an MD5 check returns false. This can be improved by stepping through the raw binary file until the specific line of data is found and parsing only that single line of data into the database instead of the whole file.

Other future work would be discovered during the user testing phase of the project. Due to time and resources, our team was not able to conduct user testing.

8. Conclusion

Our client was looking for a way to change their current workflow process of collecting and viewing TLD data. Currently, aircraft technicians have to manually plug in a laptop via USB to the Engine Control Unit of whichever plane they are looking at and view the data there on their laptop. This tedious process of manually connecting a computer to the Engine Control Unit and downloading the necessary data on sight is very costly, time consuming, and inefficient.

Our proposed solution will take this manual process and make it completely cloud and web application based. Our client has currently completed the process of transferring the data from the Engine Control Unit to a database stored in the cloud via Bluetooth, but they have no way of viewing this data in the cloud. Our solution is to create a web application that will allow aircraft technicians to view the data from the cloud from anywhere using a web browser. We have created a prototype web application that serves a data viewing tool for the data stored in the cloud. This web viewing tool downloads the data from the cloud and displays it in a web browser without compromising any data integrity. We hope our solution will offer value to our client via modernization of an outdated and inefficient process and ultimately save them time. We also hope the prototype we have created will be of use to our client when they implement a solution of their own.

Our team is happy to have been able to work on this project and feel it was a great experience for us. While the completion of this project was not always easy and clear-cut, we are glad to have been able to overcome our challenges and complete a product for our sponsor.

9. Appendix A: Development Environment and Toolchain

9.1 Hardware

The application was developed on Linux / Mac OS X. The main develop machine is a MacBook Pro 2017 with 2.3 GHz Intel Core i5 CPU and 8 GB 2133 MHz LPDDR3 Memory. And for cloud and database, we choose the services provided by Amazon AWS (S3, RDS, Elastic Beanstalk). Usually, the development process can be done with any platform with Python environment, and there are no minimum hardware requirements for this project. However, we do recommend to develop through a UNIX-based operating system (Linux, MacOS, Ubuntu, CentOS, etc) that has the similar environment with Elastic Beanstalk (based on Linux), which is much easier to deploy your code to AWS services.

9.2 Toolchain

9.2.1 Python Environment

- Python 3.6 (Python is an interpreted, high-level, general-purpose programming language. All the program and components of this project is written in Python)
- Django 2.2.1 (Django is a high-level Python Web framework. This is the basic of the web application.)
- Pip (Package manager for Python. Use it to install other python components or libraries like Virtualenv, awsebcli and Urllib3)
- Virtualenv (Python Virtual Environment. By using a virtual environment, you can discern exactly which packages are needed by your application so that the required packages are installed on the AWS instances that are running your application.)

- Awsebcli (Elastic Beanstalk Command Line Interface (EB CLI). This is used to initialize your application with the files necessary for deploying with Elastic Beanstalk.)
- Urllib3 (urllib3 is a powerful, sanity-friendly HTTP client for Python. We use it to transfer the raw TLD data from the cloud to the local environment)

9.2.2 Code Editor

- Sublime Text 3 (Sublime Text is a proprietary cross-platform source code editor with a Python API. It’s easy to develop code with Sublime Text 3)
- vim (Vim is a highly configurable text editor built to enable efficient text editing.)

9.2.3 Database

- MySQL 5.1 (MySQL is a relational database management system. We use MySQL to manage the database)
- Sequel Pro (Sequel Pro is a fast, easy-to-use Mac database management application for working with MySQL databases.)
- MySQL Workbench (MySQL Workbench is a visual database design tool)

9.3 Setup

9.3.1 Create an AWS account

To deploy this program, you need to create an Amazon AWS account (link: https://portal.aws.amazon.com/). Amazon AWS provides different services and solutions with different price and usage limit. However, for this project, we would need at least 3 services: Amazon S3 (To store the raw TLD file), RDS (local database) and Elastic Beanstalk (runtime environment).

9.3.2 RDS Database Setup

To create a database through Amazon RDS:

- Go to the AWS Management Console
- Click on the **RDS** link below the database column
- Click on the Create database button
- On the following pages, select **MySQL** -> **Production - MySQL**

Enter the DB instance identifier, Master Username and Master Password

- On the following page, click on the Create database button to complete creating database

Once you create a database through RDS, you would have database hostname, username and password. Keep these informations since you would need them in the following steps.

9.3.3 AWS S3 Setup

To create a bucket that store the raw TLD file on the cloud through AWS S3:

- Go to the AWS Management Console
- Click on the **S3** link below the storage column
- Click on the + Create bucket button
- Name your bucket as TLD.cloud
- On the following page, click on the Create bucket button to complete creating bucket on AWS S3.
- Upload a raw TLD file (ECFR_1hour_data_dump.txt) to the bucket you just created.

9.3.4 Set up the environment on local machine

Before deploying the program to the elastic beanstalk, we need to make sure that the program run perfectly in the local virtual environment. That means you need to install the following required components and libraries in your local machine first:

- Python 3.6
- Pip
- Virtualenv
- awsebcli

Amazon AWS provides the document on how to setup the python development environment. Follow the instructions (Link: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/python-development-environment.html) to set up the environment.

Besides, part of this project relies on Urllib3 to transfer the data from cloud to local environment. Install it first (link: https://urllib3.readthedocs.io/en/latest/) before the next steps.

9.3.5 Deploy Django Application to Elastic Beanstalk

As a part of the final delivery, we provide a zip file contains with all program source code. You can also download it from the project Github (link: https://github.com/fssongwei/Capstone).

Simply follow the instructions provided by Amazon AWS (link: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create-deploy-python-django.html) to deploy the program to Elastic Beanstalk.

9.4 Production Cycle

Now that a TLD application is running on Elastic Beanstalk. In future work, this application may need to update. In this section, we will explain the production cycle and mechanism of this application.

In this Django application we have two modules: User authentication module (in the **users** folder) and data viewing tool module (in the **TLD** folder).

A common way of explaining Django's architecture is to describe it as Model-Template-View (MTV). In each module, Django will construct three main files: model, views, and templates.

Models in Django mainly provides the support of different databases. In models.py file in TLD folder, we build the test_data model, plane model, and metadata model. Each of them is corresponding to a table in the database. You can change, add or delete the models in it to change the structure of the database. Notice that every time you change the model, you need to run the following commands to apply the models to the database:

- python manage.py makemigrations
- python manage.py migrate

View is simply a Python function that takes a web request and returns a web response. This response can be the HTML contents of a Web page, 404 error, XML or image, etc. In the views.py file in TLD module, there are 3 views function:

- **tableview(request)** function will fetch all the data from the local database, calculate the MD5 values of TLD data, and response the web request.

- **chartview(request)** function will only fetch the data that the user selects to plot from the local database, calculate and match the MD5 value, and response the web request.
- **dataprocess(request)** function will fetch the raw TLD data file from Amazon AWS S3 through Urllib3.

You should only modify these functions in order to change the table view or chart view, or the way to fetch the data from the cloud.

Templates in Django are mainly in charge of the appearance of this application. It also receives the data from view functions and displays them in the browser. Template files are stored in the **templates** folder, written in HTML file. We also used Bootstrap as our front-end framework here.

Also, the database connection information is stored in settings.py file under the TLDapplication folder. In here we have two databases: the **default** database stored the user authentication data, and the **db1** database stored the TLD data in RDS. If you change your database provider in thefuture, then you would need to update the information in this file. Also, since there are two databases in this application, we develop a database router in the database_app_router.py file under the TLDapplication folder, to help the application decide when to use which database. You can modify this router if you want to add another database in the future.

Every time you modify the code and run the Django application (through the command: python manage.py runserver), Django will compile all the file automatically. Once you finish updating your code, you would need to deploy the program to the AWS Elastic Beanstalk. Simply follow the instructions provided by Amazon AWS (link: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create-deploy-python-django.html) to deploy the program to Elastic Beanstalk.