

CW3-ILP

November 2025

1 Introduction

I developed a self-help drone flight data analysis tool in coursework 3. This tool uses the K-means clustering algorithm to group delivery locations into clusters, treat each cluster centre as a candidate new service point, and display them on a map. It also provides data visualisations to answer questions such as which drone is most used, which service point is most used and which day of the week is the busiest. Moreover, it allows the user to see the raw data.

2 Target Audience & Motivation

In a hypothetical situation where the system I developed in CW2 is deployed, the system manager may want to adjust the system settings to serve customers better. So, I built this tool to help the system manager better understand system usage. With this tool, they will be able to interpret data more quickly and extract patterns, thereby optimising the system.

One key innovation is that the system plans new service point locations based on real delivery data and turns raw logs into actionable insights without the need for a human data analyst.

3 System Design

3.1 AI Usage & My Implementation

In this coursework, I wrote two pieces of code. First, I added a logging functionality to my CW2 code. I also wrote a Python script with the functions of applying K-means and generating data visualisation graphs. I used GitHub Copilot to create and adapt a Flask-based Python service that calls my Python script and displays its results on a webpage for presentation.

3.2 Architecture

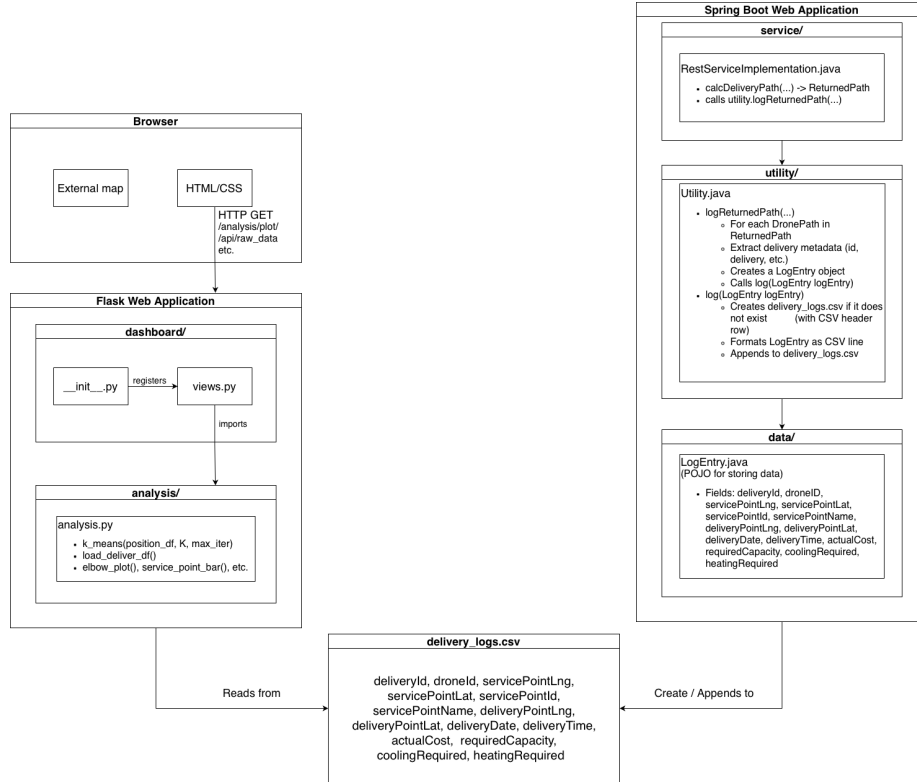


Figure 1: System Architecture

Now, I will introduce the system's composition. The system is composed of the Java logger inside my CW2 code, which is responsible for logging delivery information whenever a new dispatch order is received; the Flask-based Python backend, which applies data analysis techniques such as K-means and data visualisations to the log data and generates results; and an HTML, CSS and JavaScript based frontend, which displays the results produced by the Python backend. I designed the system so that the CW2 Java service is independent of the Python service. This separation keeps the core dispatch logic stable while allowing the analytics code to evolve, and it also means the Java and Python services can run concurrently. When a new dispatch is received, the Java service will log relevant data to the log file. When the user requests the map, graphs or raw data again, the Python service reads the log file, computes new results, and displays them to the user via the frontend.

3.3 Tools

I used several tools in this project. I used the Java Path, Paths and Files packages to implement my logging function. In my Python script, I used pandas and matplotlib to process and visualise the data, and Flask to expose the analysis as a web service. Finally, I used HTML, CSS and JavaScript for the frontend. These choices allowed me to reuse my CW2 Java code for logging, leverage Python's data science libraries for analysis, and use a standard web stack for presentation.

3.4 System Flow

In this section, I will walk through how the system works. When requests are sent to the `calcDeliveryPath` or `calcDeliveryPathAsGeoJson` endpoints of the Java REST service, the method `calcDeliveryPath(...)` in the `RestServiceImpl` class will be called. Just before the function returns the result of type `ReturnedPath`, a method named `logReturnedPath(...)` in the `Utility` class is activated. This method takes an object of the `ReturnedPath` class, extracts the necessary information, and stores it in a DTO of the `LogEntry` class, which I defined in the data module. Then, the method `log(...)` will be called, and that function will do two things. First, create the `delivery_logs.csv` file if it does not exist, and append column headings. Second, open the file once it is found or created, and append the data from the `LogEntry` DTO as a new line in `delivery_logs.csv`.

The Python service, on the other hand, will attempt to access the `delivery_logs.csv` and process the data. When the user requests a map, graph, or raw data, the frontend sends several GET requests to the Python backend, implemented in `views.py` under the dashboard module. This file uses the blueprint package, which Flask supports to build all dependencies. `views.py` imports the drawing and K-means functions that I implemented in `analysis.py` under the analysis module. Whenever the user requests raw data, a map with labels, or graphs, the corresponding endpoints in `views.py` will be invoked. Then the corresponding endpoint will call `load_delivery_df()`, defined in `analysis.py`, which loads the raw data from a CSV file into a pandas `DataFrame`. Then, depending on the requested data type, the endpoint will call a different function in `analysis.py` to process the data and return the results. Then, the endpoints will send the result from the `analysis.py` functions to the frontend for displaying.

In this way, I separated concerns into modules: `analysis.py` is responsible only for data processing, `views.py` for forwarding the results to the frontend for display, and the Java logger for providing the data.

4 Limitation & Future Work

Although the system's idea sounds promising, there are a few essential improvements I can make. Right now, everyone with the correct URL can access the

website and see the data stored, which does not comply with data security measures. In the future, authentication can be added to prevent unauthorised access to the data. Second, the K-means algorithm does not account for no-fly zones, so the newly generated service point may fall within a no-fly zone or be located at a location that's not actually better than existing service points. To solve this, I should modify the K-means algorithm to account for the no-fly zones. I could also mark the delivery points on the map using different colours based on the K-means clusters they've been assigned to, to improve information density.

5 Conclusion

In conclusion, this system aims to help the system manager better understand how users will use the system, enabling them to respond to trends more quickly and meet users' needs.