COMP30830
Dublin Bikes Report

Group No: 2

EC2 Address: http://34.242.160.31

GitHub: https://github.com/monsieurcherry/Software-Engineering-Project

Team Members and Contributions:

1. Itgel Ganbold (40%)
2. Miaomiao Shi (30%)
3. Sakura Hayashi (30%)

Product owner: Karl Roe

Table of Contents

1. Project Overview

1.1 Introduction

This project management report outlines the development of a web application showing occupancy and weather information for Dublin Bikes. The project was carried out using the Scrum methodology, which allowed our team to work collaboratively and efficiently while fulfilling our goals.

Dublin Bikes is a public, self-service bicycle rental scheme first introduced in Dublin in 2009. The scheme encourages sustainable transportation and aims to improve traffic congestion by allowing users to rent bicycles from different places across the city.

1.2 Objectives

The main objective of this project is to create a web application that displays real-time data on bike availability and weather information for Dublin Bikes users. The application will also provide a trip planner feature that employs a predictive model to inform users of bike availability. We aim to improve the overall user experience and to further promote sustainable modes of transport.

By creating a user-friendly interface, we aim to deliver an application that is accessible to all users, regardless of their technical proficiency. It is designed to be simple and intuitive, with fast and efficient access to all necessary information displayed clearly and concisely.

1.3 Target

The target audience for this web application is primarily Dublin Bikes users, both regular and casual, who are looking for real-time information on the city's weather conditions and bike availability. The application is also aimed at potential users who would like to know more about using Dublin Bikes as a transit option.

1.4 App Functionality

The application's key features and functionality include:

1. Bike availability information: The application displays real-time information on bike availability for all Dublin Bikes stations across the city.
2. Weather information: The application displays real-time weather data for the city of Dublin, including temperature and wind speed. Users can view this information together with data on bike availability to make an informed decision on whether to use Dublin Bikes or not.
3. Stations locations: Users can view the location of each Dublin Bikes station and the number of bikes available at each one using the application's interactive map.
4. Route planner: The application provides a route planner feature, allowing users to plan their journey by entering their starting point and destination. The application also suggests routes for other means of transport, such as walking, driving, and using public transportation.
5. Bike parking spot availability information: In addition to providing information on bike availability, the application also offers real-time data on parking spot availability at Dublin Bikes stations. This allows users to plan their route and locate stations with open parking spots ahead of time, minimising the time and effort required to find a parking space and thereby enhancing user experience.
6. Average overall daily availability: The application's bike availability prediction feature uses data processing techniques to display the average overall daily number of bikes available at a given Dublin Bikes station using data from the past week.
7. Future daily availability: The application uses predictive modelling in conjunction with weather forecast information to display expected bike availability for a given day at each station. It is able to predict up to 5 days in advance. This allows users to plan their routes and locate stations with available bikes ahead of time, minimising the chances of finding an empty station. This prediction tool offers a more reliable and accurate picture of bike availability and thereby enhances user experience.
8. Weather forecast: The application generates the weather forecast for a given hour on a given day. This allows users to plan their journeys based on the weather ahead of time.

1.5 App Structure

The application uses a single page to display all necessary information. It has a user-friendly, straightforward layout, making it easy for users to navigate each section and access all the key features and functionalities. The main sections of the application are as follows:

- Home Page: This is the only page of the application. It provides users with an overview of the key features and functionalities of the application.
- Map and Stations Information: This section displays an interactive map with markers indicating the locations of each Dublin Bikes station. Upon clicking a marker, users can view more detailed information on bikes and parking space availability.
- Parking Spots and Bike Availability: These sections offer real-time information on the availability of parking spaces and bikes at each Dublin Bikes station. They are displayed as buttons on the map. The number of bikes available at each station is shown on the markers by default. Upon clicking the parking spots button, the number of parking spots available at each station will appear on each marker. The numbers on the markers are colour coded: any number below 4 is displayed in red while anything above is displayed in white. By doing this, we provide the user with a visual representation of the number of bikes and parking spaces available at the present moment.
- Route Planner: This section allows users to enter their starting point and destination. The application will provide them with a cycling route by default but the user can also compare this route with that of other forms of transportation like walking, driving, and using public transport. Moreover, when the search field is activated, clicking on a marker will automatically input the station location into the search field for the user.
- Average Overall Daily Availability and Average Availability per Hour: These sections are displayed as charts. Upon clicking a station/marker, the user will see the average overall daily availability of bikes at that station. After specifying a particular day and station, the user will see a different chart displaying the expected number of bikes available per hour.
- Real-Time Weather and Weather Forecast: This section displays the real-time weather for Dublin. This changes as the user modifies the day and time of their journey. The wind speed icon is colour coded, with yellow indicating anything between 10m/s to 17m/s, and red indicating anything above 17m/s.

2. Architecture

2.1 Overview

The server, frontend, and database make up the web application's architecture. The server is implemented using Flask, a Python-based web framework. It handles incoming HTTP requests and returns the requested data to the frontend. The frontend is built using HTML, CSS, and JavaScript to display occupancy and weather information for Dublin Bikes.

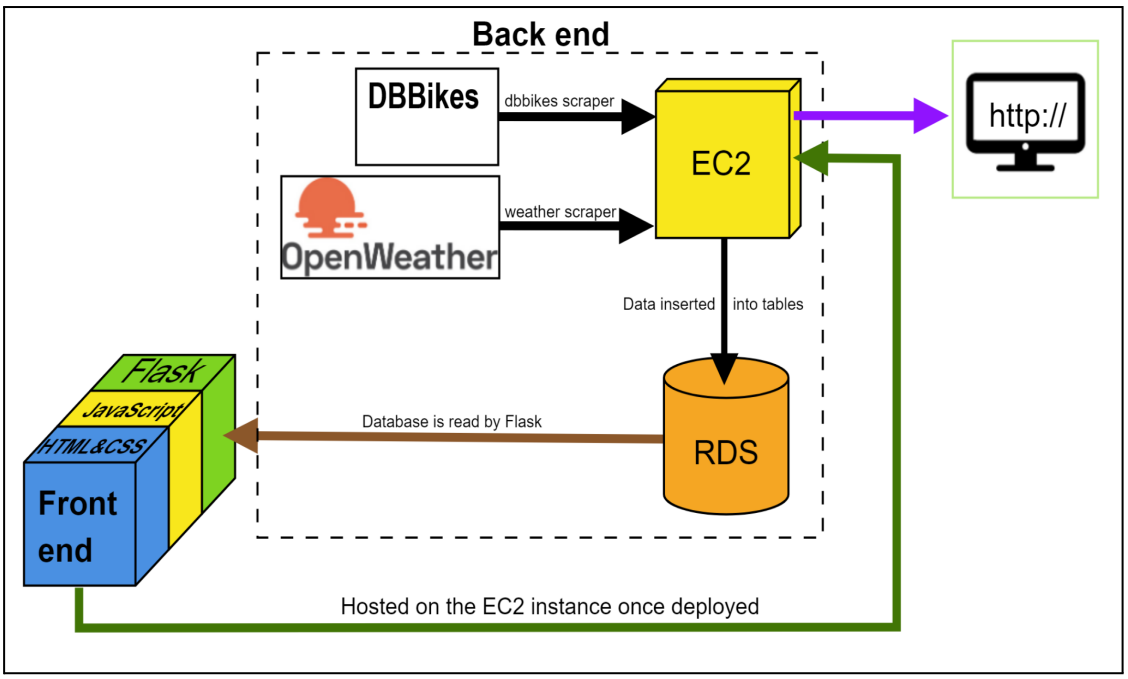


Figure 1: Web Application Structure.

The server is hosted on an Elastic Compute Cloud (EC2) instance, which continuously runs a number of Python scripts. These scripts are set up to be executed at a set time interval to make API calls to two external APIs. The data collected/scraped from the external APIs are pushed and stored in a Relational Database Service (RDS) database. The Flask application interacts with the RDS database and uses JavaScript to manipulate dynamic content in the frontend.

Overall, this architecture generates a scalable and adaptable web application that deals with incoming requests effectively and retrieves data from external sources.

2.2 Cloud Platform

The application was deployed on the Amazon Web Services (AWS) cloud platform. To launch the application, we used various AWS services, such as their RDS database and security services. The Flask-based server was deployed on an EC2 instance.

By using AWS, we managed to deploy the application in a scalable, reliable, and cost-effective manner.

2.3 Technology Used

2.3.1 Frontend: HTML, CSS, JavaScript

The **HTML code** provides the basic structure and content of the web page. The body element consists of a container div, which itself holds several divs that make up the main sections of the page: a map div, a graph div, a weather div, and a location input div. These divs are specified with unique IDs so that they can be styled and manipulated using CSS and JavaScript.

The map div is where Google Maps is displayed. It has two buttons that are contained in two divs: the marker-selector-bike div and the marker-selector-parking div. These buttons allow users to toggle between bike availability markers and parking spot markers on the map. The map is displayed asynchronously when the page is loaded.

The graph-parent div contains a child div called graph1, where a chart is displayed using the Chart.js library. Using the OpenWeatherMap API, the weather div presents the current weather data. The location input div contains form elements that allow users to input their starting location and destination, select a date and time for their bike journey, and choose between different forms of travel using radio buttons.

The linked **CSS code** styles the web page. The first block of code applies to the body of the page and sets the general elements such as the font family, font colour, and background colour. The container is styled to have a grid layout. The remaining blocks of code mainly set the positioning of each section within the grid layout and their font size and colour.

The **JavaScript code**, linked within the HTML code, adds interactivity to the web page. The code initialises the Google Map, centred on Dublin, and adds interactive markers to the map for each Dublin Bikes station.

The code also calls functions to fetch the bike occupancy and weather data before converting it to JSON. The user then has the choice of viewing the number of available bikes or parking spots on each marker. The colour of the number in the markers changes depending on the number of bikes available.

2.3.2 Backend: Python, Flask, SQL

A **Python script** was written to develop the application using the **Flask web framework**. To obtain information on bike-sharing stations, current availability, and weather conditions, the application establishes a connection to a **MySQL database**. The route handlers *get_stations*, *get_current_stations_info*, and *get_current_weather_info* all produce JSON objects that contain details on the bike-sharing stations, their current availability, and the weather. The data is returned as a JSON response by these functions after being retrieved from the database using SQL queries.

The weather data is retrieved through a weather forecast function that uses the OpenWeatherMap API. To specify the date and time for the weather prediction, the function takes a future date and time as arguments. After calculating the future date and time, the new date-time's Unix timestamp is computed. The script searches the forecast JSON file for the date that is the closest to that timestamp. This weather data is then extracted and returned as a pandas DataFrame.

To predict future bike availability, the script calls a function that loads a pre-trained machine-learning model (linear regression) for a given station. The model predicts the anticipated number of bikes and bike stands at a given station for a given time and date by using inputs such as weather conditions, day of the week, time of day, and historical bike usage patterns. The model is built on a regression technique that has been trained to determine the correlation between the input features (the weather, date and time, bike usage patterns, etc.) and the output variables. The model's output provides an estimate of the future availability of bikes and bike stands at a particular station based on the given time and location.

2.3.3 Data Analytics: Python/Jupyter Notebook

We trained our machine learning model to predict bike availability using **Python** programming language and **Jupyter Notebook**. We used several

libraries like *scikit-learn* for machine learning, pandas for data analysis, and *matplotlib* for data visualisation.

By using Jupyter Notebook, we were able to write and test our code for data preprocessing, feature engineering, model training, and evaluation effectively and interactively. The Notebook provided a user-friendly environment for visualising our data and results.

2.4 Architecture Tools Difficulties

Some of the major issues we faced include deploying a Python file with the incorrect date format. While it took a long while to figure out the error, we successfully fixed it and managed to collect station information. The issue was caused by our inability to stop and check *nohup* processes that were running on EC2. Essentially, whenever we adjusted our scraper, we were not properly stopping the previous scraper instance, which would overwrite the new scraper each time.

The deployment of the application itself was extremely dreadful. Nothing seemed to work as intended: as one bug was fixed, another one appeared. Our previous attempts at optimisation proved to be too demanding for EC2, causing server crashes. We attempted to optimise by running a Flask function that preloaded all necessary data during the application's initial run, which involved calling a table with 1.5 million rows. The file size was too large for transfer, crashing the EC2 server.

While exporting our environment, packages went missing due to improper environment export caused by slow package installation resulting in several errors. The web scraping process caused memory depletion on our EC2 instance. With the assistance of our lecturer and TA, we resolved these problems. Debugging was difficult as the tools we used to set up the deployment, namely *Gunicorn* and *NginX,* provided limited error logging.

3. Design

3.1 How the Application Works

The web application is a simple one-page website that allows users to explore their options on the map by comparing available bikes and parking spaces at different locations around Dublin. Users have the option to select their starting and end locations by either clicking on a station marker after activating the search field or manually entering their addresses. Each marker clearly displays the available number of bikes and parking spots. The displayed weather information includes both current weather conditions and the weather forecast that users can select for specific days and times. If no day and time are specified, clicking on a station marker will display the average bike availability at that station for the past week. On the other hand, if a user selects a day in the future, clicking on a marker will display the predicted bike availability at that station for the selected day.

3.2 Meeting the Technical Requirements

Below is a table outlining the technical requirements we needed to fulfil and how we accomplished each requirement.

Technical Requirements	*Implementation*
1. Data collection through JCDecaux API	Scraper on our EC2 collected 1.5 million rows of bike station data.
2. Data management/storage in RDS DB on AWS	Our EC2 instance fed the data to our RDS on AWS.
3. Display bike stations on map	Successfully displayed the stations and the number of bikes and stands.
4. Flask web application (Python)	Used Flask as the main web framework.
5. Occupancy information	An info window appears when the mouse hovers over a station marker that shows occupancy information.
6. Weather information	Current and future weather information is shown.
7. Interactivity (click, API request, handle response)	Clicking on a station when a different day is selected shows predicted occupancy.

8. ML model for predicting occupancy based on weather patterns, trained on collected data	A linear model was trained using the historic weather information and station occupancy data.
9. The project served on a named host over the web on your EC2 instance	The project was successfully deployed to our EC2 instance using Gunicorn and Nginx as a proxy.
10. GitHub repo, including source code, logs, commit history, branches and so on	The GitHub link is available at the cover of this report.

3.3 User Flow Diagram

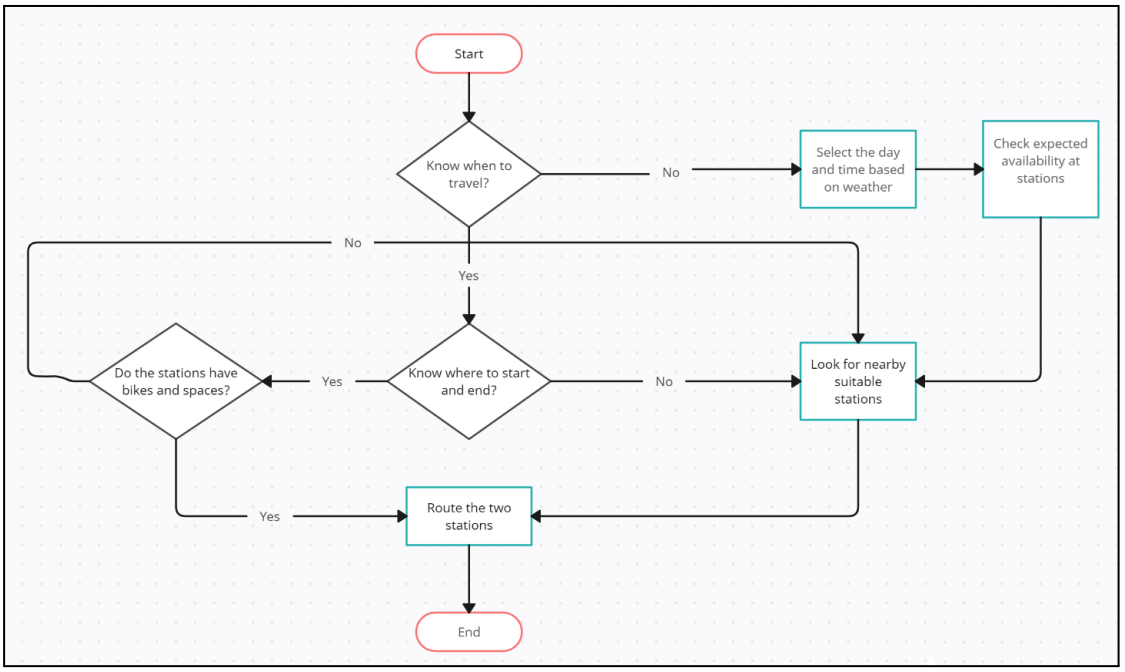


Figure 2: User Flow Diagram.

3.4 Interactivity

The app was envisioned to be highly interactive and provide a dynamic and seamless user experience from the beginning. As a result, the implemented functionalities required extensive information exchange with other parts of the back-end architecture.

When the app runs, it retrieves weather data from the OpenWeather API to display weather information. Using the current availability table stored in RDS,

the *infowindow* displays the current bike and parking spot availability each time a user clicks on a marker.

When a user clicks on a marker without selecting a specific day and time, the *infowindow* displays the average availability from the past week. This information is obtained by retrieving the last 2016 rows (7 days) from the historic availability table where the station number matches the marker. However, if the user selects a day in the future and clicks on a marker, a function is called to use the machine learning model to predict the bike availability at that station. This function uses the weather forecast information to make the prediction.

Furthermore, a routing functionality was implemented. When the user activates the search inputs, they can click on the starting and ending station markers to generate a bike route between the two stations. Alternatively, they can enter their starting and ending addresses to generate walking, driving, cycling or public transit routes.

3.5 Early Mockups - Sketches and Notes

Figure 3: Design mockup.

The web application mockup shown above draws inspiration from the examples demonstrated in class. It includes almost all the features specified in the project description. However, our final version of the application has

additional features that were not initially considered during the design phase, such as the ability to click and create a route and the option to toggle between displaying the number of bikes or parking spots available on the marker.

During the design phase, we considered implementing a routing functionality that uses suitable bike stations as checkpoints to route between two locations. However, we were not able to implement this feature in the final version of the web application. In the mockup, the journey to the bike station was shown in purple, while the route between the stations was shown in red.

3.6 Design Process

We followed the basic structure of our mockup website. Once the overall design had started, we settled on using a CSS grid layout. The divs that we used were arranged in the diagram shown below.

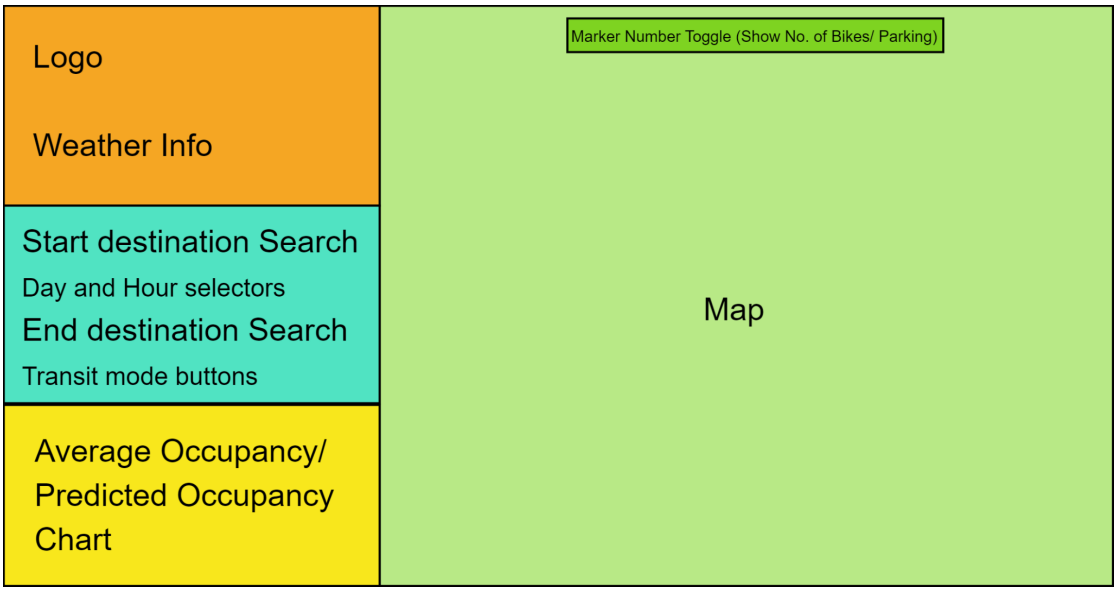


Figure 4: Layout of the div elements in our website.

The final product remained faithful to the design choices, as depicted in the screenshot below.

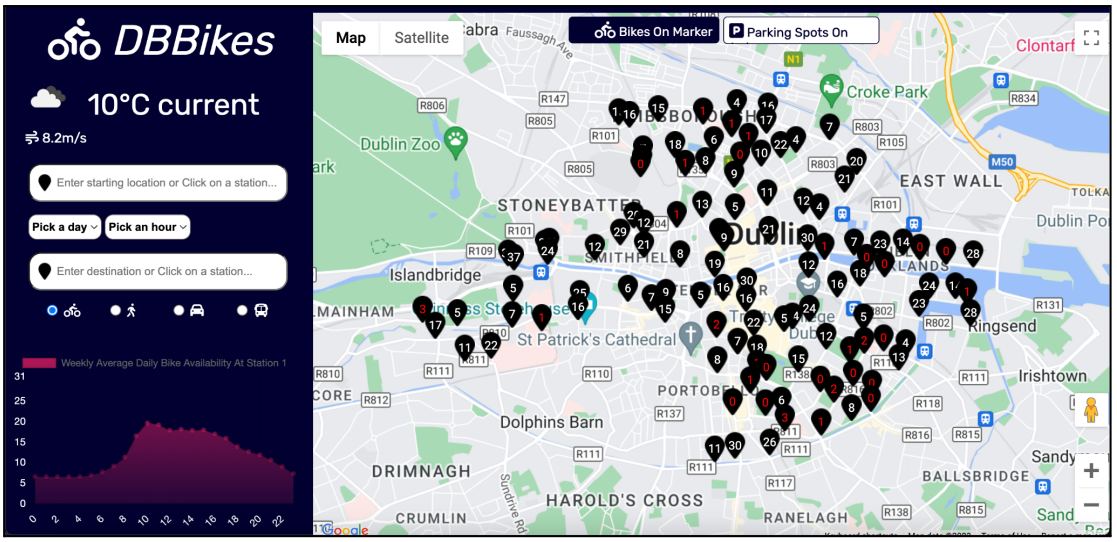


Figure 5: The end result.

4. Data Analytics

4.1 Overview

To predict the number of bikes available at a particular bike station at a given date and time we used a function with the following signature:

Prediction = f (date, time, station id, weather (time, location))[1]

Thus, the prediction is based on the station ID, the selected date, time, and the weather at that time.

Meanwhile, the training data can be described using the following equation:

Training data = occupancy, availability, weather (rain, etc.) from scrapers[2]

This means that the availability and occupancy of bikes at a specific station at various times and dates were among the historical data used to train the model. The weather conditions at the station, such as the amount of rain, the temperature, the wind speed, etc., were also included in the training data. The

[1] Lawlor, Aonghus. “Model Development.” COMP30830. Class Lecture at University College Dublin, Dublin, 27 March 2023.
[2] Ibid.

historical data was obtained using web scrapers which gathered the data from JCDecaux and OpenWeatherMap, and stored this in the RDS database.

During the training phase, the model was trained to determine the connection between the input features (date, time, station ID, and weather conditions) and the output variable (available number of bikes). Upon learning the relationship between the input and output variables, the model was then implemented to make predictions on the number of bikes available in the future at a specific station given the weather forecast.

4.2 Model Building

4.2.1 Data Collection

To build the model, we collected over two months' worth of data between 07 February 2023 and 18 April 2023 on bike availability, equating to almost 1.5 million rows of data. Meanwhile, a total of 2186 rows of data were gathered between 03 March 2023 and 18 April 2023 on the weather conditions in Dublin. These are displayed on two CSV files which were fed into the code to train the machine-learning model (please refer to the 'predictive_model' directory on our GitHub repository).

```
station_data
```

	Unnamed: 0	number	available_bikes	available_bike_stands	last_update
0	0	42	19	11	2023-03-03 21:36:02
1	1	30	9	11	2023-03-03 21:38:10
2	2	54	5	28	2023-03-03 21:37:09
3	3	108	12	23	2023-03-03 21:33:44
4	4	20	10	20	2023-03-03 21:33:42
...	...	...	...	...	...
1486337	1486337	39	0	20	2023-04-18 10:28:10
1486338	1486338	83	15	25	2023-04-18 10:34:23
1486339	1486339	92	0	40	2023-04-18 10:31:18
1486340	1486340	21	13	17	2023-04-18 10:31:02
1486341	1486341	88	7	23	2023-04-18 10:29:14

1486342 rows × 5 columns

Figure 6: Head and Tail of Station Dataset.

```
weather_data
```

	Unnamed: 0	time	temperature	windspeed	pressure	description	cloudiness
0	0	2023-03-03 21:38:52	5.20	2.06	1033.0	light intensity drizzle	75
1	1	2023-03-03 22:10:54	4.78	2.06	1033.0	broken clouds	75
2	2	2023-03-03 22:32:40	4.93	3.60	1033.0	broken clouds	75
3	3	2023-03-03 23:07:44	4.98	3.60	1033.0	broken clouds	75
4	4	2023-03-03 23:37:27	5.14	2.68	1033.0	broken clouds	75
...	...	...	...	...	...	...	...
2181	2181	2023-04-18 08:16:13	9.99	5.66	1030.0	broken clouds	75
2182	2182	2023-04-18 08:47:37	9.99	5.14	1030.0	broken clouds	75
2183	2183	2023-04-18 09:19:01	9.99	5.66	1030.0	scattered clouds	40
2184	2184	2023-04-18 09:40:08	9.99	5.66	1030.0	scattered clouds	40
2185	2185	2023-04-18 10:09:33	9.99	5.14	1030.0	scattered clouds	40

2186 rows × 7 columns

Figure 7: Head and Tail of Weather Dataset.

Before preprocessing the data, we cleaned our collected data by removing redundant columns. This allowed us to conduct a more suitable analysis.

```
weather_data = weather_data.drop(columns=['Unnamed: 0'])
station_data = station_data.drop(columns=['Unnamed: 0'])
```

Figure 8: Dropping redundant columns from datasets.

4.2.2 Preprocessing

The data was preprocessed by extracting essential details like the year, month, day, hour, and minute as well as whether the date is a weekday or not.

```
# Preprocess the data
weather_data['time'] = pd.to_datetime(weather_data['time'])
station_data['last_update'] = pd.to_datetime(station_data['last_update'])
```

Figure 9: Preprocessing of datasets.

```
# Extract day and hour
weather_data['year'] = weather_data['time'].dt.year
weather_data['month'] = weather_data['time'].dt.month
weather_data['day'] = weather_data['time'].dt.day
weather_data['hour'] = weather_data['time'].dt.hour
weather_data['minute'] = weather_data['time'].dt.minute
weather_data['is_weekday'] = ((weather_data['time'].dt.weekday >= 0) & (weather_data['time'].dt.weekday
<= 4)).astype(int)


station_data['year'] = station_data['last_update'].dt.year
station_data['month'] = station_data['last_update'].dt.month
station_data['day'] = station_data['last_update'].dt.day
station_data['hour'] = station_data['last_update'].dt.hour
station_data['minute'] = station_data['last_update'].dt.minute
```

Figure 10: Extracting essential details.

We also one-hot encoded the ‘description’ column by creating dummy variables, resulting in multiple new features.

```
# One-hot encode the 'description' column
weather_data = pd.get_dummies(weather_data, columns= ['description'])
```

Figure 11: One-hot encoding of the ‘description’ column.

```
weather_data
```

Python

	time	temperature	windspeed	pressure	cloudiness	year	month	day	hour	minute	...	description_light rain	description_light snow	description_mist	description_moderate rain	description_over cl
0	2023-03-03 21:38:52	5.20	2.06	1033.0	75	2023	3	3	21	38	...	0	0	0	0	
1	2023-03-03 22:10:54	4.78	2.06	1033.0	75	2023	3	3	22	10	...	0	0	0	0	
2	2023-03-03 22:32:40	4.93	3.60	1033.0	75	2023	3	3	22	32	...	0	0	0	0	
3	2023-03-03 23:07:44	4.98	3.60	1033.0	75	2023	3	3	23	7	...	0	0	0	0	
4	2023-03-03 23:37:27	5.14	2.68	1033.0	75	2023	3	3	23	37	...	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
2181	2023-04-18 08:16:13	9.99	5.66	1030.0	75	2023	4	18	8	16	...	0	0	0	0	
2182	2023-04-18 08:47:37	9.99	5.14	1030.0	75	2023	4	18	8	47	...	0	0	0	0	
2183	2023-04-18 09:19:01	9.99	5.66	1030.0	40	2023	4	18	9	19	...	0	0	0	0	
2184	2023-04-18 09:40:08	9.99	5.66	1030.0	40	2023	4	18	9	40	...	0	0	0	0	
2185	2023-04-18 10:09:33	9.99	5.14	1030.0	40	2023	4	18	10	9	...	0	0	0	0	

2186 rows × 29 columns

Figure 12: Result of one-hot encoding ‘description’ column.

The two datasets were then merged on the date and time columns and redundant columns were removed.

```
# Merge the datasets
merged_data = pd.merge(station_data, weather_data, on = ['year','month','day','hour'])
```

Python

```
# Drop every redundant columns
merged_data = merged_data.drop(columns=['last_update', 'time'])
```

Python

```
merged_data.head(10)
```

Python

	number	available_bikes	available_bike_stands	year	month	day	hour	minute_x	temperature	windspeed	...	description_light rain	description_light snow	description_mist	description_moderate rain	d
0	42	19	11	2023	3	3	21	36	5.2	2.06	...	0	0	0	0	
1	30	9	11	2023	3	3	21	38	5.2	2.06	...	0	0	0	0	
2	54	5	28	2023	3	3	21	37	5.2	2.06	...	0	0	0	0	
3	108	12	23	2023	3	3	21	33	5.2	2.06	...	0	0	0	0	
4	20	10	20	2023	3	3	21	33	5.2	2.06	...	0	0	0	0	
5	56	7	33	2023	3	3	21	33	5.2	2.06	...	0	0	0	0	
6	6	4	16	2023	3	3	21	36	5.2	2.06	...	0	0	0	0	
7	18	15	15	2023	3	3	21	36	5.2	2.06	...	0	0	0	0	
8	32	5	25	2023	3	3	21	34	5.2	2.06	...	0	0	0	0	
9	52	2	30	2023	3	3	21	36	5.2	2.06	...	0	0	0	0	

10 rows × 32 columns

Figure 13: Merging datasets and dropping redundant columns.

New features were then added to the merged dataset. These new features include useful information such as whether it is a busy hour (i.e. between hours 7-10 inclusive or 16-19 inclusive) or whether the weather is windy or cold.

```
# Engineer is_busy_hours feature
merged_data['is_busy_hours'] = ((merged_data['hour'] >= 7) & (merged_data['hour'] <= 10)) | ((merged_data['hour'] >= 16) & (merged_data['hour'] <= 19)).astype(int)


merged_data['cold_weather'] = (merged_data['temperature'] < 5).astype(float)
merged_data['windy_weather'] = (merged_data['windspeed'] > 8).astype(float)
```

Figure 14: Adding new features, derived from datasets.

4.2.3 Model Training and Evaluation

Before training the model, we created a list of unique station IDs from the merged dataset and sorted this in ascending order.

The merged dataset then underwent machine learning for each distinct station using a for loop. The data was split into two sets: the training set and the testing set using the *train_test_split* function from the *sklearn.model_selection* library. This function randomly divides the data into training and testing sets according to a specified test size (in this case, 0.2, which means that 20% of the data was used for testing and 80% for training) and a random seed value (in this case, 42).

```
unique_stations = list(merged_data['number'].unique())
unique_stations.sort()


for station_id in unique_stations:
    station_data = merged_data[merged_data['number'] == station_id]

    # Split the data into training and testing sets
    X = station_data.drop(columns=['available_bikes', 'available_bike_stands'])
    y = station_data[['available_bikes', 'available_bike_stands']]

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Train a machine learning model
    model = LinearRegression()
    model.fit(X_train, y_train)

    # Serialize the trained model into a file called model.pkl
    with open(f'model_{station_id}.pkl', 'wb') as handle:
        pickle.dump(model, handle, pickle.HIGHEST_PROTOCOL)

    # Deserialize the model.pkl file into an object called model
    with open(f'model_{station_id}.pkl', 'rb') as handle:
        model = pickle.load(handle)

    # Evaluate the model
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    print(f"Mean Squared Error for station {station_id}:", mse)
```

Figure 15: Merged dataset undergoing machine learning for each unique station ID.

A linear regression model was trained on the training data, which was serialised to a Pickle file, before being deserialised back into an object, and tested on the testing data using the mean squared error (MSE) evaluation measure. The MSE value was then outputted for each station.

```
Mean Squared Error for station 1: 62.54825611130398
Mean Squared Error for station 2: 15.885984311983869
Mean Squared Error for station 3: 22.636853810507105
Mean Squared Error for station 4: 31.582817115094706
Mean Squared Error for station 5: 158.77461102573955
Mean Squared Error for station 6: 28.866973965900904
Mean Squared Error for station 7: 38.728416974745535
Mean Squared Error for station 8: 51.08759124087591
Mean Squared Error for station 9: 43.25511657936523
Mean Squared Error for station 10: 19.36393752923111
Mean Squared Error for station 11: 58.39016588246591
Mean Squared Error for station 12: 19.087842987804876
Mean Squared Error for station 13: 34.221459951081584
Mean Squared Error for station 14: 53.859275974649506
Mean Squared Error for station 15: 6.69305227107231
Mean Squared Error for station 16: 27.786906363550884
Mean Squared Error for station 17: 19.592180757440758
Mean Squared Error for station 18: 46.21239259628128
Mean Squared Error for station 19: 80.42282342489435
Mean Squared Error for station 20: 46.606751237748085
Mean Squared Error for station 21: 63.59785136066307
Mean Squared Error for station 22: 27.193116709767523
Mean Squared Error for station 23: 64.06368853246255
Mean Squared Error for station 24: 30.056437283903186
Mean Squared Error for station 25: 36.51561129840535
...
Mean Squared Error for station 114: 136.00390963378072
Mean Squared Error for station 115: 71.86384014307663
Mean Squared Error for station 116: 3.3386705349642094
Mean Squared Error for station 117: 73.85529170220425
```

Figure 16: Preview of MSE values for each station.

The average MSE value for a given station was 55.5 while the standard deviation was 34.5. Even though this large spread is undesirable, some stations saw frequent use while others were mostly static in their occupancy rate. Thus, this was not totally unexpected either.

We performed a logistic regression model as a comparison, but the results were disappointing. The average MSE was 113.0 with a standard deviation of 82.0. As we were not dealing with a classification problem, we expected worse results. However, this comparison confirmed that our decision to use the linear regression model was justified.

We considered using a random forest model to train our predictive model, but we decided against it due to the large number of parameters the model would require. We were concerned about the size of the model and the limited space available on our EC2 instance. Our concerns proved to be valid, as we encountered space issues when attempting to deploy the model on our EC2 instance.

4.2.4 Prediction

Before predicting the number of bikes and bike stands available, we first created a function to retrieve the weather forecast for a given future date and time by making an API call to the OpenWeatherMap API, specifying the longitude and latitude of Dublin and metric as the unit of measurement. After receiving the API response, the returned data was examined to obtain the forecast for the time that is the closest to the specified date and time. The pertinent weather data was then taken out of the forecast and stored in the dictionary *weather_data*.

```
def get_future_weather_data(day_from_today, hour):

    #get today's date
    today = datetime.date.today()
    date_time = datetime.datetime(today.year, today.month, today.day + day_from_today, hour)
    time_unix_timestamp = int(time.mktime(date_time.timetuple()))
    api_call = f"https://api.openweathermap.org/data/2.5/forecast?lat=53.35&lon=-6.26&appid=b7d6a55bc0fff59fb0d5f7c3c1668417&units=metric"
    forecast_info = requests.get(api_call)
    forecast_info_json = json.loads(forecast_info.text)
    #from the forecast, we need to find the time that is the closest to the time we input
    index = 0
    closest_date = abs(forecast_info_json.get('list')[0].get('dt') - time_unix_timestamp)
    for i in range(40):
        if abs(forecast_info_json.get('list')[i].get('dt') - time_unix_timestamp) < closest_date:
            closest_date = abs(forecast_info_json.get('list')[i].get('dt') - time_unix_timestamp)
            index = i
    weather_info = forecast_info_json.get('list')[index]

    weather_data = {}
    weather_data['temperature'] = weather_info.get('main').get('temp')
    weather_data['windspeed'] = weather_info.get('wind').get('speed')
    weather_data['pressure'] = weather_info.get('main').get('pressure')
    weather_data['cloudiness'] = weather_info.get('clouds').get('all')
    weather_data['description'] = weather_info.get('weather')[0].get('description')


#     # Convert the dictionary to a DataFrame
    future_weather_data = pd.DataFrame(weather_data, index=[0])

#     # One-hot encode the 'description' column
    future_weather_data = pd.get_dummies(future_weather_data, columns=['description'])

    return future_weather_data
```

Figure 17: Code for retrieving future weather data.

The *pd.DataFrame* function was used to transform this dictionary into a Pandas DataFrame. The *pd.get_dummies* function was used to one-hot encode the ‘description’ column in the DataFrame before returning the DataFrame containing the weather data for the requested date and time.

To predict the available number of bikes and bike stands at a particular station at a given time and day in the future, we implemented the equation mentioned above by creating the function *predict_for_future_date*, which takes in a station ID and a future date and time as parameters. To perform its prediction, the function carries out the following:

1. First, it loads a machine-learning model that has already been trained for the given station.
2. To retrieve the weather forecast for the specified future date and time, it calls the *get_future_weather_data()* function.

3. The weather information is preprocessed and an input *DataFrame* containing the weather data and other necessary features is generated.
4. It makes sure that the columns in the input DataFrame match those in the training set and are arranged correctly.
5. Using the loaded model and the input DataFrame, it makes the prediction.
6. It checks that the predicted number of bicycles and bike stands is within the acceptable range (i.e. 0 or more bicycles and 0 or more bike stands).
7. The predicted number of available bikes and available bike stands are returned as integers.

```python
def predict_for_future_date(station_id, days_from_today, hour):
    # Load the model
    with open(f'model_{station_id}.pkl', 'rb') as handle:
        model = pickle.load(handle)

    # Get future weather data
    future_weather_data = get_future_weather_data(days_from_today, hour)
    today_date = datetime.date.today()

    # Preprocess future_weather_data and make the prediction
    future_date = pd.to_datetime(today_date)
    day = future_date.day + days_from_today
    hour = hour
    year = future_date.year
    month = future_date.month
    is_weekday = int((future_date.weekday() >= 0) & (future_date.weekday() <= 4))
    is_busy_hours = int(((hour >= 7) & (hour <= 10)) | ((hour >= 16) & (hour <= 19)))
    cold_weather = (future_weather_data['temperature'][0] < 5).astype(float)
    windy_weather = (future_weather_data['windspeed'][0] > 8).astype(float)

    # Create input DataFrame
    input_df = future_weather_data.copy()
    input_df['year'] = year
    input_df['month'] = month
    input_df['day'] = day
    input_df['hour'] = hour
    input_df['number'] = station_id
    input_df['is_weekday'] = is_weekday
    input_df['is_busy_hours'] = is_busy_hours
    input_df['cold_weather'] = cold_weather
    input_df['windy_weather'] = windy_weather

    # Make sure the input DataFrame has the same columns as the training data
    for col in X.columns:
        if col not in input_df.columns:
            input_df[col] = 0

    # Reorder the columns to match the training data
    input_df = input_df[X.columns]

    # Make the prediction
    prediction = model.predict(input_df)

    available_bikes, available_bike_stands = prediction[0]

    # Ensure available_bikes is greater than 0
    available_bikes = max(0, available_bikes)

    # Ensure available_bike_stands is greater than 0 and less than or equal to 114
    available_bike_stands = max(0, min(available_bike_stands, 114))

    return int(available_bikes), int(available_bike_stands)
```

Figure 18: Code for predicting bike availability for future dates.

In summary, this function uses machine learning and weather data to predict the number of bikes and bike stands available at a given station for a particular date and time in the future.

We also implemented a for loop to predict the number of bikes available on an hourly basis for the upcoming 24 hours at a given station number.

In each iteration, the above-mentioned function *predict_for_future_date* is called with *station_number*, 1, and *i* as arguments, whereby 1 indicates 'one day from today' and *i* indicates the hour of the day for which the prediction is being made.

The returned prediction includes two values: *available_bikes* and *available_bike_stands*. The *available_bike_stands* represents the anticipated number of empty stands at the station at the specified hour, while *available_bikes* is the anticipated number of bikes available at the station at the specified hour. In actuality, we simply computed the expected number of parking spots at the station by subtracting the predicted number of bikes from the total number of stands. This approach ensured that the sum of bikes and parking spots remained constant for each station.

The list *no_of_bikes_2* is updated to include the expected *available_bikes* value. The total_stands field also contains the total of the *available_bikes* and *available_bike_stands* variables.

```
no_of_bikes_2 = []
station_number = 43
total_stands = 0
for i in range(24):
    data = predict_for_future_date(station_number, 1, i)
    no_of_bikes_2.append(data[0])
    total_stands = data[0] + data[1]
```

Figure 19: Code to retrieve prediction for bike availability every hour for 24 hours.

4.3 Combining Model with Flask

Before integrating the machine learning model with the frontend, it was first combined with the Flask web framework. The Flask application contains two routes that are relevant to the machine learning model: */predicted_occupancy* and */station_avg_data*.

The */predicted_occupancy* route calls the *predicted_station_occupancy* function which takes in two integer parameters: *station_id* and

days_from_today. The function *predict_for_future_date* is used to predict the occupancy of a given station indicated by *station_id* on a future date or '*days_from_today*'.

The *predict_for_future_date* function loads the pre-trained machine learning model from the Pickle files. It preprocesses the input data (i.e. station ID, date and time, weather forecast) before using the loaded model to make predictions. The predictions are returned in a list.

The expected number of bikes available is then stored in a dictionary by the *predicted_station_occupancy* function, which invokes the function *predict_for_future_date* for every hour of the day. The dictionary is returned as a JSON object.

The *station_avg_data* function is called by the /*station_avg_data* route. To acquire availability information for a given station, it performs a SQL database query. The data is preprocessed through the removal of duplicates, setting the index to the *last_update* column, and resampling the data at 10-minute intervals. The average number of bikes available for each hour is then calculated after grouping the data by hour. The result is returned as a JSON object.

```
@app.route("/predicted_occupancy/<int:station_id>&<int:days_from_today>")
@functools.lru_cache(maxsize=128)
def predicted_station_occupancy(station_id, days_from_today):
    no_of_bikes = {}
    for i in range(24):
        mydata = predict_for_future_date(station_id, days_from_today, i)
        no_of_bikes[i] = mydata[0]
    print("Predicted occupancy ran")
    return no_of_bikes

@app.route("/station_avg_data/<int:station_id>")
@functools.lru_cache(maxsize=128)
def station_avg_data(station_id):
    engine = get_db()
    try:
        with engine.connect() as connection:
            df = pd.read_sql_query(f"select * from availability where number = {station_id}
            limit 2016;", connection)
            print("df head: ", df.head)
            connection.close()
            print(df.head())
            df = df.drop_duplicates(keep='first')
            df['last_update'] = pd.to_datetime(df['last_update'])
            df = df.set_index('last_update')
            df = df.resample('10T').mean()
            df = df.groupby(df.index.hour).mean()
            result = df['available_bikes'].to_json(orient="split")
            parsed = loads(result)
            return dumps(parsed, indent=4)
    except:
        print(traceback.format_exc())
        return "error in station availability", 404
```

Figure 20: Integrating Model with Flask web framework.

4.4 Integrating the Model with the Frontend

The charts themselves were displayed on the web application with the help of the Chart.js library in the JavaScript code.

```
DailyAvg = new Chart(ctx, {
type: 'line',
data: {
labels: timeList,
datasets: [{
    label: 'Weekly Average Daily Bike Availability At Station ' + station_id,
    labelColor: '#fff',
    data: dailyAvgOccupancy,
    fill: true,
    backgroundColor: gradient,
    }]
},
options: {
    tension: 0.1,


scales: {
    x: {
        ticks: {
            color: "#fff"
          },

    },
    y: {
        min: 0,
        max: stationStands ,
        ticks: {
            color: "#fff"
          }
    }
}
```

Figure 21: Implementing chart of Weekly Average Daily Bike Availability at a given station using Chart.js.

```
let selectElement = document.getElementById('pick-a-day')
let chosenDay = selectElement.options[selectElement.selectedIndex].innerHTML;
DailyAvg = new Chart(ctx, {
type: 'line',
data: {
labels: timeList,
datasets: [{
    label: 'Expected Daily Occupancy at Station ' + station_id + ' on ' + chosenDay,
    labelColor: '#fff',
    data: dailyAvgOccupancy,
    fill: true,
    backgroundColor: gradient,
    }]
},
options: {
    tension: 0.1,


scales: {
    x: {
        ticks: {
            color: "#fff"
          },

    },
    y: {
        min: 0,
        max: stationStands + 5,
        ticks: {
            color: "#fff"
          }
    }
}
```

Figure 22: Implementing chart of Expected Daily Occupancy at a given station on a specified day using Chart.js.

The charts are displayed on the web application as shown below at the bottom left corner of the webpage:

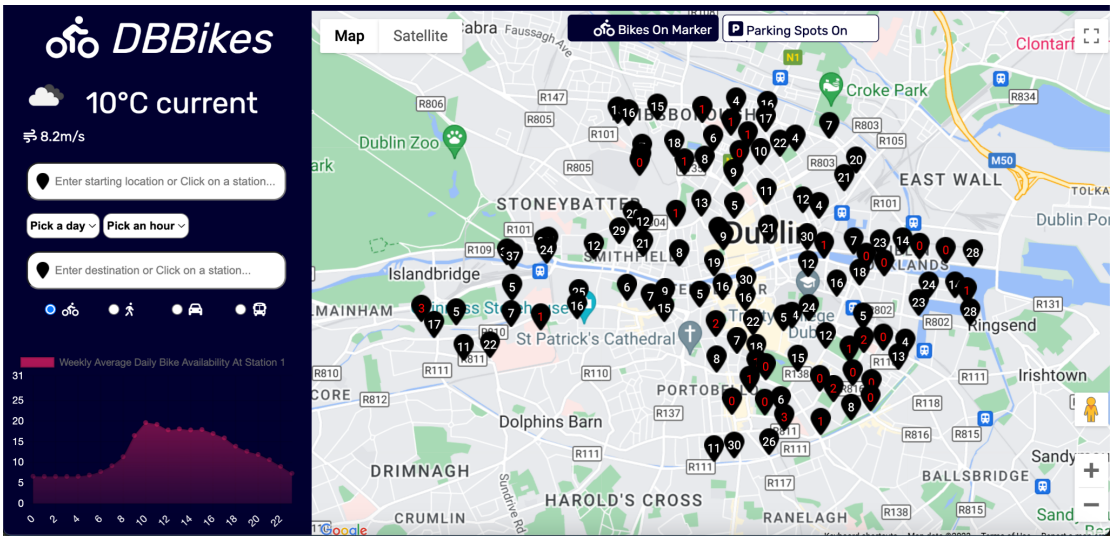


Figure 23: Web application showing the Weekly Average Daily Bike Availability at Station 1 (default station is Station 1).

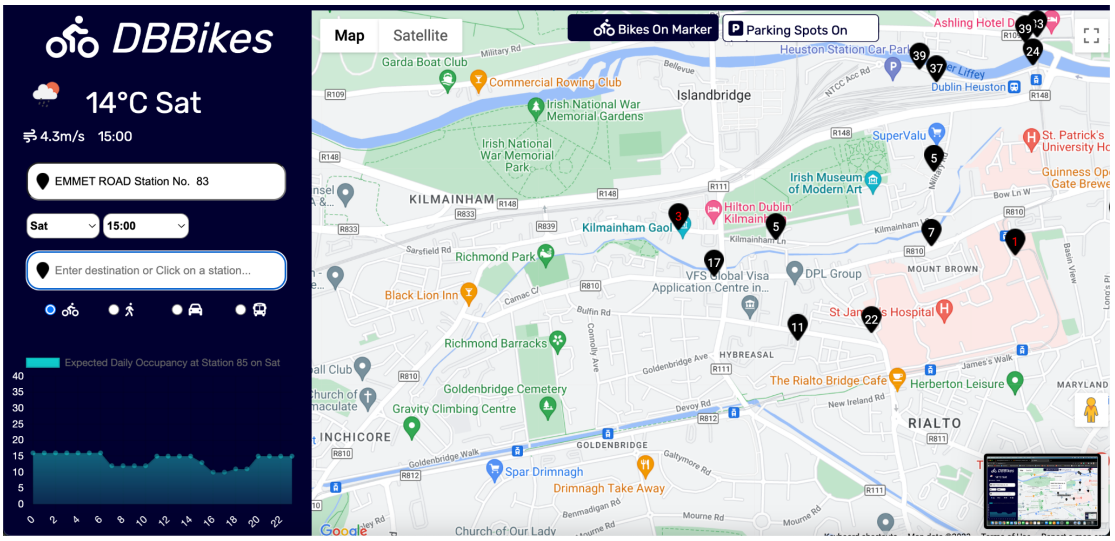


Figure 24: Web application showing the Expected Occupancy at Station 85 on a Saturday.

4.5 Integrating the Model with the Backend

To integrate the model with the backend, we created three tables, of which only two were used to support the prediction program: one for storing the real-time availability data for all bike stations, and one for storing the current weather information.

To do this, we created two Python scripts that perform web scraping: one to scrape data from Dublin Bikes bike-sharing system/JCDecaux API every 5

minutes and another to scrape weather information from the OpenWeatherMap API every 30 minutes. The scraped data was then stored in a MySQL database hosted on Amazon RDS. To ensure that the data was up-to-date, these scripts were run on the EC2 instance as well.

```
def availability_to_db(text):
    stations = json.loads(text)
    for station in stations:
        vals = (int(station.get('number')),
                int(station.get('available_bikes')),
                int(station.get('available_bike_stands')),
                str(datetime.datetime.fromtimestamp(int(str(station.get('last_update'))
                [0:10])))
                )
        connection.execute("insert into availability values(%s,%s,%s,%s);", vals)
    return


bike_api_key = 'a471198f1d4a279171da8f17892b64eb12c32f33'
bike_api_query = f'https://api.jcdecaux.com/vls/v1/stations?contract=dublin&apiKey=
{bike_api_key}'



def main():
    while True:
        try:
            connection = engine.connect()
            r = requests.get(bike_api_query)
            station_to_db(r.text)
            availability_to_db(r.text)
            print("Scraping is done, now waiting...")
            connection.close()
            time.sleep(5*60) #Scrape every 5 minutes
        except:
            print("Error. Something went wrong.")
    return

main()
```

Figure 25: Python script for performing web scraping from JCDeccaux API.

```
def weather_to_db(text):
    sql = """
    CREATE TABLE IF NOT EXISTS weather (
    time DATETIME,
    temperature FLOAT(3,2),
    windspeed FLOAT(4,2),
    pressure FLOAT(5,1),
    description VARCHAR(256),
    cloudiness VARCHAR(256)
    );
    """

    res = connection.execute(sql)


    weather_infos = json.loads(text)
    vals = (str(datetime.datetime.fromtimestamp(int(weather_infos.get('dt')))),
            float(weather_infos.get('main').get('temp')),
            float(weather_infos.get('wind').get('speed')),
            float(weather_infos.get('main').get('pressure')),
            weather_infos.get('weather')[0].get('description'),
            weather_infos.get('clouds').get('all'))
    connection.execute("insert into weather values(%s,%s,%s,%s,%s,%s);", vals)
    return

weather_api_key = 'b7d6a55bc0fff59fb0d5f7c3c1668417'
lat='53.35'
lon='-6.26'
weather_api_query = f"https://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}&appid={weather_api_key}&units=metric"

def main():
    while True:
        try:
            connection = engine.connect()
            weather_r = requests.get(weather_api_query)
            weather_to_db(weather_r.text)
            print("Weather scraping is done, now waiting...")
            connection.close()
            time.sleep(30*60) #Scrape every 30 minutes
        except:
            print("Error. Something went wrong.")
    return

main()
```

Figure 26: Python script for performing web scraping from OpenWeatherMap API.

We ended up collecting almost 1.5 million rows of data for the *stations* table and 2186 rows for the *weather* table. The bike data collection was done at a smaller time interval because the bike availability at each station is likely to change quickly, so gathering the data more frequently provided more real-time information for users who rely on the data to find available bikes. In contrast, weather conditions are less likely to change frequently and were thereby not updated as regularly.

station_historic_availability_data

	number	available_bikes	available_bike_stands	last_update
0	42	19	11	2023-03-03 21:36:02
1	30	9	11	2023-03-03 21:38:10
2	54	5	28	2023-03-03 21:37:09
3	108	12	23	2023-03-03 21:33:44
4	20	10	20	2023-03-03 21:33:42
5	56	7	33	2023-03-03 21:33:44
6	6	4	16	2023-03-03 21:36:55
7	18	15	15	2023-03-03 21:36:44
8	32	5	25	2023-03-03 21:34:37
9	52	2	30	2023-03-03 21:36:35
10	48	13	27	2023-03-03 21:37:25

Figure 27: Head of historic Station dataset.

weather_historic_data

	time	temperature	windspeed	pressure	description	cloudiness
0	2023-03-03 21:38:52	5.2	2.06	1033.0	light intensity drizzle	75
1	2023-03-03 22:10:54	4.78	2.06	1033.0	broken clouds	75
2	2023-03-03 22:32:40	4.93	3.6	1033.0	broken clouds	75
3	2023-03-03 23:07:44	4.98	3.6	1033.0	broken clouds	75
4	2023-03-03 23:37:27	5.14	2.68	1033.0	broken clouds	75
5	2023-03-04 00:09:02	4.69	3.09	1033.0	broken clouds	75
6	2023-03-04 00:41:10	4.52	1.54	1033.0	broken clouds	75
7	2023-03-04 01:08:36	5.27	2.57	1033.0	broken clouds	75
8	2023-03-04 01:34:25	5.34	3.09	1032.0	broken clouds	75
9	2023-03-04 02:03:52	5.42	2.68	1033.0	broken clouds	75
10	2023-03-04 02:41:11	5.29	3.09	1032.0	broken clouds	75

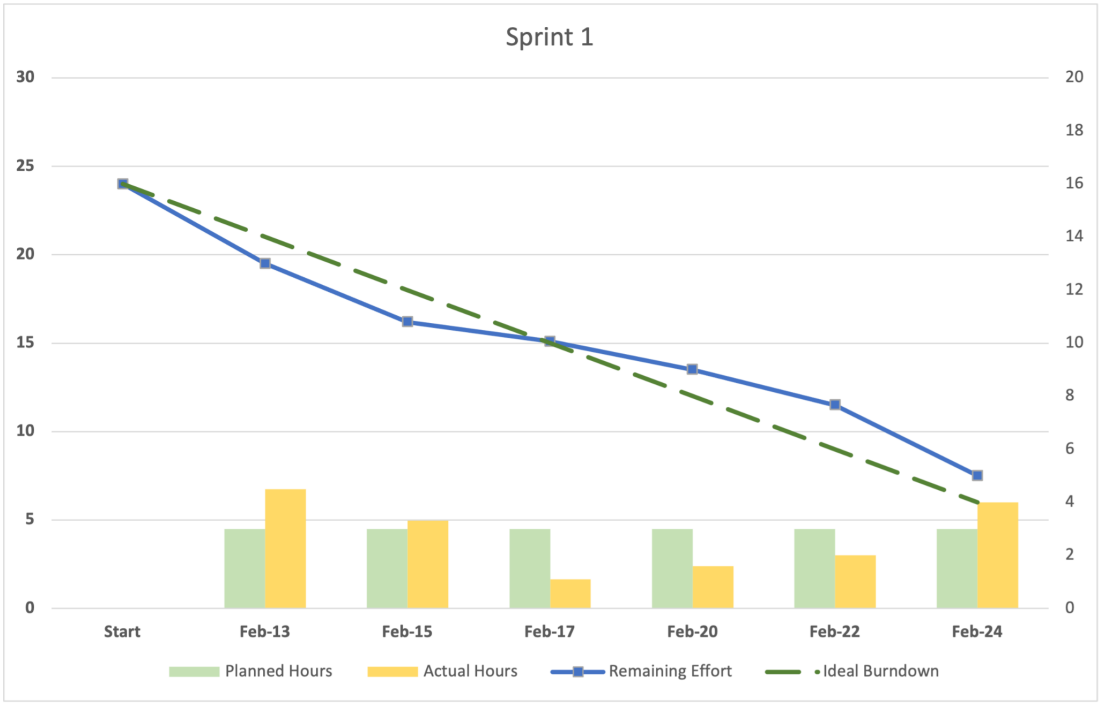
Figure 28: Head of the historic Weather dataset.

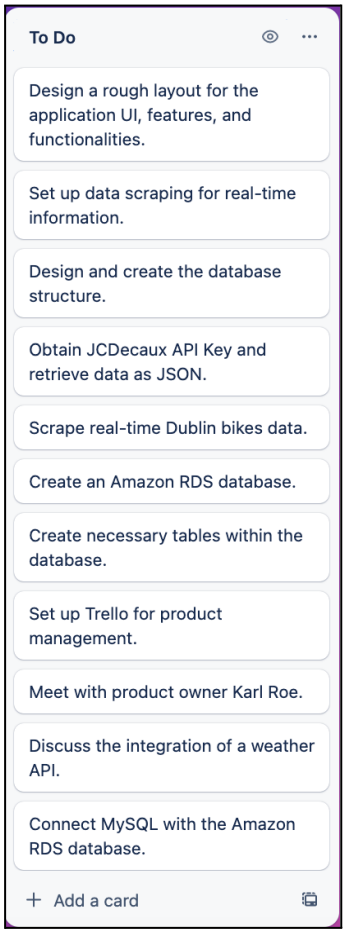
4.6 Improvements

Although the current model is performing adequately, its performance could be further improved by incorporating a more extensive dataset spanning a period of 12 months, as this would offer a more complete perspective. Using an alternative model like a random forest could also result in better outcomes, provided there are no memory constraints.

5. Process

5.1 Sprint 1 (13.02.2023 – 24.02.2023)





In **Sprint 1**, our team accomplished several tasks that established a strong foundation for the project. We created a UI layout, obtained the JCDecaux API Key, and scraped real-time Dublin Bikes data. This allowed us to set up an effective RDS database. Additionally, we established a product management system using Trello and met with our product owner Karl Roe to discuss integrating a weather API.

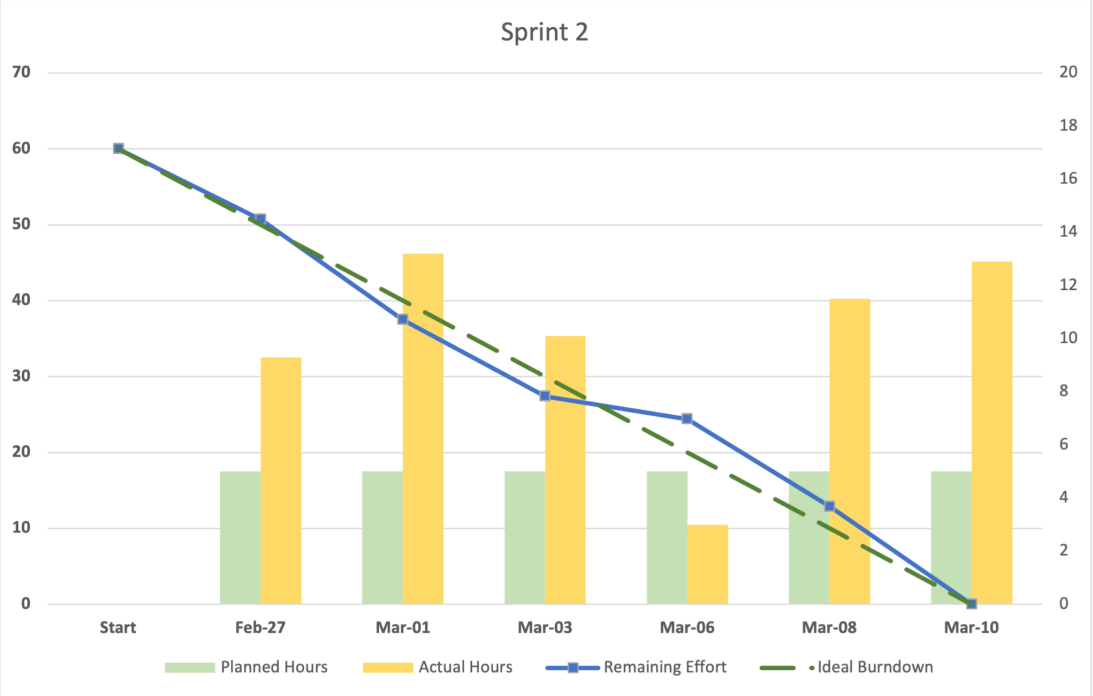
Effective communication and task management were key factors in our team's success in staying on track. We planned to maintain this approach by continuing to utilise Trello and scheduling regular meetings with our product owner. However, our approach to API integration and data management could have been improved, as these tasks were intricate and required a significant amount of time.

Some of the challenges we faced include data scraping and RDS database setup. To overcome these obstacles, we conducted extensive research

to find the best practices and sought advice from experienced developers. We realised the importance of allocating enough time for complex tasks such as connecting MySQL with Amazon RDS, which required more effort than we anticipated.

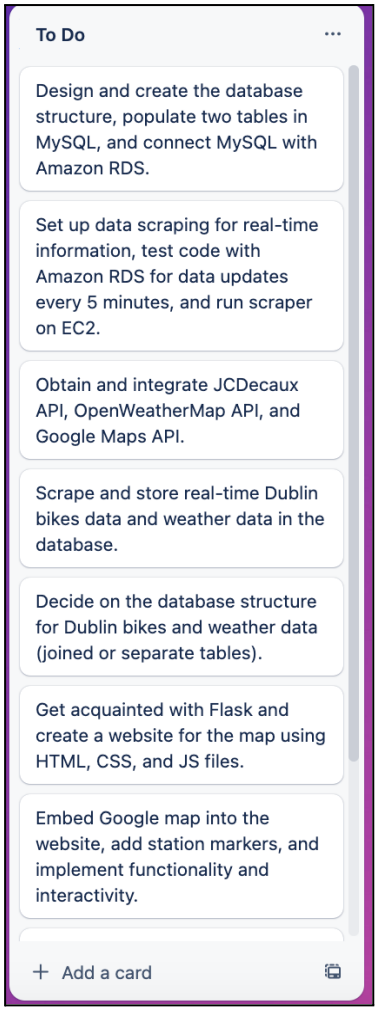
For the upcoming sprint, we planned to create tables for the RDS database and connect MySQL to the Amazon RDS database. We also aimed to optimise our workflow and make necessary adjustments to tackle any obstacles that might arise.

5.2 Sprint 2 (27.02.2023 – 10.03.2023)



During **Sprint 2**, our team made significant progress. We populated two tables in the MySQL database, connected it with Amazon RDS, fixed timestamp issues, and tested the code for updating data every five minutes. The scraper ran smoothly on EC2, and the weather data was also successfully scraped and updated every five minutes.

During our meetings, we discussed our individual work progress and decided to familiarise ourselves with Flask for handling the weather data. We also retrieved the Google Maps API key, created a map website, embedded the map, and designed a template for info boxes. We added station markers on the map and worked on improving their appearance and functionality.

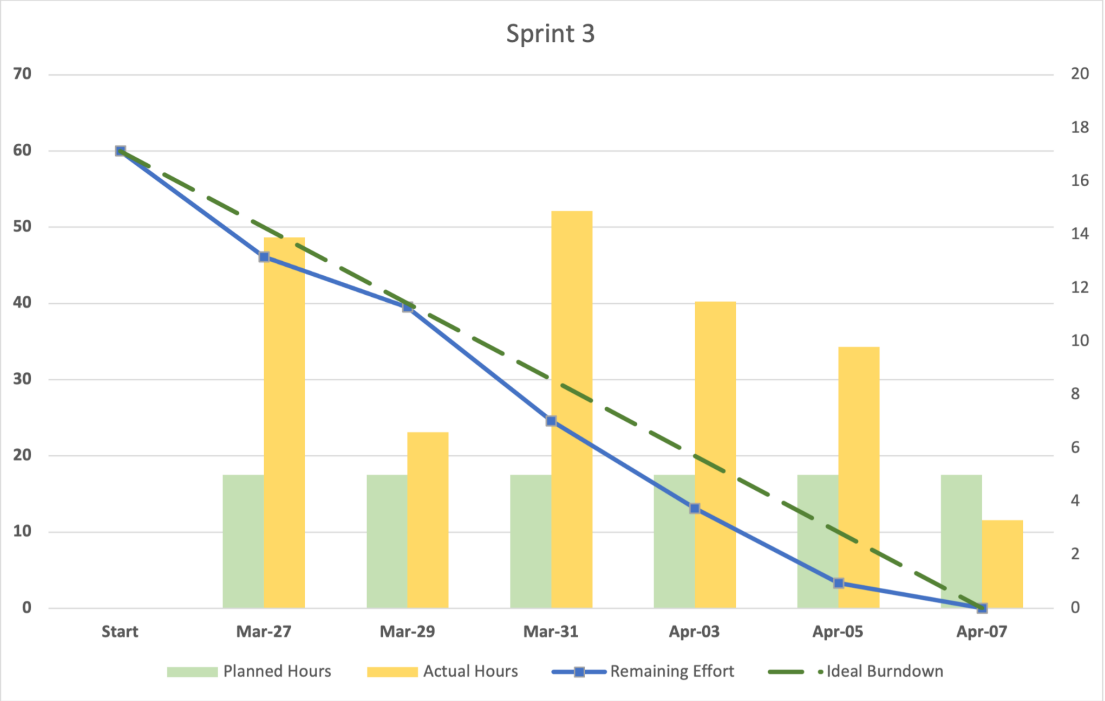


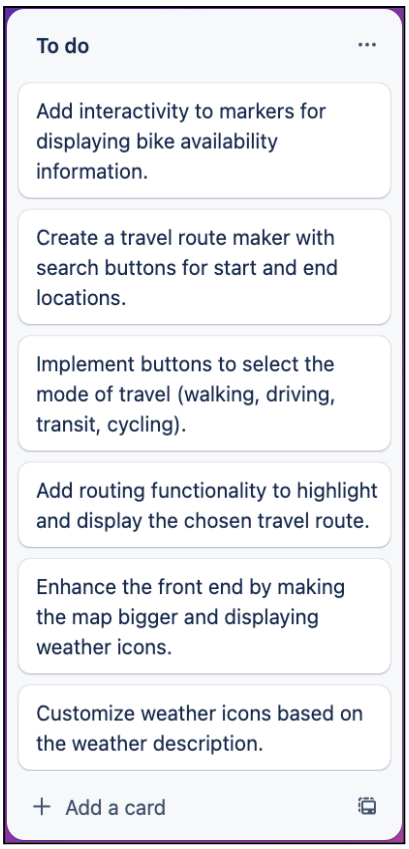
Our team successfully adapted to new technologies like Flask and Google Maps API. We made significant progress in building the core functionalities of the application. However, we had a slight drawback in handling and storing scraped data, deciding on table structures, and ensuring marker reliability.

To overcome these issues, we could have dedicated more time to researching optimal data handling and storage practices and developing a clear plan for managing weather data. Our decision on using one joined table or two separate tables for Dublin bikes and weather data ended up impacting our API design and data storage efficiency. We later discovered that omitting the 'iconid' code from the weather API would complicate visually representing the weather information on our website.

For the upcoming sprint, we planned to focus on adding functionality and interactivity to markers, improving their visuals, showing availability on the map, and connecting the Flask app with the database.

5.3 Sprint 3 (27.03.2023 – 07.04.2023)





During **Sprint 3**, our team continued to make significant progress by enhancing the map's functionality and interactivity. We added marker interactivity by displaying information on bike availability upon clicking. We created a travel route maker with search buttons and different modes of travel options (like walking, driving, public transit, and cycling). The map now displays the chosen route.

To improve the user experience, we aimed to increase the map size and display weather icons based on descriptions. We planned to create two charts using Chart.js for daily and hourly data but decided to have a single chart that changes based on user selection. For machine learning, we explored multiple options for predicting bike availability in all stations, eventually settling on a Python linear regression model.

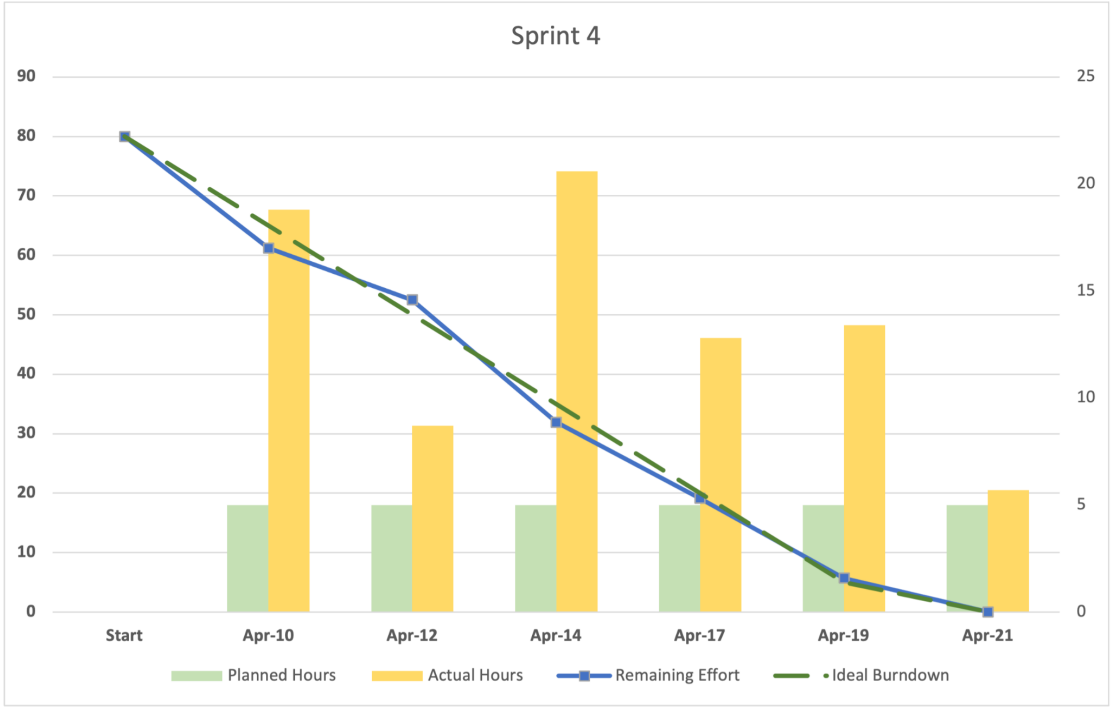
Our application's functionality significantly improved with the successful deployment of a predictive machine learning model. It estimates the number

of available bikes and bike stands by utilising input weather information and bike station data, providing users with valuable insights.

Sprint 3 saw successful enhancements to the user experience and the integration of machine learning, which were major accomplishments. However, challenges arose in managing large amounts of data and ensuring the accuracy of our predictive model. Efforts to optimise response times also caused unexpected issues, especially in the later stages of this project.

For the upcoming sprint, our goals were to improve the machine learning model, incorporate filtering options into the route maker, and tackle any data management or visualisation issues. Because we were approaching the project's conclusion, we needed to shift our focus towards the application's frontend development.

5.4 Sprint 4 (10.04.2023 – 21.04.2023)



In **Sprint 4**, our team tackled deployment challenges and improved the application's functionality. We aimed to resolve git issues and deploy the project on EC2 using AWS CodeDeploy or other standard web servers. We also explored adding geolocation features and different marker colours to indicate bike availability.

We faced difficulties in deploying the project on AWS throughout Sprint 4, as our EC2 instance kept on crashing. However, we persisted and managed to

overcome these obstacles and deploy our project on AWS. Although we encountered some issues, such as the predictive model not loading properly due to an internal server error, we managed to run the project on port 80 as a privileged user. Running the application in this way enabled us to debug more efficiently than using Gunicorn and NginX.

To address the deployment issues, we focused on error handling by examining the JavaScript file, installing necessary modules in sudo, and looking at the Flask error logs for internal server errors. We also considered using containerisation with Docker as an alternative deployment method. During this sprint, we managed to deploy the project on AWS successfully and planned to take down our EC2 instance after the application review. We would not have been able to identify and resolve the deployment challenges without the guidance and assistance of our lecturer and product owner, as we lacked the necessary expertise.

Despite facing challenges with AWS and EC2, our perseverance allowed us to overcome the deployment hurdles and successfully enhance the application's features in Sprint 4.

To ensure that we maximise our learnings and optimise our workflow, it is crucial to reflect on both the successes and challenges we experienced throughout the project now that our application is complete. Our team excelled in communication, learning new technologies, and overcoming obstacles that we encountered along the way. As we move forward, we can apply the lessons learned from this project to streamline our workflow and successfully tackle other software development challenges. The experience we gained from this project will undoubtedly prove invaluable in our future endeavours.

6. Meetups

The majority of our meetings were held during the tutorial sessions, which provided us with the opportunity to engage with our product owner. For a comprehensive log of our meetings, please refer to the Appendix (Section 8). During each meeting, we took detailed notes that included the date, time, and attendees. We also documented completed tasks and tasks that needed to be completed by specific deadlines, usually within a week or by the end of the Sprint. An example meeting log is shown below.

02/03/23 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi Karl Roe	- Retrieved Weather API from OpenWeatherMap - Managed to scrape/update current weather data every five minutes - Managed to store scraped data in the database	Decide on having one joined or two separate tables for Dublin bikes and weather data - Option 1: API for bikes and API for weather - Scrapers in EC2 can fetch data from RDS - Option 2: optimiser = another table, will look at combined data - Will look at the most recent data on both bike and weather	Report: - Keep records up to date - Structure: should be divided into Sprints Weather data: - Temperature (!!!) - Wind speed (!!) - Pressure (!) - Description (?) API calls: - Every 5 minutes should suffice Sprint 2 - Have a Flask app - Connect the app with DB - Do not need to run Flask thru EC2 until Sprint 4

Figure 29: An example meeting log from 02/03/23.

7. Future Work

Due to time constraints, we were unable to implement media queries to our CSS code so that the website would be responsive and work on all screen resolutions. This is something that can be improved upon in the future.

As previously stated in Section 3, the current routing feature does not include the use of checkpoints. An optimal routing process would involve directing the user to the nearest bike station with available bikes before proceeding with a bike route to the closest station near the final destination, which also has available bike parking spots. It would be beneficial to include an estimated travel time for each transit option when displaying a route to the user. The routing feature should also provide directions from the starting bike station to the user's final destination on the map.

The predictive model's accuracy is limited because the scraped data covers only the last 2 months, making it difficult to capture availability patterns during non-winter months. To improve the model's accuracy, regular retraining is necessary to include data on bike usage during spring and summer. Moreover, utilising a random forest model may boost the model's predictive capabilities, requiring an EC2 instance with a larger memory capacity.

We did not prioritise security initially, but it is crucial for any application exposed to the internet to have stringent security measures. We can improve our security measures in the future. Currently, anyone can inspect the elements of our website and access the Google Maps API and the OpenWeather API. Although we have implemented IP restrictions on some of these API keys, they may not be sufficient for ensuring security.

The web app's overall architecture could be improved by integrating more adaptable deployment techniques, such as containerisation. As previously stated, the deployment process for our app was challenging, and we encountered several issues. The utilisation of Docker would have simplified the deployment process significantly.

Furthermore, our application could have been further improved by adding a feature that informs users how much CO2 they save by cycling, promoting sustainable transit. We could also have enhanced the user experience by implementing features such as: user feedback and ratings, allowing users to rate and review individual bikes and docking stations; personalised recommendations based on a user's past usage patterns and preferences; and user account management, which would provide users with the ability to create an account and save their favourite bikes and docking stations, thus increasing the website's usability. These features would enhance the overall user experience and provide more value to our users.

One reproducible bug we have identified is that the map occasionally displays two routes instead of one when a user switches from a routing selected using station markers to typing out an address. To avoid this issue, users should clear the search fields before performing a new routing. However, it would be preferable if this were an automatic process, and the map displayed only one route at a time.

8. Appendix

8.1 Stand-up meeting records

Meeting Records Group 2

Sprint 1: 13.02.2023 – 24.02.2023
Sprint 2: 27.02.2023 – 10.03.2023
Sprint 3: 27.03.2023 – 07.04.2023
Sprint 4: 10.04.2023 – 21.04.2023

Date	Completed Work	Planned Work	Comments
17/02/23 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi	A rough layout of our application UI and its features, and functionalities.	Get data scraping up and running Design the Database.	
21/02/23 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi	- Obtained JCDecaux API Key + returned data as JSON dump. - Scraped real time Dublin bikes data. - Created RDS database.	Start creating tables for the database.	RDS DB info: **User** - admin **Password** - mypassword
23/02/23 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi Karl Roe	- Setting up trello product management. - Met with our product owner Karl Roe. - Talk about weather API.	Connect MySQL with Amazon RDS database.	Focus on the product management aspect more, instead of focusing on the technical aspect too much.
27/02/23 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi	- Populate 2 tables in MySQL Database - Connect MySQL with Amazon RDS database	Test code with Amazon RDS to see if the data can be updated every 5 mins	

01/03/23 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi	- Talk about our individual work progress. We have our scraper running on EC2 without issues. RDS tables are being updated as intended. Also managed to scrape weather info.	Get acquainted with Flask. Plan how to handle the weather scraped information.	
02/03/23 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi Karl Roe	- Retrieved Weather API from OpenWeatherMap - Managed to scrape/update current weather data every five minutes - Managed to store scraped data in the database	Decide on having one joined or two separate tables for Dublin bikes and weather data - Option 1: API for bikes and API for weather - Scrapers in EC2 can fetch data from RDS - Option 2: optimiser = another table, will look at combined data - Will look at the most recent data on both bike and weather	Report: - Keep records up to date - Structure: should be divided into Sprints Weather data: - Temperature (!!!) - Wind speed (!!) - Pressure (!) - Description (?) API calls: - Every 5 minutes should suffice Sprint 2 - Have a Flask app - Connect the app with DB - Do not need to run Flask thru EC2 until Sprint 4
07/03/23 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi	- Retrieved Google Maps API key - Created a website for the map using an HTML file that loads CSS/js from independent files - Embedded Google map into the website - Created template for info boxes (eg journey planner, weather, charts, etc.)	- Add station markers on the map	On Python scraper code: - Run on your own RDS!

09/03/23 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi Karl Roe	- Added markers	- Either copy everything into index.html file or move everything into static folder - Add functionality + interactivity to markers	Try Ajax? - Would need to use jQuery Improve reliability of markers - Render code through server, not through your computer Showing availability on map - Create "most recent" table - Overwrites itself, constantly refreshes itself
28/03/23 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi	- Added interactivity to markers (eg obtain info on availability upon clicking)	Create a travel route maker	

30/03/23
 	
Present:
 	
Itgel Ganbold
Miaomiao Shi
Sakura Hayashi
 	
Karl Roe
	- Created travel route maker
	  - Search buttons for start & end locations
	  - Buttons to select mode of travel (walking, driving, transit, cycling)
	  - Added routing: map highlights and displays route
		Front end:
		- Make map bigger
		- Show icons for weather
		- Display icon based on description
			By end of Sprint 3:
			Two charts => can use chart.js:
			1. Daily data -> average daily bikes available (Monday to Sunday)
			2. Hourly data for past few weeks (note: they close at night time)
			
			Machine learning => two options:
			1. Ditch allowing user to enter any place as starting location + end destination and force/hard code the starting location or end destination:
			2. Create good machine learning model that predicts bike availability in all stations => subtract no. of stations
			
			Predict future bike availability => use data for 3-5 days; use day of the week rather than specific date
			- Can use python linear regression model for prediction

			Sprint 4: Add filtering to route maker (search buttons, which is what we currently have)
04/04/2023 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi	- Made map bigger - Weather icons are now displayed based on description	Write machine learning code	
06/04/2023 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi Karl Roe	Wrote machine learning code for predictive model	- Decide on how to manage large amounts of data + how to ensure accuracy - Optimise response times	**Sprint 4:** - Improve ML model - Incorporate filtering options into the route maker - Fix data management + visualisation issues - Focus on frontend!
11/04/2023 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi	Incorporate filtering options into the route maker	Focus on frontend	
13/04/2023 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi Karl Roe	Improved app functionality - Colour-coded availability number on markers and wind speed	Fix git – can make another repo if first one gets too messy	Get everything deployed to EC2 by next week - Doing it through AWS would be faster bc it's over the internet -> AWS CodeDeploy: automates the deployment process to EC2 - 'Hacky' way: clone repo from GitHub to to EC2, install dependencies, run explicitly on 0.0.0.0:80 - Proper way: deploy it with Apache or some standard web server - Containerisation - Merging issues: could create new repo + push all files if current repo is at risk of being deleted Optional: could also add geolocation

			features If there are no bikes available, might be better to flag those markers - Could do it in text form - Could use different colours based on availability - Implementation: just use loops
18/04/2023 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi	Attempted to deploy project on AWS -> EC2 keeps on crashing	Continue deployment, resolve EC2 crashes	Could use Gunicorn and NginX? Global variables were causing issues, esp for managing stations dataset
20/04/2023 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi Karl Roe	Deployed project on AWS with issues -> predictive model not loading (internal server error but working properly/instantly locally) - Runs successfully on port 80 as privileged user	Continue deployment, resolve internal server error	Error handling: - Look at javascript file - Find what modules you need to install in sudo Server (EC2): - Look at flask error logs for internal server errors - Debug in server Things to fix: - Round weather to whole number - Add description of weather? Tips on other ways to fix: - Use containerisation docker (virtualisation tool) -> install dependencies that bundles all components of app as image - Export app as image - Once it works locally, it should work globally -> v useful tool Tips for next step: - Make it more location based, recommendation based, etc.

20/04/2023 Present: Itgel Ganbold Miaomiao Shi Sakura Hayashi	Deployed project on AWS successfully!	Once the application is reviewed, take down EC2	URL: http://34.251.13.69/