EBA5006: Graduate Certificate in Big Data Analytics

Smart Car Park Availability System Using Big Data Analytics

Date of Report
27th April 2025
Team Name
Group 8
Team Members
Ashish Nagineni (A0297393L)
Eswar Raj Rajendran (A0123045W)
Koh Zhen Wen Ian (A0033871H)
Mohammed Muzaffar Ali (A0297946A)
Sudharshan Murugesan (A0299028R)

Contribution and Man-Day Effort

NUS ID No	Name	Contribution
A0297393L	Ashish Nagineni	Data Collection, EDA, Data Correlation Analysis
A0123045W	Eswar Raj Rajendran	Architecture diagrams Training & Inference pipeline(Pub/Sub, Cloud Function, DataFlow, BigQuery tables & views, Vertex AI, GCS, CloudRun), Model Monitoring Script & scheduling
A0033871H	Koh Zhen Wen Ian	Data merging, cleaning, preprocessing Time Series models (GBTR, LSTM) PySpark Inference pipeline and workflow (GCS, DataProc, CloudRun, CloudScheduler) Vertex AI Kubeflow Inference pipeline and scheduling
A0297946A	Mohammed Muzaffar Ali	Data Collection, Data correlation analysis, Visualization
A0299028R	Sudharshan Murugesan	Looker studio dashboard Smart Parking Recommendation , Routing & Visualization

Total Man Day Effort: 55 Man Days

Executive Summary

Singapore faces significant challenges in carpark availability due to its high urban density. To address this critical issue, this project proposes the development of an intelligent system to visualize real-time carpark availability and predict future occupancy across the island. This solution directly tackles the need of vehicle owners for a reliable and easily accessible system that minimizes search times and optimizes journeys.

By leveraging time-series machine learning techniques on historical data, the system will provide accurate predictions for future carpark availability. This real-time and predictive information will be presented on an intuitive interactive dashboard, empowering drivers to locate available parking quickly, reducing frustration and fuel consumption. Furthermore, this system offers significant value to carpark operators for optimized resource management and city planners for data-driven urban development, potentially contributing to reduced traffic congestion and a more sustainable environment.

The technical objectives of this project include building predictive models for 30-minute intervals with a target of over 15% improvement in RMSE and MSE compared to a benchmark. A scalable big data platform and inference pipeline, incorporating distributed computing, streaming, and containerization principles, will be established to handle future data growth. Continuous monitoring for model decay and automated retraining mechanisms will ensure sustained accuracy. Finally, an interactive dashboard will provide users with both current and predicted carpark availability. This comprehensive solution promises to significantly improve the parking experience in Singapore for drivers and provide valuable insights for urban stakeholders.

Table of Contents

List of Figures

List of Tables

1. Introduction & Background

Carpark availability is a critical concern in urban environments like Singapore, where high population density and limited parking spaces often lead to significant challenges for drivers. As urban areas grow, the demand for parking spaces outpaces the supply, resulting in wasted time, increased fuel consumption, and frustration for both drivers and authorities. Additionally, inefficiencies in carpark utilization contribute to higher congestion and environmental pollution.

2. Problem Statement

Business Problem:
Vehicle owners in Singapore need a reliable and easily accessible system that provides real-time information on carpark availability across the island to minimize search times, reduce frustration, and optimize their journeys.

Proposed Solution & Business Value:
This project aims to visualize real-time car park availability and predict carpark availability for future time slots. By leveraging time-series machine learning techniques, this system will predict carpark occupancy based on historical data. Finally, the real-time and predicted carpark availability will be displayed on an interactive dashboard. This dashboard will provide stakeholders—such as drivers, carpark operators, and city planners—with intuitive access to real-time information and predictive insights, enabling data-driven decision-making and optimized resource management.

3. Business Objectives

This system can provide immense value to various stakeholders:

1. **For Drivers**: The system can guide driver to available parking spaces, reducing the time spent searching for parking, enhancing their overall experience, and reducing frustration.
2. **For the Environment**: Efficient parking management can help in reducing traffic congestion and thus CO2 emissions, contributing to a more sustainable urban ecosystem.

4. Technical Objectives

1. Build Time Series models to predict available carpark lots in 30 mins, all around Singapore.
2. Compare Time Series models RMSE and MSE to simplistic benchmark model and achieve >15% improvement in these scores.
3. Build a Big Data Platform Ingestion and Inference Pipeline that are scalable to handle future data growth. Principles applied are distributed computing, streaming and containerization.
4. Continue to monitor Inference Pipeline Metrics to detect for Model Decay. Models can be retrained on newer data should RMSE or MSE metrics fall below 15% improvement.
5. Build an interactive dashboard to provide predicted as well as current available lots to the user.

5. Project Design & Scope of Work

5.1 Project Design

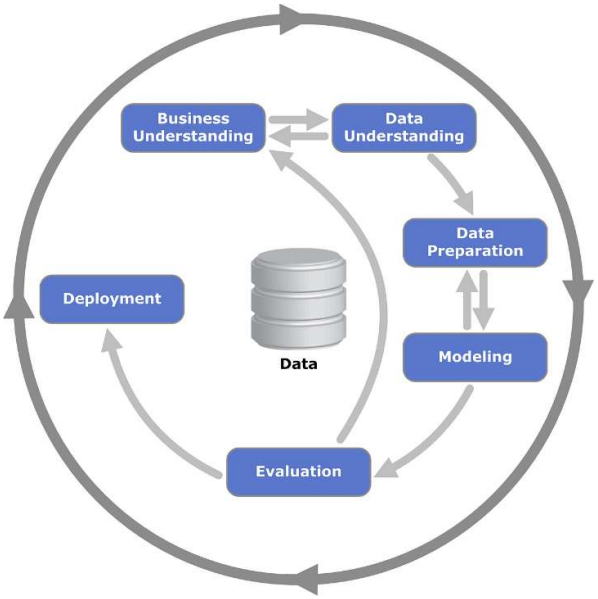

Figure 1 CRISP-DM cycle

1. **Business Understanding**
   - Analyse the needs the concerns of the vehicle owners and other stakeholders
2. **Data Collection and Understanding**
   - Find relevant data pertaining to car park availability
   - Analyse and qualify the data collected
3. **Data Preparation**
   - Merging, cleaning, transforming and filtering of the raw data
4. **Modelling**
   - Build Machine Learning models to predict Availability of car park lots
5. **Evaluation**
   - Quantify model performance with model metrics, comparing it with benchmark
6. **Deployment**
   - Deploy the following on a cloud data platform with big data, scalability and distributed computing in mind:
     - Data Ingestion pipeline
     - Data Warehouse Storage
     - Model Training pipeline
     - Inference pipeline

5.2 Scope of Work

1. Identifying the Analytics Problem

Urban parking congestion is a growing challenge in Singapore, with limited parking spaces leading to traffic delays, higher fuel consumption, and increased carbon emissions. Drivers struggle to find available parking spots, causing frustration and contributing to further congestion.

While existing parking management systems provide real-time occupancy data, they lack advance predictive and AI-driven capabilities to help drivers anticipate parking availability ahead of time. This

project aims to bridge that gap by developing an AI-powered smart parking system tailored to Singapore's urban landscape.
By leveraging real-time parking availability data, predictive analytics, and smart routing, the system will:

- Predict future parking availability based on historical trends, time of day, and location.
- Help drivers navigate efficiently to available parking spots through AI-driven routing.
- Optimize city-wide parking utilization using machine learning to adapt to dynamic urban conditions.

2. Data Ingestion Strategy

Real-Time Data Sources

Land Transport Authority Carpark Availability API [1] – Provides real time data on the number of available lots for HDB, LTA and URA carparks in Singapore

Batch Data Sources

The above real time data source would be collected for 2 months at regular 5 minutes intervals. This static data will be used to train machine learning models.

API Deployment

3. Storage Strategy

The project utilizes a scalable and efficient cloud-based storage solution to manage the large volumes of data generated. Google Big Query serves as the data warehouse.

Raw data collected from the API are stored in Google BigQuery. For real-time predictions, the machine learning model retrieves relevant data from Bigtable to generate parking availability forecasts.

4. Model Training – Time-Series Machine Leaning model

A Time-series ML model is trained on historical parking data to predict future availability. The model can be re-trained as new data flows in, enhancing its ability to forecast parking and optimize routing.

5. Inference pipeline

PySpark Inference pipeline is utilized to ensure pipeline is scalable. To provide the spark environment, Google Data Proc is used.

A FastAPI-based backend is also deployed on Google Cloud Run, providing scalable endpoints for real-time parking availability, predictive analytics, and optimized routing. This ensures that the system can efficiently handle large volumes of concurrent requests without sacrificing performance.

6. End User Application

Dashboard & Visualization

Google Data Studio is used to create an interactive dashboard that provides real-time insights into parking occupancy and system performance. This dashboard helps users make informed decisions based on actionable data, with a focus on offering real-time availability and routing

information. The dashboard can be used by drivers to monitor parking trends in real-time and make data-driven decisions

Smart Routing

By integrating Google Maps API/ OpenStreetMap API, the system calculates optimal routes to parking spots based on real-time data, reducing travel time and congestion. The routing engine dynamically adjusts to changing traffic conditions, providing the most efficient path to available parking, directly helping users navigate to their nearest available parking spot.

6. Data Set Used & Data Description

Table 1 LTA Carpark Dataset

Attributes	Description	Data Type
CarParkID	A unique code for this carpark	string
Area	Area of development / building	string
Development	Major landmark or address where carpark is located	string
Location	Latitude and Longitude map coordinates	string
AvailableLots	Number of lots available at point of data retrieval	int
LotType	Type of lots: C (for Cars), H (for Heavy Vehicles), Y (for Motorcycles)	string
Agency	Agencies: HDB, LTA, URA	string

7. Data Collection & Data Processing

Using a locally run python script, data was fetched from the Land Transport Authority Carpark Availability API.

The python scripts save the data pulled from the real-time API to a local MongoDB database. Data is then extracted from the database in CSV format and placed on Big Query as batch data. This would provide the batch data that would be used for training the Time Series Models, before the ingestion pipeline is established. The code can be found in the appendix section.

Handling Missing Data (due to python script not running) – Redundancy, Merging and Imputation

Redundancy

Two computers ran the script continuously for a period of 45 days to fetch data concurrently. The API fetch script is called every 5 mins and at specific timings (00min, 05min, 10min, 15min… etc). This provides redundancy, increasing the chances that complete data could be obtained for model training.

After retrieving the data from both computers, it was found that both API scripts had failed at various points of operation. The next step was to merge the 2 datasets.

Merging

After merging the 2 datasets, there were still some missing timestamps. Thankfully these were only small gaps. These time gaps were taken note of and the missing values had to be handled.

Imputation

Next data imputation was carried out. The following steps were taken:

- Dataset was separated by various carparks in dictionary (one time-series per key)
- For every car park, linear interpolation was carefully performed for all the missing values

It is important to note that this only took care of the missing values due to the failure of the scripts to run. Only those time gaps were handled.

Data Quality and Data Filtering

After handling the missing data above, the data was further checked. It was found that:

- The number of entries per each carpark were not identical; there is a discrepancy in the value
- We can conclude that even when the python script is running properly, the API is not consistent in providing available lot data of every carpark i.e. some car parks are left out in some fetches.
- It was found that the carparks under URA provided the most consistent and complete data
- To obtain consistent and unbroken data for the time-series prediction, the carpark data was filtered for only Agency - URA

Also to further simplify the car park data, dataset was filtered for only entries with lot type = “Cars”

1. All carparks have lots for Cars, but only some carparks have lots for Heavy Vehicles and Motorcycles
2. Thus, for this project, we have limited the scope to only focus on the Available Lots for Cars
3. 2 separate models could be built for the other 2 lot types

All relevant code can be found in the appendix section.

8. Big Data Architecture

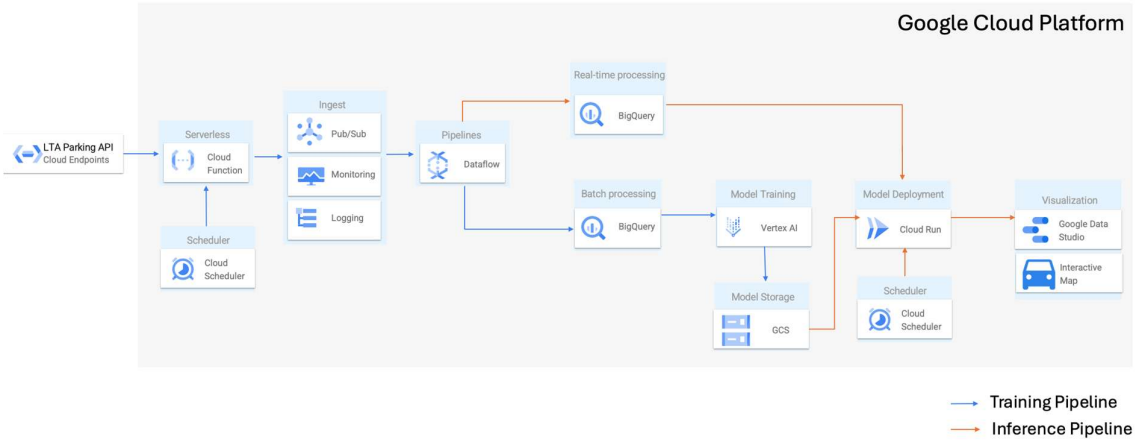

Figure 2 End-to-End Smart Car Park Availability Prediction Architecture on Google Cloud Platform using LSTM Model

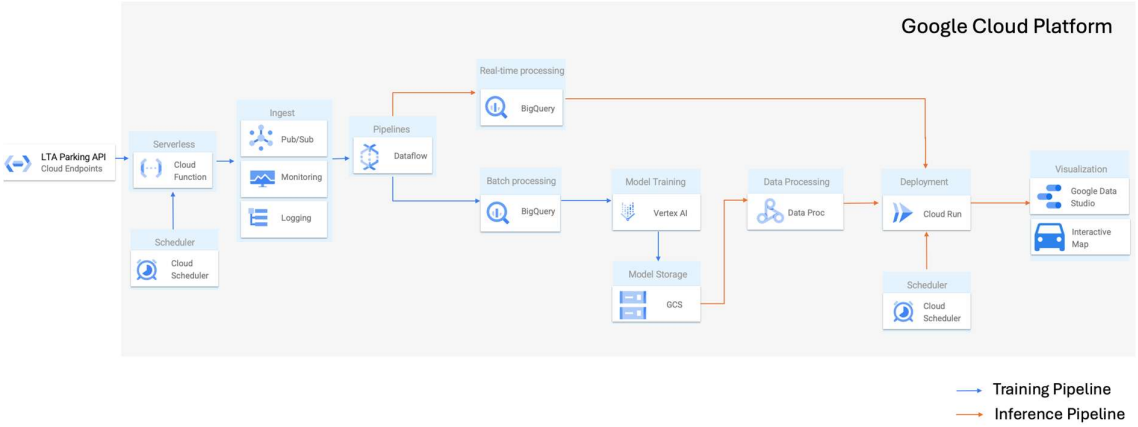

Figure 3 End-to-End Smart Car Park Availability Prediction Architecture on Google Cloud Platform using GBT Regressor Model

9. Training pipeline

Component	Purpose
Cloud Function + Pub/Sub	Collects real-time LTA Parking API data and streams it into the system
Dataflow (Streaming)	Processes and cleans the data before pushing into BigQuery (real-time store)
BigQuery (Batch)	Stores historical data used for model training
Vertex AI	Trains the prediction model using historical data
GCS (Model Storage)	Stores trained model artifacts for deployment

Figure 4 Training Pipeline Components

9.1 Cloud Function + Pub/Sub for LTA Carpark API Ingestion

To enable real-time ingestion of car park availability data from the LTA (Land Transport Authority) API, a Google Cloud Function was developed and deployed. This function retrieves live car park availability information and publishes the records to a Pub/Sub topic for downstream streaming and processing.

Key Implementation Details:

- **Cloud Function Setup:**
  - The function is written in Python (main.py).
  - It fetches data from the LTA Carpark Availability API using an API key.
  - For each carpark record received, the function publishes a message to a Pub/Sub topic (carpark-data-topic).
  - Error handling is incorporated to manage failed API calls or publishing issues.
- **Pub/Sub Integration:**
  - A Pub/Sub publisher client is initialized within the Cloud Function.

- Each incoming carpark data record is encoded in UTF-8 and sent to the designated topic for real-time ingestion into the pipeline.

- **Deployment:**
  - The Cloud Function was deployed using the GCP (Google Cloud Platform) console under the project smart-car-park-availability-1.
  - Multiple deployment iterations were created to support ingestion and publishing operations (fetch-and-publish, fetch-and-publish-carpark, fetch-lta-data).

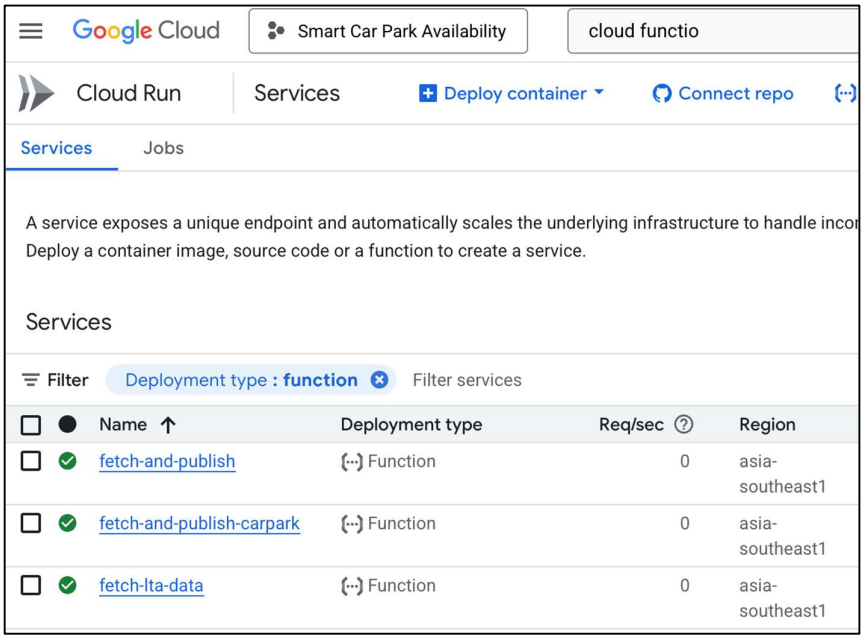

Figure 5 Screenshot of the deployed Cloud Functions within the GCP Console, showing active services.

9.2 Dataflow (Streaming) for Real-Time Carpark Data Ingestion

To enable real-time processing of car park availability data, an Apache Beam pipeline was developed and deployed on Google Cloud Dataflow. This pipeline ingests messages from the Pub/Sub topic and streams them directly into a BigQuery table for downstream analytics and model monitoring.

Key Implementation Details:

- **Apache Beam Pipeline:**
  - The pipeline script (stream_to_bigquery.py) uses Apache Beam in Python to orchestrate the streaming job.
  - It reads car park availability data from a Pub/Sub topic (carpark-data-topic).
  - The data is parsed from JSON format using a custom DoFn class (ParseJson).
  - After parsing, the records are written into a BigQuery table (lta_data.carpark_availability).
- **Streaming Configuration:**

- The pipeline is configured in **streaming mode** to allow continuous ingestion without manual triggers.
- Schema for the BigQuery table is explicitly defined to match the incoming fields (e.g., CarParkID, Area, Development, Location, AvailableLots, LotType, Agency).
- Write operation uses WRITE_APPEND to add new records, ensuring no data overwrites.

- **Deployment Details:**
  - The Dataflow job is deployed in the asia-southeast1 region.
  - The job type is **streaming**, and the current status is **running** successfully with low data lag (< 1 minute).
  - Pipeline stages include:
    - ReadFromPubSub
    - ParseJSON
    - WriteToBQ
  - Monitoring shows most of the time is spent writing to BigQuery, with minimal operational latency.

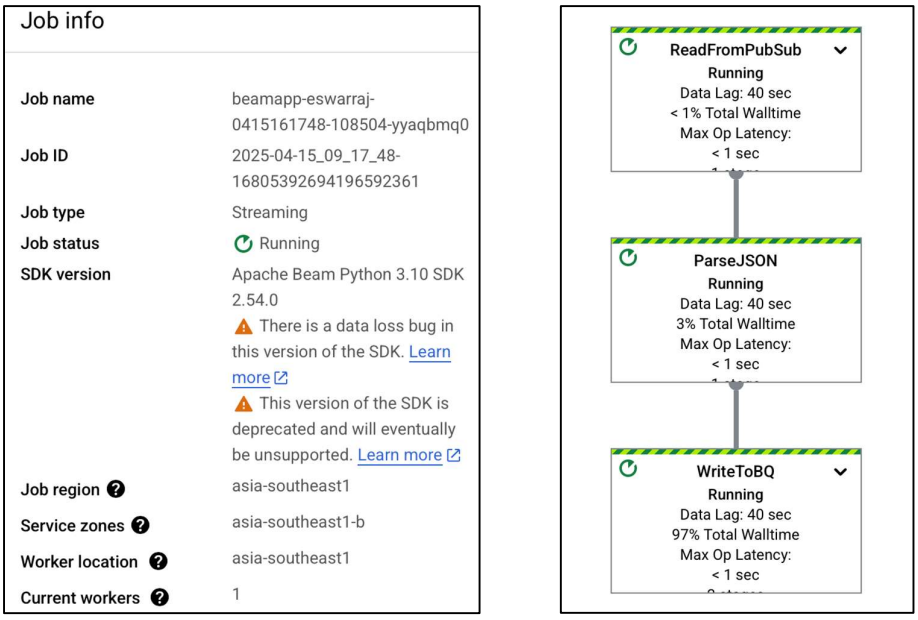

Figure 6 Dataflow Diagrams

a) Dataflow job details showing deployment information and status in GCP Console.

b) Visualization of pipeline stages: reading, parsing, and writing data.

9.3 BigQuery for Structured Data Storage and Model Training

BigQuery serves as the central data warehouse for storing both historical and real-time car park availability data. It supports structured data storage, data aggregation, and is a key source for machine learning model training and monitoring.

Key Implementation Details:

- **Purpose:**
  - To store structured and cleansed car park availability data for downstream analytics, model training, and model monitoring activities.
- **Tables:**
  - **bulkdata:**
    - Stores approximately two months of historic car park data.
  - **bulkdata_latest:**
    - Contains the latest two weeks of delta data for incremental updates.
  - **carpark_availability:**
    - Real-time car park availability data ingested every 5 minutes from LTA APIs.
  - **carpark_predictions_30mins:**
    - Holds the predicted available lots for each car park 30 minutes into the future.
  - **model_monitoring_metrics:**
    - Stores evaluation metrics (MAE, RMSE, MAPE) for ongoing model performance monitoring.
- **Views:**
  - **view_carpark_availability:**
    - Combines (UNION) data from bulkdata, bulkdata_latest, and carpark_availability.
    - Provides a unified dataset used for training the machine learning prediction models.

9.4 Vertex AI (Training)

Vertex AI was used as the environment for model training activities, providing a scalable and managed infrastructure to build the parking lot availability prediction model.

Key Implementation Details:

- **Input Data:**
  - The training input is sourced from the **aggregated BigQuery view** view_carpark_availability.
  - This view provides unified car park data at **5-minute intervals**, combining historical, delta, and real-time records.
- **Training Pipeline:**
  - Training was performed within a **Vertex AI Workbench** instance, using a Jupyter Notebook environment.
  - The notebook instance is configured with:
    - **Zone:** asia-southeast1-a
    - **Machine Type:** e2-standard-4 (4 vCPUs, 16 GB RAM)
    - **Environment:** TensorFlow Enterprise 2.17 (Intel® MKL-DNN Optimized)
- **Output:**
  - After training, the trained machine learning model is **exported and saved to Google Cloud Storage (GCS)** for further deployment and inference activities.

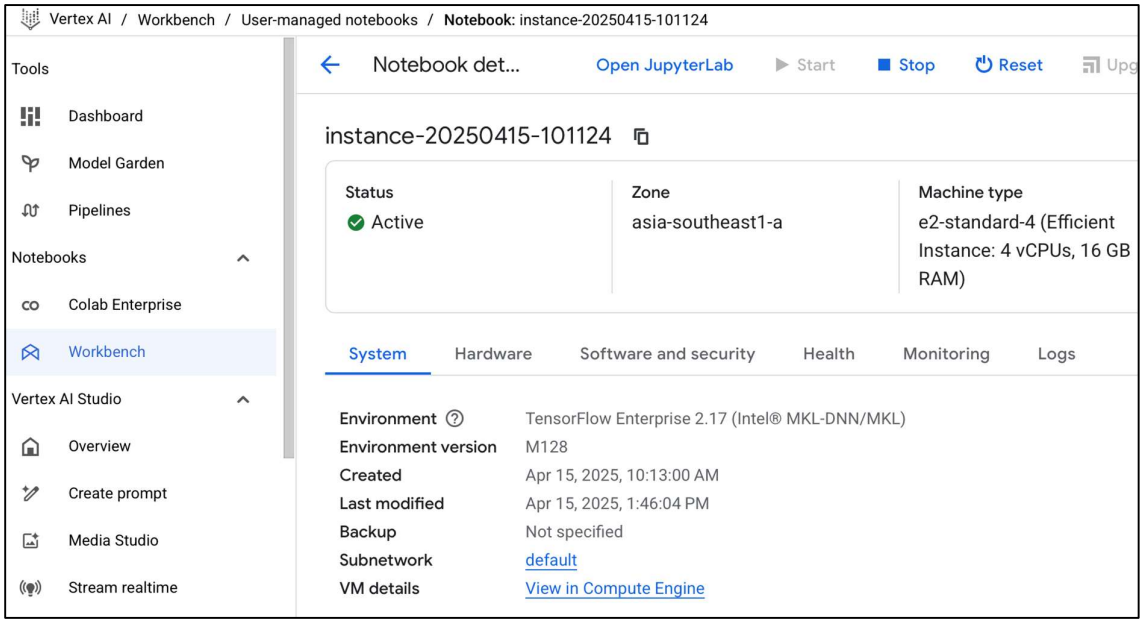

Figure 7 Screenshot of the active Vertex AI Workbench Notebook instance used for training

9.5 Google Cloud Storage for Model Versioning

Google Cloud Storage (GCS) is used to store, manage, and version the trained machine learning models produced during the car park availability prediction project.

Key Implementation Details:

- **Purpose:**
  - To maintain a centralized and organized repository for storing trained model files.
  - Enables version control and backup of machine learning models for future deployment or retraining.
- **Storage Structure:**
  - **Bucket:** gs://prediction_models_x
  - The bucket stores multiple versions of LSTM models (models_lstm_model_x1.keras, models_lstm_model_x2.keras, etc.), each corresponding to different training iterations or hyperparameter tuning experiments.
- **Benefits:**
  - Seamless integration with downstream GCP services like **Cloud Run** and **Vertex AI** for model deployment and inference.
  - Supports **versioned backups**, allowing easy rollback or comparison between different trained models.

10. Time Series Modelling & Evaluation Methodologies

10.1 Preprocessing for Time Series

Preprocessing is the first step in preparing data for Time Series Modelling. It ensures the right feature are selected, feature engineer is performed, and data is scaled to the right format and range for the time series machine learning algorithms.

Below details the steps performed:

1. Drop columns not used in training

"Area", "Development", "Location", "LotType", "Agency"

These data were deemed to not contribute anything to the time series training. Also they are stationary values and are tied to the "CarParkID" variable. So only "CarParkID" is necessary for the Machine Learning algorithm.

2. Perform Feature Engineering

"Min", "Hour", "DayOfWeek" features were created from timestamp. This provides the algorithm with more data to work with.

	CarParkID	AvailableLots	timestamp
0	A0007	77.0	2025-02-16 00:00:00+00:00
1	A0011	5.0	2025-02-16 00:00:00+00:00
2	A0017	120.0	2025-02-16 00:00:00+00:00
3	A0021	14.0	2025-02-16 00:00:00+00:00
4	A0024	0.0	2025-02-16 00:00:00+00:00
...	...	...	...
717780	U0042	31.0	2025-03-16 00:00:00+00:00
717781	V0007	0.0	2025-03-16 00:00:00+00:00
717782	W0029	65.0	2025-03-16 00:00:00+00:00
717783	W0055	33.0	2025-03-16 00:00:00+00:00
717784	Y0019	19.0	2025-03-16 00:00:00+00:00

717785 rows × 3 columns

Feature Engineering

	CarParkID	AvailableLots	timestamp	Min	Hour	DayOfWeek
0	A0007	77.0	2025-02-16 00:00:00+00:00	0	0	6
1	A0011	5.0	2025-02-16 00:00:00+00:00	0	0	6
2	A0017	120.0	2025-02-16 00:00:00+00:00	0	0	6
3	A0021	14.0	2025-02-16 00:00:00+00:00	0	0	6
4	A0024	0.0	2025-02-16 00:00:00+00:00	0	0	6
...	...	...	...	...	...	...
717780	U0042	31.0	2025-03-16 00:00:00+00:00	0	0	6
717781	V0007	0.0	2025-03-16 00:00:00+00:00	0	0	6
717782	W0029	65.0	2025-03-16 00:00:00+00:00	0	0	6
717783	W0055	33.0	2025-03-16 00:00:00+00:00	0	0	6
717784	Y0019	19.0	2025-03-16 00:00:00+00:00	0	0	6

717785 rows × 6 columns

Figure 8 Feature Engineering

3. Apply LabelEncoder

Label Encoder is then applied to CarParkID to convert categorical data to a numerical one. Label Encoder was chosen instead of One hot encoding as there are many carparks. Using One hot encoding avoids potential bias by treating each category as a separate, independent feature. However, it will lead to a significant increase in the number of features. Thus, label encoder was used despite it introducing some bias.

	CarParkID	AvailableLots	timestamp	Min	Hour	DayOfWeek	CarParkEncoded
0	A0007	77.0	2025-02-16 00:00:00+00:00	0	0	6	0
89	A0007	77.0	2025-02-16 00:05:00+00:00	5	0	6	0
178	A0007	77.0	2025-02-16 00:10:00+00:00	10	0	6	0
267	A0007	77.0	2025-02-16 00:15:00+00:00	15	0	6	0
356	A0007	77.0	2025-02-16 00:20:00+00:00	20	0	6	0
...	...	...	...	...	...	...	...
717428	Y0019	19.0	2025-03-15 23:40:00+00:00	40	23	5	88
717517	Y0019	19.0	2025-03-15 23:45:00+00:00	45	23	5	88
717606	Y0019	19.0	2025-03-15 23:50:00+00:00	50	23	5	88
717695	Y0019	19.0	2025-03-15 23:55:00+00:00	55	23	5	88
717784	Y0019	19.0	2025-03-16 00:00:00+00:00	0	0	6	88

717785 rows × 7 columns

Figure 9 Label Encoder

After fitting and applying this label encoder, it is saved to a .pkl (pickle) file to be used later in the inference pipeline.

4. <u>Normalise values using MinMaxScalar</u>

Finally, MinMaxScalar (0,1) is applied to “CarParkEncoeded”, “AvailableLots”, “Min”, “Hour” and “DayOfWeek” values.

This is necessary for LSTM input as it ensures:

a. Equal Contribution of Features
b. Gradient Stability and Faster Convergence
   Matching to Input range to all Activation Functions

Min-Max Scalar

	CarParkID	AvailableLots	timestamp	Min	Hour	DayOfWeek	CarParkEncoded	AvailableLots_MM	CarParkEncoded_MM	Min_MM	Hour_MM	DayOfWeek_MM
0	A0007	77.0	2025-02-16 00:00:00+00:00	0	0	6	0	0.140255	0.0	0.000000	0.0	1.000000
89	A0007	77.0	2025-02-16 00:05:00+00:00	5	0	6	0	0.140255	0.0	0.090909	0.0	1.000000
178	A0007	77.0	2025-02-16 00:10:00+00:00	10	0	6	0	0.140255	0.0	0.181818	0.0	1.000000
267	A0007	77.0	2025-02-16 00:15:00+00:00	15	0	6	0	0.140255	0.0	0.272727	0.0	1.000000
356	A0007	77.0	2025-02-16 00:20:00+00:00	20	0	6	0	0.140255	0.0	0.363636	0.0	1.000000
...	...	...	...	...	...	...	...	...	...	...	...	...
717428	Y0019	19.0	2025-03-15 23:40:00+00:00	40	23	5	88	0.034608	1.0	0.727273	1.0	0.833333
717517	Y0019	19.0	2025-03-15 23:45:00+00:00	45	23	5	88	0.034608	1.0	0.818182	1.0	0.833333
717606	Y0019	19.0	2025-03-15 23:50:00+00:00	50	23	5	88	0.034608	1.0	0.909091	1.0	0.833333
717695	Y0019	19.0	2025-03-15 23:55:00+00:00	55	23	5	88	0.034608	1.0	1.000000	1.0	0.833333
717784	Y0019	19.0	2025-03-16 00:00:00+00:00	0	0	6	88	0.034608	1.0	0.000000	0.0	1.000000

717785 rows × 12 columns

Figure 10 Normalization using Min-Max Scalar

Similarly, after fitting and applying the various min-max scalars, they are saved to a .pkl (pickle) file to be used later in the inference pipeline.

All relevant code can be found in the appendix section.

10.2 Sequencing for Time Series

Sequencing is an important step in the preparation for training a Time Series Machine Learning algorithm. It would determine the length of the time series used to do predictions, as well as the target prediction time.
Outlined below are the methodology and details of the sequencing performed

1. Demand Prediction Objective

The prediction target is set to be the available lots for each car park in 30mins. This value was chosen as for any shorter time period, the current available lots in the car park would already serve as a good indicator. For LSTM model, 2 more sequences were also prepared, predictions for available lots in 45 mins and 1 hour. This would allow user flexibility to choose the prediction window.

2. Input to Time Series Machine Learning

A sequence of 5 scaled data of the following were used: “CarparkID”, “AvailableLots”, “Min”, “Hour”, “DayOfWeek”. This results in a 5 x 5 input for each time series training input, which would be used later as the inference input as well.

3. Prediction Target

Based on the objective stated above, the target prediction is set to be the number of available lots of every carpark 30 minutes (45/1??) later. Thus the time series value is taken 6 (6 x5 minutes intervals), (the rest?) number of time steps from the last input value.

4. Train-Test Split

Finally for training and testing the time series, a 80-20 split was employed. To ensure that all the car park data are captured in the training and tested subsequently, a custom code was written to carefully split the data to a train set and test set, ensuring that every car park is represented in both sets.

All relevant code can be found in the appendix section.

10.3 Models Trained

10.3.1 Model 1: Gradient Boosted Tree Regressor - SparkML

A Gradient Boosted Tree (GBT) Regressor was first chosen because it is natively supported in SparkML and was a good candidate for time series algorithms. Working in Spark environment for the inference pipeline would use distributed computing and provide scalability, which are in line with the objectives of a big data project.
The following hyperparameters were chosen after many iterations:

Table 2 GBT Regressor Hyperparameters

Hyperparameter	Value	Description
numTrees (maxIter)	250	The number of trees in the ensemble (must be set).
maxDepth	10	Maximum depth of each decision tree.
stepSize (learningRate)	0.01	Shrinkage parameter applied to the contribution of each tree.
maxBins	32	Maximum number of bins used for discretizing continuous features.

minInstancesPerNode	1	Minimum number of instances each child node must have after a split.
minInfoGain	0	Minimum information gain required for a split to be considered.
maxMemoryInMB	256 MB	Maximum memory allocated for histogram aggregation during tree building.
cacheNodeIds	FALSE	Whether to cache node IDs for each instance (can speed up training for deep trees).
minWeightFractionPerNode	0	Minimum fraction of the sum of instance weights that a child node must represent after a split.
subsamplingRate	1	Fraction of the training data used for training each tree.
lossType	squared	Loss function to be minimized ("squared" for L2 loss, "absolute" for L1 loss).
featureSubsetStrategy	all	Number of features to consider for splits at each node ("auto", "all", "onethird", "sqrt", "log2").
checkpointInterval	10	Frequency (in iterations) at which the algorithm checkpoints the ensemble of trees.
seed	(random)	Random seed for reproducibility.

All relevant code can be found in the appendix section.

10.3.2 Model 2: Long Short=Term Memory Network -Tensorflow

To improve upon the performance of the predictions, Long Short-Term Memory (LSTM) networks model was also trained. LSTM are a type of recurrent neural that are well-suited for time series data because they can capture the temporal dynamics and patterns that evolve over time, considering past observations to predict future values.
The following architecture, hyperparameters, and training epcoch/ training batch size were chosen after many iterations:

```
model = Sequential()
model.add(LSTM(64, activation='tanh', input_shape=(X_train.shape[1], X_train.shape[2])))
model.add(Dense(32, activation="relu"))
model.add(Dense(16, activation="relu"))
model.add(Dense(8, activation="relu"))
model.add(Dense(4, activation="relu"))
model.add(Dense(1, activation="sigmoid"))
model.compile(optimizer='adam', loss='mse')
model.summary()

hist = model.fit(X_train, y_train, epochs=600, batch_size=5096, validation_data=(X_test, y_test))
```

Figure 11 LSTM Architecture

Table 3 LSTM Architecture Details

Layer	Type	Nodes	Activation
1	LSTM	64	tanh
2	Dense	32	relu
3	Dense	16	relu
4	Dense	8	relu
5	Dense	4	relu
6	Dense	1	sigmoid

Parameter	Value
Input shape	5 x 5
Output	1
Optimizer	Adam
Loss Function	MSE
Epochs	600
Batch Size	5096

All relevant code can be found in the appendix section.

10.3.3 Benchmark Model

A simple Benchmark model was also obtained by using the current available lots as the predicted value. This benchmark model is critical in evaluating the model's predictive effectiveness (versus relying on the current available lots).

10.4 Evaluation Metrics

Three sets of sequencing were done. Below are the evaluation metrics for 30, 45, and 60 minutes predictions of available lots.

30 minutes predictions

Table 4 30 minutes prediction metrics

Metrics	Benchmark 30 mins	LSTM 30 mins	GBT Regressor 30 mins
MSE	41.58	26.85 (35.4%)	39.06 (6.1%)
RMSE	6.45	5.18 (20.0%)	6.25 (3.1%)

For the prediction of available lots in 30 minutes, both the LSTM and GBT performed better than the benchmark. However, LSTM is the better model of the two, improving from the benchmark by 20% in the RMSE.

45 minutes predictions

Table 5 45 minutes predictions metrics

Metrics	Benchmark 45 mins	LSTM 45 mins
MSE	77.76	41.58 (46.5%)
RMSE	8.81	6.45 (26.8%)

For the prediction of available lots in 45 minutes, only LSTM was trained. It performed much better than the benchmark, improving by 26.8% for its RMSE.

60 minutes predictions

Table 6 60 minutes predictions metrics

Metrics	**Benchmark 60 mins**	**LSTM 60 mins**
MSE	121.94	59.10 (51.5%)
RMSE	11.04	7.69 (30.3%)

Finally, for the prediction of available lots in 60 minutes, only LSTM was trained as well. It performed even better than the benchmark, improving by 30.3% for its RMSE.

One trend observed is that the performance of both the benchmark and the model declines as the prediction horizon extends. This is a common occurrence as the uncertainty associated with forecasting increases over longer timeframes.

Notably, the model's performance degrades at a slower rate. This is anticipated as well because while relying on the immediately available slots becomes increasingly unreliable with time, a model specifically trained for these kinds of longer-term predictions should maintain a more reasonable level of accuracy

Overall, the LSTM model achieved our technical goals of >15% improvement over benchmark, while the GBT Regressor only marginally improved over the benchmark model.

11. Inference Pipelines

To meet our technical goals of achieving predictive accuracy and establishing a future-proof, scalable big data architecture, we explored and developed a total of three distinct inference pipeline solutions.

Firstly, a PySpark pipeline was built to utilise the a Spark environment for the pipeline, ensuring scalability and big-data readiness. However, it was found that it could only run the GBT Regressor, which turned out to perform not as good.

Secondly, a solution for LSTM model was found using CloudRun, utilising REST API built with Flask. This proved to be a valid solution for deploying the LSTM model. This solution is not fully automated, and the API call overhead may become a bottleneck.

Finally, a Vertex AI Kubeflow pipeline was built. This pipeline is fully optimized for big data, and scales with distributed compute, large datasets flowing between tasks.

11.1 PySpark Inference Pipeline

Data Proc Setup

Data Proc is a fully managed, scalable service by Google Cloud that allows users to run open-source, big data processing frameworks like Apache Spark and Apache Hadoop on Google Cloud Platform. It will provide the necessary platform for running the PySpark inference pipeline.

Master node	Standard (1 master, N workers)
Machine type	n4-standard-2
Number of GPUs	0
Primary disk type	hyperdisk-balanced
Primary disk size	100GB
Local SSDs	0
Worker nodes	3
Machine type	n4-standard-2
Number of GPUs	0
Primary disk type	hyperdisk-balanced
Primary disk size	100GB
Local SSDs	0

Figure 12 DataProc Setup

PySpark Code Running on DataProc

The PySpark inference pipeline is outlined below:

1. Data is fetched from Big Query table
2. Data is checked to ensure it can be sequenced for prediction
3. GBT Regression Model, Label Encoder and Min-max scalars are loaded from Google Cloud Storage
4. All preprocessing steps are performed
    1. Feature Engineering
    2. Label Encoding
    3. Normalise (Min-Max scalar)
    4. Data is sequenced for the prediction model
5. Sequenced data is fed to the prediction model
6. Prediction data is processed to match prediction table output schema and update Big Query table

Pipeline automated workflow:
The automated workflow for running the PySpark inference pipeline is outlined below:

1. Cloud Scheduler triggers every 5 mins based on a cron job expression to run a program on Cloud Run
2. Cloud Run function fetches the PySpark Program from Google Cloud Storage and then runs it on Data Proc
3. PySpark program runs on Data Proc

All relevant code for this section can be found in the appendix section.

11.2 CloudRun Inference Pipeline – LSTM Model

To serve real-time predictions, a REST API was built using Flask, containerized with Docker, and deployed to Google Cloud Run. This allows scalable, on-demand inference from the trained LSTM model.

11.2.1 Key Implementation Details:

- **Purpose:**
  - Serve ML predictions via a scalable, serverless REST API endpoint.
- **Deployment:**
  - **Flask Application:** A lightweight Flask app was developed to expose a /predict endpoint.
  - **Model Loading:**
    - The trained .keras LSTM model is loaded from a GCS bucket at application startup.
  - **Containerization and Deployment:**
    - The app was containerized using Docker and deployed to Cloud Run in the asia-southeast1 region.
- **Endpoint Details:**
  - **Route:** /predict
  - **Method:** POST
  - **Payload:** JSON containing:
    - sequences — list of sequences for each car park
    - carpark_ids — list of corresponding CarParkIDs
  - **Output:** JSON list of predicted lots for each CarParkID.

11.2.2 Prediction Pipeline:

The deployed Cloud Run service is integrated into the prediction pipeline by sending batch requests and writing the output back into BigQuery.

11.2.2.1 Key Implementation Details:

- **Step 1: Prepare Input Sequences**
  - Grouped the latest data from BigQuery by CarParkID.

  - Formatted each carpark's historical sequence to match LSTM input requirements.
- **Step 2: Call Cloud Run API**
  - Made POST requests to the /predict endpoint hosted on Cloud Run (`https://lstm-predictor-<id>.run.app/predict`).
  - Received predicted available lots for each carpark in the response.
- **Step 3: Save Predictions Locally**
  - Compiled predictions into a Pandas DataFrame (`predicted_df`).
  - Saved the results to a CSV file (`carpark_predicted_30min_from_cloudrun.csv`) for inspection.
- **Step 4: Write Predictions to BigQuery**
  - Used BigQuery Python client to load the DataFrame into the carpark_predictions_30min table.
  - Configured with WRITE_TRUNCATE to replace previous prediction data with each new prediction cycle.

Execution Environment:

Vertex AI Workbench (Jupyter environment) was used to run the end-to-end prediction script.

11.3 Vertex AI Kubeflow Inference Pipeline -LSTM Model

11.3.1 Kubeflow Pipeline Containerized Components

To Setup the Vertex AI Kubeflow pipeline, a 3-stage approach was adopted. This corresponds to 3 Kubeflow Pipeline containerized components.

The containerized components are:

1. Load Big Query Table and Check Data
    - Load the data from Big Query Table
    - Check data to ensure it can be sequenced for prediction

2. Preprocess and sequence Data
    - Label Encoder and Min-max scalars are loaded from Google Cloud Storage
    - All preprocessing steps are performed
    - Sequencing of the data is performed

3. Prediction output and Big Query Table update
    - 3 LSTM models are loaded from GCS (30, 45, and 60 minutes)
    - Prediction results are arranged with 12 historical data points
    - Big Query Table is updated

Furthermore, between each component, there is a logical check to ensure that the previous component has the data in the exact size and structure.

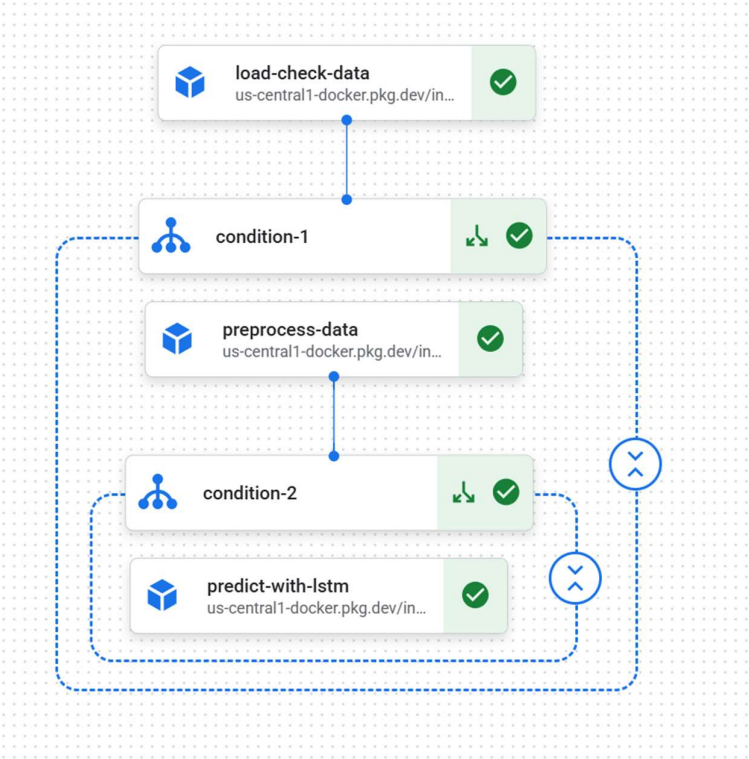

Figure 13 Vertex AI Kubeflow Pipeline

11.3.2 Custom Container Image

To speed up the pipeline, custom docker container images were used. This allows the packages to be pre-installed for the components, so they don't have to be installed during each run=time. This allowed the pipeline to run significantly faster. The custom contain was then uploaded to Google Artifacts Registry for the pipeline code to access.

11.3.3 Scheduling and Cache

Vertex pipeline has a built-in scheduler that takes a cron job expression to run submitted pipelines at a regular interval. A schedule was setup to run the pipeline every 5mins starting at the 1st minute of each hour.

However, Vertex pipelines also has a cache behaviour that caches the result of previous runs should the inputs remain unchanged. This behaviour was be managed carefully in the pipeline code:

```
task.set_caching_options(False) # Disable caching for this component
```

Display name	Status	Frequency	Pipeline	Next execution time	Last run	Last executions
my-newest-pipeline-20250427033719	Completed	TZ=Asia/Singapore 1-59/5 * * * *	64a17be50362	Apr 27, 2025, 4:31:00 AM	my-newest-pipeline-20250427033719-20250426132600425	✓✓✓✓✓

Figure 14 Vertex AI pipeline schedule and successful run

The successful runs of the Kubeflow pipeline are shown above. As this pipeline also loaded three LSTM models to predict for 30, 45, and 60 minutes, below is the Big Query table after this pipeline is run. More historical availability was also made available for easy access by the dashboard.

Prediction for lots in 30, 45, 60 minutes

CarParkID	Location	AvailableL...	LatestTimestamp	Pre...	Predi...	Pred...	t-55	t-50	t-45	t-40	t-35	t-30	t-25	t-20	t-15	t-10	t-5	t
B0031	1.31411103114537 103.85960...	0	2025-04-27 04:25:00 UTC	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
N0013	1.3004003577425163 103.856...	0	2025-04-27 04:25:00 UTC	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
P0109	1.3845964153673076 103.943...	0	2025-04-27 04:25:00 UTC	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
M0059	1.3058536755285977 103.847...	3	2025-04-27 04:25:00 UTC	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3
C0119	1.3098549757016427 103.899...	4	2025-04-27 04:25:00 UTC	4	3	4	4	4	4	4	4	4	4	4	4	4	4	4
N0012	1.3060825735308885 103.863...	4	2025-04-27 04:25:00 UTC	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
M0084	1.2746496875172093 103.849...	8	2025-04-27 04:25:00 UTC	7	8	8	8	8	8	8	8	8	8	8	8	8	8	8
P0013	1.2751330286026736 103.848...	8	2025-04-27 04:25:00 UTC	7	8	8	8	8	8	8	8	8	8	8	8	8	8	8
A0046	1.3138928367737375 103.881...	10	2025-04-27 04:25:00 UTC	9	10	10	10	10	10	10	10	10	10	10	10	10	10	10
H0057	1.307959413362438 103.8544...	10	2025-04-27 04:25:00 UTC	9	9	9	10	10	10	10	10	10	10	10	10	10	10	10
S0108	1.3186024746953806 103.850...	11	2025-04-27 04:25:00 UTC	11	12	11	11	11	11	11	11	11	11	11	11	11	11	11
V0007	1.310324224867271 103.8561...	13	2025-04-27 04:25:00 UTC	13	13	13	11	11	11	11	11	11	11	12	12	13	13	13
L0116	1.312971928021 34 103.88051...	22	2025-04-27 04:25:00 UTC	22	23	21	11	11	11	11	11	11	18	20	20	22	22	22
H0015	1.3085901317745132 103.854...	12	2025-04-27 04:25:00 UTC	12	11	10	12	12	12	12	12	12	12	12	12	12	12	12
P0075	1.3099941806456645 103.862...	12	2025-04-27 04:25:00 UTC	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
T0017	1.2869316650017533 103.824...	14	2025-04-27 04:25:00 UTC	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
H0003	1.3116896699384313 103.861...	15	2025-04-27 04:25:00 UTC	14	15	14	15	15	15	15	15	15	15	15	15	15	15	15
H0024	1.2864010712092964 103.847...	15	2025-04-27 04:25:00 UTC	15	15	15	15	15	15	15	15	15	14	14	14	15	15	15
K0082	1.3060643999336945 103.867...	15	2025-04-27 04:25:00 UTC	14	16	15	15	15	15	15	15	15	15	15	15	15	15	15
N0006	1.2793383109971352 103.841...	15	2025-04-27 04:25:00 UTC	14	16	15	15	15	15	15	15	15	15	15	15	15	15	15
A0021	1.280790422979401 103.8471...	16	2025-04-27 04:25:00 UTC	15	16	18	15	15	15	15	15	15	15	16	16	16	16	16
S0020	1.3722654677464599 103.828...	16	2025-04-27 04:25:00 UTC	16	16	17	16	16	16	16	16	16	16	16	16	16	16	16

Figure 15 Vertex AI Pipeline Output

As usual, all relevant code for this section can be found in the appendix section.

11.4 Inference Pipeline Monitoring

To continuously monitor the performance of the car park availability prediction model, a BigQuery SQL script was developed. This script measures key evaluation metrics by comparing the predicted available lots (PredictedLots) with the actual observed available lots (actual_lots) after a 30-minute interval.

Key Steps Performed:

1. **Rank Actual Data Points:**
   - The RankedActuals CTE organizes the ground truth (actual_lots) data from the carpark_availability table.
   - It assigns a row number (rn) partitioned by CarParkID and minute-truncated timestamps to uniquely identify the nearest actual observation per minute.
2. **Align Predictions and Actuals:**
   - The prediction_vs_actual CTE matches each prediction (PredictedLots) from the view_carpark_predictions_30min_1 view to the corresponding ground truth (actual_lots).
   - Matching is done by:
     - Shifting the prediction timestamp by +30 minutes.
     - Truncating both the prediction and actual timestamps to the nearest minute.
     - Joining based on CarParkID and matched timestamps.
3. **Calculate Monitoring Metrics:**
   - **Monitoring Timestamp:** A fixed timestamp is recorded for when the monitoring snapshot is taken.
   - **Total Records:** The count of prediction-actual pairs evaluated.
   - **MAE (Mean Absolute Error):** Average of the absolute differences between predicted and actual lots.
   - **RMSE (Root Mean Square Error):** Square root of the average of squared differences between predicted and actual lots.

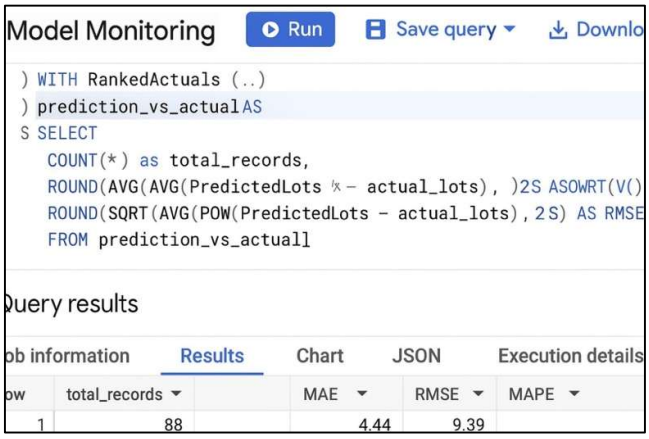

Figure 16 Model Monitoring

This monitoring query provides a snapshot of model prediction accuracy in production, enabling ongoing evaluation of model health and facilitating decisions for potential retraining or model adjustments.

12. Dashboard

This project delivers a comprehensive, interactive parking availability dashboard developed using Looker Studio, embedded with a Google Maps visualization. The dashboard provides real-time and predicted parking lot availability across major locations in Singapore, making it an essential tool for both urban mobility planning and daily commuter decision-making.

Each car park is represented as a bubble on the map. The size of the bubble indicates the current number of available parking lots — larger bubbles mean more available lots. Meanwhile, the color of the bubble transitions from red to green, representing the forecasted number of available lots 30 minutes into the future (red for low predicted availability, green for high predicted availability). This dual encoding of size and color makes it easy for users to quickly assess both present and future parking conditions at a glance.

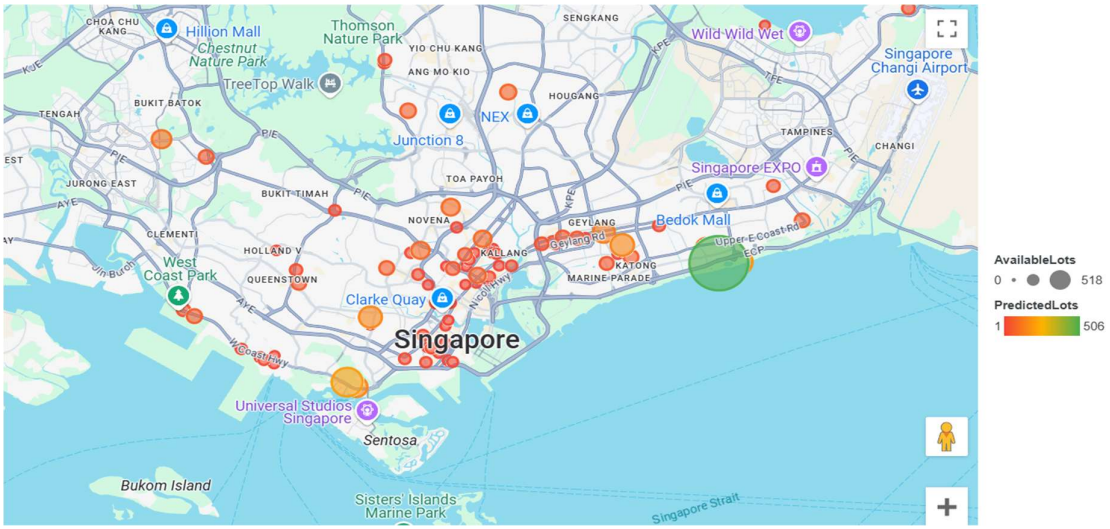

Figure 17 Dashboard

When users hover over any bubble, a tooltip displays important detailed information including:

- Car park name
- Current number of available lots
- Predicted number of lots after 30 minutes

The goal of this dashboard is to empower drivers with predictive insights to plan parking more efficiently, while also assisting city planners in identifying areas that may require parking expansion, better traffic management, or demand balancing strategies. This aligns with broader smart city initiatives focused on optimizing urban mobility and enhancing commuter experience.

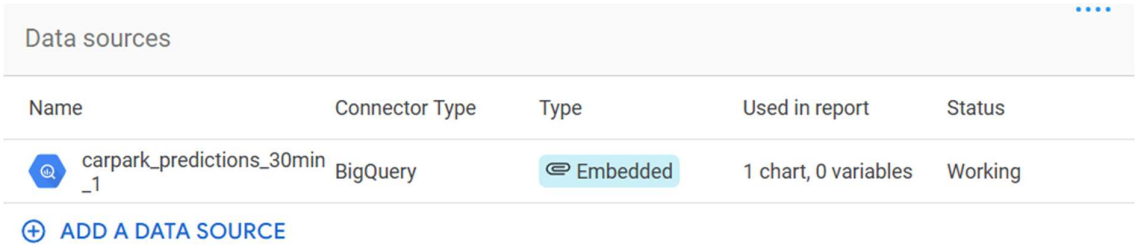

Figure 18 Data from Big Query to Looker Studio

The data powering this dashboard is sourced directly from Google BigQuery, where live and predictive parking data (table: carpark_predictions_30min) is stored and maintained. By connecting Looker Studio to BigQuery, the dashboard ensures seamless, near real-time data updates and scalability, offering reliable and timely information without manual intervention. This robust integration between data and visualization platforms makes the system highly adaptable and future-ready.

Overall, this solution demonstrates how predictive analytics combined with interactive visual tools can transform traditional urban challenges into data-driven, intelligent ecosystems.

13. Smart Routing

The Smart Parking Finder project integrates predictive analytics, real-time data, and interactive visualization to simplify urban parking across Singapore. It fetches parking availability predictions from Google BigQuery, processes the data through a Colab function to recommend nearby lots, and provides direct Google Maps navigation links for seamless routing.

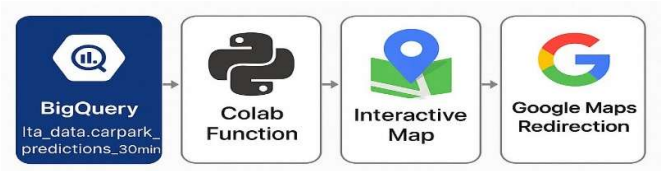

Figure 19 Smart Routing Dataflow

The solution centers around BigQuery-powered cloud queries. The fetch_parking() function retrieves predicted availability, filters parking lots based on available slots and distance (using the Haversine formula), and returns the nearest, most relevant options. This ensures recommendations are both timely and location-specific.

```
Function: fetch_parking(user_lat, user_lon, min_slots,
                        max_slots, max_distance, max_results)

1. Query BigQuery Table:
   `carpark_predictions_30min_1`
   - WHERE PredictedLots BETWEEN min_slots AND max_slots.
2. For each row:
   - Extract lat/lon from Location.
   - Calculate distance (Haversine) from user.
   - If distance ≤ max_distance → add to list.
3. Sort list by distance.
4. Return top N closest lots.
```

```
Function: show_map(user_lat, user_lon, parking_lots)

1. Create Folium map at user's coordinates.
2. Place markers:
   - 🟢 Best, 🟠 Second-Best, ⚪ Other lots.
3. Each marker shows:
   - Name, slots, distance, Google Maps link.
4. Add color-coded legend.
```

Figure 20 Smart Routing Code Snippet

The top parking spots are visualized using Folium on an interactive map. Markers highlight the best and second-best options, color-coded for quick understanding. Each marker displays key details: lot name, available slots, distance from the user, and a Google Maps navigation link for immediate directions.

Additionally, the Colab notebook outputs a simple text-based recommendation list, showing the user's coordinates, timestamp, and two top parking suggestions along with their respective distances and navigation links. This dual approach—textual and visual—caters to different user preferences for faster decision-making.

Sample Output :

📍 Your Coordinates: 1.3479 103.9583

🔎 Fetching parking recommendations...

✅ Top Recommendation:

🏢 UPPER CHANGI ROAD - BEDOK ROAD OFF STREET

🚗 Slots: 36

📏 Distance: 2.19 km

🗺️ https://www.google.com/maps/dir/1.3479,103.9583/1.3321242011794179,103.9464677456276/

Once the top parking recommendations are generated using BigQuery queries, the results are visualized on an interactive Folium map. The best parking option is marked with a green icon, and the second-best with an orange icon. Each marker displays key information such as the development name, predicted available slots, distance from the user's location, and a direct Google Maps link for navigation. Alongside the map, a text-based output lists the top recommendations with their location names, slot availability, distances, and navigation links, offering a simple and intuitive way for users to select and reach nearby parking spots.

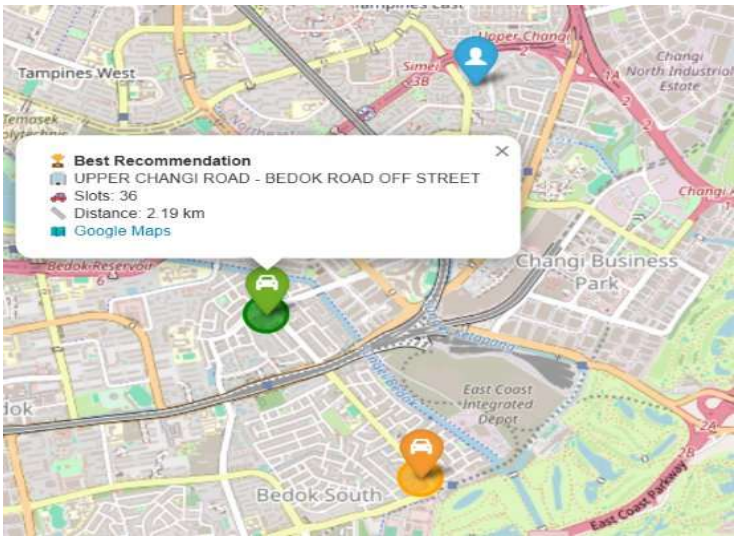

Figure 21 Smart Routing Output

This project demonstrates how real-time cloud data integration, predictive filtering, and geospatial visualization can together create a practical, user-friendly solution for urban parking challenges. By leveraging BigQuery for scalable data management, Colab for intelligent recommendations, and Folium for interactive mapping, the Smart Parking Finder enables faster parking decisions and improves the overall commuter experience in a dynamic city environment.

14. Results of Models/Key Deliverables/Key Outcomes

1. Two time series machine learning algorithms were built for this project.
    a. Gradient Boosted Tree (GBT) Regressor -SparkML
        i. Predict for available lots in 30 minutes
    b. Long Short-Term Memory (LSTM) network
        i. Predict for available lots in 30 minutes
        ii. Predict for available lots in 45 minutes
        iii. Predict for available lots in 60 minutes

2. GBT Regressor did not achieve the target improvement of >15% over the benchmark. However, the LSTM model managed to achieve that for predictions for available lots in 30, 45, and 60 minutes.
3. Big Data Platform Ingestion, Training and Inference Pipelines were built. 3 Inference pipelines were built while searching for the best solution. We have chosen to use the Vertex AI Kubeflow Inference Pipeline as the final inference pipeline.
4. Inference pipeline is monitored to detect for Model Decay.
5. An interactive dashboard and smart car routing system deliver timely feedback through a user-friendly interface. Predicted as well as current available lots are provided to the user.

15. Outcome Discussion & Analysis

The project have met all the targets and deliverables that we set out to achieve. Furthermore, we added 2 more predictions, available lots in 45 min and 1hour to give users more options. However, there is still plenty of room for improvement, which will be detailed in section 17.

16. Conclusions

This project successfully delivered a Big Data-ready, scalable pipeline for ingestion, training, and inference. The outcomes are then made actionable for users through an interactive dashboard and smart routing system.

1. Google Cloud Platform provided the necessary architecture for scalable Ingestion and Training pipelines.
2. Two time series models were used and the LSTM proved to be the most accurate for our application.
3. Various inference pipelines were explored to find the best solution that provides a accurate, scalable, and automated workflow. The Vertex AI Kubeflow Inference Pipeline was finally chosen
4. Interactive dashboard and smart routing system was developed to effectively utilize the results.

17. Future Work and Recommendations

The following future improvements could be done for the project:

1. Incorporate the weather and traffic incidence data collected to the training data,a s well as holiday data.
    - Include in final models if RMSE and MSE improves
    - Huddle: Data from these APIs are messy, thus a lot of preprocessing is required to ready the data.
2. Handle inconsistent data from API fetch
    1. Determine a solution to handle the carparks with minimal (<10%) missing values from API
    2. Potentially allowing for predictions of more carparks
3. Include motorcycles as well as heavy vehicles
    1. Build separate models for these categories
    2. Alternatively, include them in main model and check for accuracy
4. Collect more data for model training
    1. As no historical data was available, data for model training had to be collected over time
    2. Model would be more accurate when there is more data
5. Dashboard to show more predictions as well as existing trends
    1. Data for this has been package to Big Query
    2. A new dashboard can be easily made from this data

18. Acknowledgements

We would like to extend our sincere gratitude to our lecturers for their dedication and the generous amount of time they have invested in our education. Their patient guidance and the wealth of real-life industry examples they shared have significantly enriched our learning experience, helping us understand complex concepts and their practical applications.

19. References

[1] Land Transport Authority, Land Transport Authority Carpark Availability API. Available: https://datamall2.mytransport.sg/ltaodataservice/CarParkAvailabilityv2

20. Appendix/Annex

20.1 Data Collection from API

```
import requests
import schedule
import time
import os
from pymongo import MongoClient
from datetime import datetime

# Environment variables for API keys (Ensure you set them in your environment)
LTA_API_KEY = os.getenv("LTA_API_KEY", "7GQ4fcMqRTuEm4Tb681Y6A==")  # Replace
if not using environment variable

# API URLs
API_ENDPOINTS = {
    "carpark_availability":
"https://datamall2.mytransport.sg/ltaodataservice/CarParkAvailabilityv2",
    "traffic_incidents":
"https://datamall2.mytransport.sg/ltaodataservice/TrafficIncidents",
    "weather_forecast": "https://api-open.data.gov.sg/v2/real-time/api/twenty-
four-hr-forecast",
    "weather_forecast_2hrs": "https://api-open.data.gov.sg/v2/real-
time/api/two-hr-forecast"
}

# Headers for LTA APIs
HEADERS_LTA = {
    "AccountKey": LTA_API_KEY,
    "accept": "application/json"
}

# MongoDB Configuration
MONGO_URI = "mongodb://localhost:27017/"
DATABASE_NAME = "lta_data"

# Connect to MongoDB (Use a single connection for efficiency)
client = MongoClient(MONGO_URI)
db = client[DATABASE_NAME]

# Function to fetch data from an API
def fetch_data(api_name, url, headers=None):
    try:
        response = requests.get(url, headers=headers, timeout=20)
        #response = requests.get(url, headers=headers)
```

```
        if response.status_code == 200:
            data = response.json()
            return data.get("value", data)  # LTA APIs have "value", weather
API returns full JSON
        else:
            print(f"[ERROR] Failed to fetch data from {api_name}: HTTP
{response.status_code}")
            return None
    except Exception as e:
        print(f"[ERROR] Exception while fetching {api_name}: {e}")
        return None

# Function to store data in MongoDB and print total count
def store_data(collection_name, data):
    if not data:
        print(f"[WARNING] No data received for {collection_name}, skipping
storage.")
        return

    collection = db[collection_name]
    timestamp = datetime.now()

    # Add timestamp to each record and insert into MongoDB
    if isinstance(data, list):  # LTA APIs return lists
        for item in data:
            item["timestamp"] = timestamp
        collection.insert_many(data)
    else:  # Weather API returns a dictionary
        data["timestamp"] = timestamp
        collection.insert_one(data)

    # Get and print total count after insertion
    total_records = collection.count_documents({})
    print(f"[INFO] Stored {len(data) if isinstance(data, list) else 1} records
in {collection_name} at {timestamp}")
    print(f"[INFO] Total records in {collection_name}: {total_records}")

# Function to fetch and store data
def fetch_and_store_data():
    print("\n[INFO] Fetching latest data...")

    # Fetch and store data for each API
    carpark_data = fetch_data("Carpark Availability",
API_ENDPOINTS["carpark_availability"], HEADERS_LTA)
    store_data("carpark_availability", carpark_data)
```

```
    traffic_data = fetch_data("Traffic Incidents",
API_ENDPOINTS["traffic_incidents"], HEADERS_LTA)
    store_data("traffic_incidents", traffic_data)

    weather_data = fetch_data("Weather Forecast",
API_ENDPOINTS["weather_forecast"])
    store_data("weather_forecast", weather_data)

    weather_data_2hr = fetch_data("Weather Forecast 2hrs",
API_ENDPOINTS["weather_forecast_2hrs"])
    store_data("weather_forecast_2hrs", weather_data_2hr)

    print("[INFO] Data fetch complete.")

# Schedule the job every 5 minutes throughout the entire day (00:00 to 23:55)
for hour in range(24):  # Loop from 00 to 23 hours
    for minute in range(0, 60, 5):  # Loop from 0 to 55 minutes with a step of
5
        time_str = f"{hour:02d}:{minute:02d}"  # Format the time as HH:MM
        schedule.every().day.at(time_str).do(fetch_and_store_data)

# Main function to run the scheduled tasks
if __name__ == "__main__":
    while True:
        schedule.run_pending()  # Run any pending scheduled tasks
        time.sleep(1)  # Wait a second to avoid high CPU usage
```

20.2 Data Preprocessing Code

20.2.1 Data Merging

```
# -*- coding: utf-8 -*-
"""mergeCarparkDataset.ipynb

#1. IMPORTS
"""

import pandas as pd
import numpy as np


"""#2. Read CSVs"""

df_eswar = pd.read_csv("eswar/lta_data.carpark_availability.csv")  # Replace
with your actual file path
df_ian = pd.read_csv("ian/lta_data.carpark_availability.csv")  # Replace with
your actual file path
```

```
"""#2. Filter out for minute grid"""

min_grid = {'min': [0,5,10,15,20,25,30,35,40,45,50,55,]}

df_ian["timestamp"] = pd.to_datetime(df_ian["timestamp"])
df_ian['timestamp'] = df_ian['timestamp'].dt.round('min')
df_ian['min'] = df_ian['timestamp'].dt.minute
df_ian = df_ian[df_ian['min'].isin(min_grid['min'])]
df_ian = df_ian.reset_index(drop=True)

#df_ian["time_diff"] = df_ian['timestamp'].diff()
#df_ian["time_diff_min"] = df_ian["time_diff"].astype('int') // 10**9 // 60
#print(df_ian["time_diff_min"].value_counts())

df_eswar["timestamp"] = pd.to_datetime(df_eswar["timestamp"])
df_eswar['timestamp'] = df_eswar['timestamp'].dt.round('min')
df_eswar['min'] = df_eswar['timestamp'].dt.minute
df_eswar = df_eswar[df_eswar['min'].isin(min_grid['min'])]
df_eswar = df_eswar.reset_index(drop=True)

#df_eswar["time_diff"] = df_eswar['timestamp'].diff()
#df_eswar["time_diff_min"] = df_eswar["time_diff"].astype('int') // 10**9 //
60
#print(df_eswar["time_diff_min"].value_counts())

"""# 2b. Filter out only for cars and create unique id to search for"""

df_ian = df_ian[df_ian['LotType'] == 'C']
df_eswar = df_eswar[df_eswar['LotType'] == 'C']

df_ian['unique_id'] = df_ian['CarParkID'] + '_' +
df_ian['timestamp'].astype(str)
df_eswar['unique_id'] = df_eswar['CarParkID'] + '_' +
df_eswar['timestamp'].astype(str)

print(df_ian.shape)
unique_values = df_ian['unique_id'].unique()
print(unique_values.shape)

"""#3. Define Data Start to End Time and filter Dataframe too"""

#decide the start and end time
start_time = pd.to_datetime('2025-02-16 00:00:00').tz_localize('UTC')
end_time = pd.to_datetime('2025-03-16 00:00:00').tz_localize('UTC')

df_ian = df_ian[df_ian['timestamp'].between(start_time, end_time)]
df_eswar = df_eswar[df_eswar['timestamp'].between(start_time, end_time)]
```

```
all_timestamps = pd.date_range(start=start_time, end=end_time, freq='5min')
all_timestamps

"""# 3b. ALL possible Unique IDs"""

unique_cp = df_ian['CarParkID'].unique()
all_timestamps = all_timestamps.astype(str)
print(unique_cp)
print(all_timestamps)

all_UID = [a + "_" + b for a in unique_cp for b in all_timestamps]

#print(all_UID)

type(all_timestamps)

all_UID = pd.Index(all_UID)
type(all_UID)
all_UID

"""#4. Combine the datasets"""

missing_UID = all_UID[~all_UID.isin(df_ian['unique_id'])]
matching_rows = df_eswar[df_eswar['unique_id'].isin(missing_UID)]
# Step 2: Append the matching rows to df_ian
df_combined = pd.concat([df_ian, matching_rows], ignore_index=True)
# Step 3: Sort the new dataframe by timestamp
df_combined = df_combined.sort_values(by=['timestamp','unique_id'])
# Print the final result
df_combined = df_combined.reset_index(drop=True)
df_combined

df_combined["time_diff"] = df_combined['timestamp'].diff()
df_combined["time_diff_min"] = df_combined["time_diff"].astype('int') // 10**9
// 60
print(df_combined["time_diff_min"].value_counts())

"""# 5. Save to CSV"""

# Define the file path in Google Drive
file_path = '/content/gdrive/My
Drive/Proj5006/concat/carpark_availability.csv'

# Save DataFrame to CSV
df_combined.to_csv(file_path, index=False)
```

```
print(f"CSV file saved at: {file_path}")

"""# 6. Explore large gaps

#6a. Explore in original DFs
"""

df_ian["time_diff"] = df_ian['timestamp'].diff()
df_ian["time_diff_min"] = df_ian["time_diff"].astype('int') // 10**9 // 60
print(df_ian["time_diff_min"].value_counts())

df_eswar["time_diff"] = df_eswar['timestamp'].diff()
df_eswar["time_diff_min"] = df_eswar["time_diff"].astype('int') // 10**9 // 60
print(df_eswar["time_diff_min"].value_counts())

gaps_i = df_ian[(df_ian["time_diff_min"] != 5) & (df_ian["time_diff_min"] !=
0)]
gaps_i

gaps_e = df_eswar[(df_eswar["time_diff_min"] != 5) &
(df_eswar["time_diff_min"] != 0)]
gaps_e

"""#6b Explore in combined DFs"""

gaps = df_combined[(df_combined["time_diff_min"] != 5) &
(df_combined["time_diff_min"] != 0) ].index
gaps

df_combined.loc[gaps]

gaps = gaps[gaps != 0]  # Remove 0

# Step 2: Add (value - 1) for each remaining value
gaps = gaps.tolist()  # Convert to list if it's an Index object
gaps_extended = set(gaps + [x - 1 for x in gaps])  # Use set to avoid
duplicates

# Step 3: Sort the final list
gaps_sorted = sorted(gaps_extended)

# Print the final result
print(gaps_sorted)

df_missing = df_combined.loc[gaps_sorted]
df_missing
```

```
"""#6c Save to CSV"""

# Define the file path in Google Drive
file_path = '/content/gdrive/My
Drive/Proj5006/concat/missing_carpark_availability.csv'

# Save DataFrame to CSV
df_missing.to_csv(file_path, index=False)

print(f"CSV file saved at: {file_path}")
```

20.2.2 Explore Data Quality

```
# -*- coding: utf-8 -*-
"""ExploreMissingValuesCP.ipynb
"""

import pandas as pd
import numpy as np

import matplotlib.pyplot as plt

df_cp = pd.read_csv("concat/16febto30mar_carpark_availability.csv")

df_cp_missing =
pd.read_csv("concat/16febto30mar_missing_carpark_availability.csv")

gaps = df_cp[(df_cp["time_diff_min"] != 5) & (df_cp["time_diff_min"] != 0)
].index
gaps

df_cp.loc[gaps]

df_cp['timestamp'].unique().size

df_cp['timestamp'].value_counts()

df_cp['CarParkID'].unique().size

value_counts= df_cp['CarParkID'].value_counts()

# Define the file path in Google Drive
file_path = '/content/gdrive/My
Drive/Proj5006/concat/16febto30mar_carparks_value_count.csv'

# Save DataFrame to CSV
value_counts.to_csv(file_path)
```

```
print(f"CSV file saved at: {file_path}")

with pd.option_context('display.max_rows', None, 'display.max_columns', None):
    print(value_counts)
```

20.2.3 Data Imputation using Linear Interpolation

```
# -*- coding: utf-8 -*-
"""impute_carpark_missing.ipynb

# 1. Imports and load data
"""

import pandas as pd
import numpy as np

df_cp = pd.read_csv("concat/16febto15mar_carpark_availability.csv")
df_cp_missing =
pd.read_csv("concat/16febto30mar_missing_carpark_availability.csv")

"""# 2. Find the missing and generate the missing date times"""

df_cp_missing = df_cp_missing[df_cp_missing['time_diff_min']!= 0]

df_cp_missing['missing_times'] = (df_cp_missing['time_diff_min'] - 5) / 5

df_cp_missing

def generate_and_sort_previous_timestamps(timestamp_str, x):
    """
    Generates a list of x timestamps at 5-minute intervals before the given
timestamp,
    and returns them in chronological (ascending) order.

    Args:
        timestamp_str: A string representing the timestamp (e.g., "2025-02-17
00:30:00+00:00").
        x: The number of timestamps to generate.

    Returns:
        A sorted list of timestamp strings, or an error message if the input
is invalid.
    """
    try:
        timestamp = pd.to_datetime(timestamp_str)
        previous_timestamps = []
        for i in range(1, x + 1):
```

```
            previous_timestamp = timestamp - pd.Timedelta(minutes=5 * i)
            previous_timestamps.append(str(previous_timestamp))

        # Sort the timestamps chronologically (ascending order)
        sorted_timestamps = sorted(previous_timestamps)
        return sorted_timestamps

    except ValueError:
        return "Invalid timestamp format"
    except TypeError:
        return "Invalid number of timestamps requested"

results = []

df_cp_missing['missing_times'] = df_cp_missing['missing_times'].astype(int)

for index, row in df_cp_missing.iterrows():
    timestamp_str = row['timestamp']
    x = row['missing_times']
    result = generate_and_sort_previous_timestamps(timestamp_str, x)
    results.append(result)

results

"""# 3. Filter out URA and create CARPARK DF Dictionary"""

df_cp_URA = df_cp[df_cp['Agency'] == 'URA']

df_cp_URA.shape

# List of columns to drop
columns_to_drop = ['_id', 'Area', 'Development', 'Location', 'LotType',
'Agency' ]

# Drop the columns
df_cp_URA = df_cp_URA.drop(columns=columns_to_drop)

df_cp_URA

df_cp_URA.reset_index(drop=True, inplace=True)

df_cp_URA.isna().sum()

dfs = {carpark: df_cp_URA[df_cp_URA["CarParkID"] == carpark] for carpark in
df_cp_URA["CarParkID"].unique()}

cp_list = df_cp_URA["CarParkID"].unique()
```

```
def append_missing_timestamps_to_dataframes_all(dataframes_dict,
missing_timestamps_list):
    """
    Appends the same list of lists of missing timestamps to every DataFrame in
the dictionary.

    Args:
        dataframes_dict: A dictionary where keys are IDs and values are pandas
DataFrames.
        missing_timestamps_list: A list of lists, where each inner list
contains missing timestamps (strings).

    Returns:
        A dictionary of DataFrames with appended missing timestamps.
    """

    modified_dataframes_dict = dataframes_dict.copy()

    for id_val, df in modified_dataframes_dict.items():
        df_copy = df.copy()  # Create a copy to avoid modifying the original
DataFrame
        # Convert 'timestamp' column to datetime objects before appending new
timestamps
        df_copy['timestamp'] = pd.to_datetime(df_copy['timestamp'])

        for missing_times in missing_timestamps_list:
            if missing_times: #Check if the list is empty
                missing_df = pd.DataFrame({
                    'CarParkID': df_copy['CarParkID'].iloc[0],
                    'timestamp': pd.to_datetime(missing_times), # Convert to
Timestamp here
                    'AvailableLots': pd.NA
                })

                df_copy = pd.concat([df_copy, missing_df], ignore_index=True)

        modified_dataframes_dict[id_val] =
df_copy.sort_values(by='timestamp').reset_index(drop=True)

    return modified_dataframes_dict

def available_lots_interpolate(dataframes_dict):
    """
    Converts the 'AvailableLots' column to numeric for each DataFrame in the
dictionary.
```

```
    Args:
        dataframes_dict: A dictionary where keys are IDs and values are pandas
DataFrames.

    Returns:
        A dictionary of DataFrames with 'AvailableLots' converted to numeric.
    """
    modified_dataframes_dict = dataframes_dict.copy()

    for id_val, df in modified_dataframes_dict.items():
        if 'AvailableLots' in df.columns:  # Check if the column exists
            df['AvailableLots'] = pd.to_numeric(df['AvailableLots'],
errors='coerce')
            df['AvailableLots'] =
df['AvailableLots'].interpolate(method='linear')
            modified_dataframes_dict[id_val] = df

    return modified_dataframes_dict

"""# START HERE?"""

dfs_ts = append_missing_timestamps_to_dataframes_all(dfs, results)
dfs_inter = available_lots_interpolate(dfs_ts)

dfs_inter['A0007']

dfs_inter['A0007'].isna().sum()

def flatten_dataframe_dict(dataframes_dict):
    """
    Flattens a dictionary of DataFrames into a single DataFrame.

    Args:
        dataframes_dict: A dictionary where keys are IDs and values are pandas
DataFrames.

    Returns:
        A single pandas DataFrame containing all the data from the dictionary.
    """
    all_dataframes = []
    for id_val, df in dataframes_dict.items():
        df_copy = df.copy() #make copy to avoid modifying original.
        all_dataframes.append(df_copy)

    if all_dataframes: #Check if the list is empty
        combined_df = pd.concat(all_dataframes, ignore_index=True)
        return combined_df
```

```
    else:
        return pd.DataFrame() #return empty dataframe if the dictionary is
empty

df_done = flatten_dataframe_dict(dfs_inter)

"""# Ensure DF _DONE is done

"""

# Step 3: Sort the new dataframe by timestamp
df_done = df_done.sort_values(by=['timestamp','unique_id'])
# Print the final result
df_done = df_done.reset_index(drop=True)

df_done["time_diff"] = df_done['timestamp'].diff()
df_done["time_diff_min"] = df_done["time_diff"].astype('int') // 10**9 // 60
print(df_done["time_diff_min"].value_counts())

df_done.isna().sum()

"""# SAVE TO CSV

"""

df_done

# List of columns to drop
columns_to_drop = ['min', 'time_diff', 'time_diff_min']

# Drop the columns
df_done_drop = df_done.drop(columns=columns_to_drop)

df_done_drop

df_done_drop['CarParkID'].value_counts()

# Define the file path in Google Drive
file_path = '/content/gdrive/My
Drive/Proj5006/concat/16febto30mar_carpark_availability_imputed.csv'

# Save DataFrame to CSV
df_done_drop.to_csv(file_path, index=False)

print(f"CSV file saved at: {file_path}")
```

20.3 Model Training Codes

20.3.1 LSTM Model Training and Evaluation

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler, LabelEncoder

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Embedding, Concatenate, Input
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Model
from sklearn.model_selection import train_test_split

import tensorflow as tf
print(tf.__version__)

import matplotlib.pyplot as plt
import pickle

import os

# ================================
#  ◇ Step 1: Load & Preprocess Data
# ================================
df = pd.read_csv("16febto30mar_carpark_availability_final.csv")  # replace
with your actual file name

# Convert to datetime
df['timestamp'] = pd.to_datetime(df['timestamp'])

# Extract features
df['Min'] = df['timestamp'].dt.minute
df['Hour'] = df['timestamp'].dt.hour
df['DayOfWeek'] = df['timestamp'].dt.dayofweek


# Encode CarParkID
le = LabelEncoder()
df['CarParkEncoded'] = le.fit_transform(df['CarParkID'])

# Sort for time sequence modeling
df.sort_values(['CarParkEncoded', 'timestamp'], inplace=True)


# ================================
#  ◇ Step 2: Normalize Features
# ================================
```

```
scaler_AL = MinMaxScaler()
scalar_CP = MinMaxScaler()
scaler_temp = MinMaxScaler()
#df_f[['Hour', 'DayOfWeek', 'AvailableLots']] =
scaler.fit_transform(df_f[['Hour', 'DayOfWeek', 'AvailableLots']])
df['AvailableLots_MM'] = scaler_AL.fit_transform(df[['AvailableLots']])
df['CarParkEncoded_MM'] = scalar_CP.fit_transform(df[['CarParkEncoded']])
df[['Min_MM', 'Hour_MM', 'DayOfWeek_MM']] =
scaler_temp.fit_transform(df[['Min','Hour', 'DayOfWeek']])


# Save LabelEncoder
with open('label_encoder_Carparks.pkl', 'wb') as file:
    pickle.dump(le, file)

# Save MinMaxScaler
with open('min_max_scaler_AvailableLots.pkl', 'wb') as file:
    pickle.dump(scaler_AL, file)

with open('min_max_scaler_Carparks.pkl', 'wb') as file:
    pickle.dump(scalar_CP, file)

with open('min_max_scaler_Min_Hour_Day.pkl', 'wb') as file:
    pickle.dump(scaler_temp, file)

print("LabelEncoder and MinMaxScaler saved successfully!")


def check_value_counts_equal(series):
    """
    Checks if all values in a Pandas Series' value_counts() are equal.

    Args:
        series: A Pandas Series.

    Returns:
        True if all value counts are equal, False otherwise.
    """
    value_counts = series.value_counts()
    if len(value_counts) <= 1: #if there is only one unique value, or zero,
then they are equal.
        return True
    else:
        return (value_counts == value_counts.iloc[0]).all() #compare all to
the first value.

check_value_counts_equal(df['CarParkID'])
```

```
df['CarParkID'].value_counts().iloc[0]

"""# Put into dictionaries so that train test set will contain every
carpark"""

dfs = {carpark: df[df["CarParkID"] == carpark] for carpark in
df["CarParkID"].unique()}

'''
# ================================
#  ◇  Step 3: Create Sequences for LSTM
# ================================

def create_sequences_per_location(data_dict, sequence_length=5,
test_ratio=0.2):
    X_train, y_train = [], []
    X_test, y_test = [], []

    for location, df in data_dict.items():
        # Assuming df is sorted by time already
        values = df[['CarParkEncoded_MM', 'Min_MM', 'Hour_MM', 'DayOfWeek_MM',
'AvailableLots_MM']].values

        X_loc, y_loc = [], []

        for i in range(len(values) - sequence_length):
            seq = values[i:i + sequence_length]
            label = values[i + sequence_length][4]  # AvailableLots
            X_loc.append(seq)
            y_loc.append(label)

        # Split within this location
        split_idx = int(len(X_loc) * (1 - test_ratio))
        X_train += X_loc[:split_idx]
        y_train += y_loc[:split_idx]
        X_test += X_loc[split_idx:]
        y_test += y_loc[split_idx:]

    return np.array(X_train), np.array(X_test), np.array(y_train),
np.array(y_test)
'''

# ================================
#  ◇  Step 3: Create Sequences for LSTM
# ================================
```

```
"""# 30 mins version"""
def create_sequences_per_location_30minlag(data_dict, sequence_length=5,
test_ratio=0.2):
    X_train, y_train = [], []
    X_test, y_test = [], []

    for location, df in data_dict.items():
        # Assuming df is sorted by time already
        values = df[['CarParkEncoded_MM', 'Min_MM', 'Hour_MM', 'DayOfWeek_MM',
'AvailableLots_MM']].values

        X_loc, y_loc = [], []

        for i in range(len(values) - sequence_length - 5): # need to
compenstate here with minus 5
            seq = values[i:i + sequence_length]
            label = values[i + sequence_length + 5][4]  # AvailableLots (plus
5 is 30mins vs + 0 is 5mins)
            X_loc.append(seq)
            y_loc.append(label)

        # Split within this location
        split_idx = int(len(X_loc) * (1 - test_ratio))
        X_train += X_loc[:split_idx]
        y_train += y_loc[:split_idx]
        X_test += X_loc[split_idx:]
        y_test += y_loc[split_idx:]

    return np.array(X_train), np.array(X_test), np.array(y_train),
np.array(y_test)

def create_sequences_per_location_45minlag(data_dict, sequence_length=5,
test_ratio=0.2):
    X_train, y_train = [], []
    X_test, y_test = [], []

    for location, df in data_dict.items():
        # Assuming df is sorted by time already
        values = df[['CarParkEncoded_MM', 'Min_MM', 'Hour_MM', 'DayOfWeek_MM',
'AvailableLots_MM']].values

        X_loc, y_loc = [], []

        for i in range(len(values) - sequence_length - 8): # need to
compenstate here with minus 5
            seq = values[i:i + sequence_length]
```

```
            label = values[i + sequence_length + 8][4]  # AvailableLots (plus
5 is 30mins vs + 0 is 5mins)
            X_loc.append(seq)
            y_loc.append(label)

        # Split within this location
        split_idx = int(len(X_loc) * (1 - test_ratio))
        X_train += X_loc[:split_idx]
        y_train += y_loc[:split_idx]
        X_test += X_loc[split_idx:]
        y_test += y_loc[split_idx:]

    return np.array(X_train), np.array(X_test), np.array(y_train),
np.array(y_test)

def create_sequences_per_location_60minlag(data_dict, sequence_length=5,
test_ratio=0.2):
    X_train, y_train = [], []
    X_test, y_test = [], []

    for location, df in data_dict.items():
        # Assuming df is sorted by time already
        values = df[['CarParkEncoded_MM', 'Min_MM', 'Hour_MM', 'DayOfWeek_MM',
'AvailableLots_MM']].values

        X_loc, y_loc = [], []

        for i in range(len(values) - sequence_length - 11): # need to
compenstate here with minus 5
            seq = values[i:i + sequence_length]
            label = values[i + sequence_length + 11[4]  # AvailableLots (plus
5 is 30mins vs + 0 is 5mins)
            X_loc.append(seq)
            y_loc.append(label)

        # Split within this location
        split_idx = int(len(X_loc) * (1 - test_ratio))
        X_train += X_loc[:split_idx]
        y_train += y_loc[:split_idx]
        X_test += X_loc[split_idx:]
        y_test += y_loc[split_idx:]

    return np.array(X_train), np.array(X_test), np.array(y_train),
np.array(y_test)
```

```
X_train, X_test, y_train, y_test = create_sequences_per_location_30minlag(dfs,
sequence_length=5, test_ratio=0.2)

print(X_train.shape)
print (y_train.shape)
print(X_test.shape)
print (y_test.shape)


# ==============================
#  ◇ Step 4: Build & Train LSTM Model
# ==============================
model = Sequential()
model.add(LSTM(64, activation='tanh', input_shape=(X_train.shape[1],
X_train.shape[2])))
model.add(Dense(32, activation="relu"))
model.add(Dense(16, activation="relu"))
model.add(Dense(8, activation="relu"))
model.add(Dense(4, activation="relu"))
model.add(Dense(1, activation="sigmoid"))
model.compile(optimizer='adam', loss='mse')
model.summary()

hist = model.fit(X_train, y_train, epochs=1000, batch_size=5096,
validation_data=(X_test, y_test))

def plot_loss(history, start_epoch=0, end_epoch=None):
    """Plots the training and validation loss from a Keras history object for
a specified epoch range."""

    loss = history.history['loss']
    val_loss = history.history['val_loss']

    if end_epoch is None:
        end_epoch = len(loss)  # Plot all epochs if end_epoch is not specified

    epochs = range(start_epoch, end_epoch)

    plt.plot(epochs, loss[start_epoch:end_epoch], label='loss')
    plt.plot(epochs, val_loss[start_epoch:end_epoch], label='val_loss')

    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()
```

```
plot_loss(hist, 100)

preds = model.predict(X_test)

len(y_test)

from sklearn.metrics import mean_squared_error
from sklearn.metrics import root_mean_squared_error

y_test = y_test.reshape(len(y_test), 1)

preds_t = scaler_AL.inverse_transform(preds)
y_test_t = scaler_AL.inverse_transform(y_test)

MSE_N = mean_squared_error(y_test, preds)
#print(MSE_N)
MSE = mean_squared_error(y_test_t, preds_t)
print("MSE value is: ", MSE)
RMSE_N = root_mean_squared_error(y_test, preds)
#print(RMSE_N)
RMSE = root_mean_squared_error(y_test_t, preds_t)
print("RMSE value is: ", RMSE)

"""# BASELINE MODEL"""


# Extract the last element of the inner (5, 5) arrays
y_benchmark = X_test[:, -1, -1]

# Reshape the result to (143379, 1)
y_benchmark = y_benchmark.reshape(-1, 1)

y_benchmark_t = scaler_AL.inverse_transform(y_benchmark)

MSE = mean_squared_error(y_test_t, y_benchmark_t)
print(MSE)

RMSE = root_mean_squared_error(y_test_t, y_benchmark_t)
print(RMSE)


"""# Plot predictions

"""

num_CP= len(df['CarParkEncoded_MM'].unique())
```

```
y_test_t_reshape = y_test_t.reshape(y_test_t.shape[0],)
y_test_t_reshape = y_test_t_reshape.reshape(num_CP,
int(y_test_t.shape[0]/num_CP))
y_test_t_reshape.shape

preds_t_reshape = preds_t.reshape(preds_t.shape[0],)
preds_t_reshape = preds_t_reshape.reshape(num_CP,
int(preds_t.shape[0]/num_CP))
preds_t_reshape.shape

def plot_test_vs_pred(y_test, y_pred, title="Test vs. Predicted"):
    """
    Plots y_test and y_pred in the same graph for comparison.

    Args:
        y_test (array-like): Test target values.
        y_pred (array-like): Predicted values.
        title (str): Title of the plot.
    """

    plt.figure(figsize=(12, 6))  # Adjust figure size as needed

    plt.plot(y_test, label='y_test', marker='o')  # Plot y_test with markers
    plt.plot(y_pred, label='y_pred', marker='x')  # Plot y_pred with markers

    plt.title(title)
    plt.xlabel('Data Points (Test Set)')
    plt.ylabel('Values')
    plt.legend()
    plt.grid(True)
    plt.show()

CP = df['CarParkID'].unique()

for i in range(num_CP):
  plot_test_vs_pred(pd.Series(y_test_t_reshape[i]).tail(400),
pd.Series(preds_t_reshape[i]).tail(400),CP[i])


"""# Save Model"""

model.save('LSTM_Model_26APR_30mins.keras')
```

20.3.2 GBT Regression Model Training

```
import pandas as pd
import numpy as np

from pyspark.sql import SparkSession

import os

from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import GBTRegressor
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

# Start Spark session

spark = SparkSession.builder.appName("GBTRegressor") \
    .config("spark.driver.memory", "128g") \
    .getOrCreate()

# === Step 1: Load NumPy arrays ===
X_train = np.load("X_train.npy")  # shape (573516, 5, 5)
y_train = np.load("y_train.npy")  # shape (573516,)
X_test = np.load("X_test.npy")    # shape (143379, 5, 5)
y_test = np.load("y_test.npy")    # shape (143379,)

# === Step 2: Flatten 5x5 matrices to 25-element vectors ===
X_train_flat = X_train.reshape(X_train.shape[0], -1)
X_test_flat = X_test.reshape(X_test.shape[0], -1)

# === Step 3: Convert to pandas DataFrame ===
feature_cols = [f"f{i}" for i in range(X_train_flat.shape[1])]

df_train_pd = pd.DataFrame(X_train_flat, columns=feature_cols)
df_train_pd["label"] = y_train

df_test_pd = pd.DataFrame(X_test_flat, columns=feature_cols)
df_test_pd["label"] = y_test

# === Step 4: Convert to Spark DataFrame ===
df_train = spark.createDataFrame(df_train_pd)
df_test = spark.createDataFrame(df_test_pd)

# === Step 5: Assemble features ===
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
```

```
train_prepared = assembler.transform(df_train)
test_prepared = assembler.transform(df_test)


# === Step 6: Train GBT Regressor ===
gbt = GBTRegressor(featuresCol="features", labelCol="label", maxIter = 250,
maxDepth = 10, stepSize = 0.01)
model = gbt.fit(train_prepared)



# === Step 7: Predict and evaluate ===
predictions = model.transform(test_prepared)
evaluator = RegressionEvaluator(labelCol="label", predictionCol="prediction",
metricName="rmse")
rmse = evaluator.evaluate(predictions)

print(f"RMSE on test set: {rmse:.4f}")

predictions.select("label", "prediction") \
    .toPandas() \
    .to_csv("predictions_GBT_25Apr.csv", index=False)

model.write().overwrite().save("gbt_model_25Apr")

# Stop the Spark Session
spark.stop()
```

20.3.3 GBT Regression Model Evaluation

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import pickle

import os

# ================================
#  ◇ Step 1: Load & Preprocess Data
# ================================
df = pd.read_csv("16febto15mar_carpark_availability_final.csv")  # replace
with your actual file name
#df = pd.read_csv("5thApril.csv")

pred_df = pd.read_csv("predictions_GBT_25Apr.csv")

pred_df
```

```
# Load MinMaxScaler
with open('4APR_LSTM_Model/min_max_scaler_AvailableLots.pkl', 'rb') as file:
    scaler_AL = pickle.load(file)

pred_df['label_actual'] = scaler_AL.inverse_transform(pred_df[['label']])

pred_df['pred_actual'] = scaler_AL.inverse_transform(pred_df[['prediction']])

from sklearn.metrics import mean_squared_error
from sklearn.metrics import root_mean_squared_error

MSE_N = mean_squared_error(pred_df['label'], pred_df['prediction'])
#print(MSE_N)
MSE = mean_squared_error(pred_df['label_actual'], pred_df['pred_actual'])
print("MSE Value is: ", MSE)
RMSE_N = root_mean_squared_error(pred_df['label'], pred_df['prediction'])
#print(RMSE_N)
RMSE = root_mean_squared_error(pred_df['label_actual'],
pred_df['pred_actual'])
print("RMSE Value is: ",RMSE)

y_test_t = pred_df['label_actual']
preds_t = pred_df['pred_actual']

y_test_t = y_test_t.to_numpy()
preds_t = preds_t.to_numpy()

"""# Plot predictions

"""
num_CP= len(df['CarParkID'].unique())

CP = df['CarParkID'].unique()

y_test_t_reshape = y_test_t.reshape(y_test_t.shape[0],)
y_test_t_reshape = y_test_t_reshape.reshape(num_CP,
int(y_test_t.shape[0]/num_CP))
y_test_t_reshape.shape

preds_t_reshape = preds_t.reshape(preds_t.shape[0],)
preds_t_reshape = preds_t_reshape.reshape(num_CP,
int(preds_t.shape[0]/num_CP))
preds_t_reshape.shape

def plot_test_vs_pred(y_test, y_pred, title="Test vs. Predicted"):
    """
```

```
    Plots y_test and y_pred in the same graph for comparison.

    Args:
        y_test (array-like): Test target values.
        y_pred (array-like): Predicted values.
        title (str): Title of the plot.
    """

    plt.figure(figsize=(12, 6))  # Adjust figure size as needed

    plt.plot(y_test, label='y_test', marker='o')  # Plot y_test with markers
    plt.plot(y_pred, label='y_pred', marker='x')  # Plot y_pred with markers

    plt.title(title)
    plt.xlabel('Data Points (Test Set)')
    plt.ylabel('Values')
    plt.legend()
    plt.grid(True)
    plt.show()

for i in range(num_CP):
  plot_test_vs_pred(pd.Series(y_test_t_reshape[i]).tail(400),
pd.Series(preds_t_reshape[i]).tail(400),CP[i])
```

20.4 PySpark Inference Pipeline

20.4.1 PySpark Inference Pipeline code

```
import os
import pickle

from datetime import datetime
from pyspark.sql.types import TimestampType

from pyspark.sql import SparkSession, functions as F
from pyspark.sql.types import *

import sys
from pyspark.sql.functions import to_json, struct, col, lit
from pyspark.sql.functions import monotonically_increasing_id
from pyspark.sql.functions import round
from pyspark.sql.types import IntegerType

from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import GBTRegressor
```

```
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.regression import GBTRegressionModel

from pyspark.sql.functions import udf
from pyspark.sql.types import ArrayType, FloatType
from pyspark.ml.linalg import Vectors, VectorUDT

from pyspark.sql.functions import lit
from pyspark.sql import Row
from pyspark.sql.functions import max as spark_max, expr

spark =
SparkSession.builder.appName("carpark_prediction_pipeline").getOrCreate()

my_project_id = "involuted-forge-456406-a1"  # this is my projectid
bigquery_dataset_input = "smart-car-park-availability-
1.lta_data.view_carpark_availability_for_prediction"
bigquery_dataset_output = "smart-car-park-availability-
1.lta_data.carpark_predictions_30min_1"

temp_bucket = "gs://dataproc-temp-us-central1-773964370565-z5wnonnu/"
pkl_file_paths =
["gs://carpark_predict_models/4april2025/label_encoder_Carparks.pkl",
"gs://carpark_predict_models/4april2025/min_max_scaler_AvailableLots.pkl"
,"gs://carpark_predict_models/4april2025/min_max_scaler_Carparks.pkl",
"gs://carpark_predict_models/4april2025/min_max_scaler_Min_Hour_Day.pkl"]
model_dir = "gs://carpark_predict_models/GBT_15Apr_First"
output_dir = "gs://carpark_predict_temp/spark_predictions"

# DECLARE SEQUENCE LENGTH CONSTANT
#########################################################
SEQUENCE_LENGTH = 5

##### PART ONE: LOAD AND CHECK
#############################################################################################
####################################

# FUNCTIONS USED
#########################################################
def check_value_counts_equal_pyspark(df, col_name):
    """
    Checks if all value counts in a PySpark DataFrame column are equal.
    """
    counts = df.groupBy(col_name).count()
    first_count = counts.collect()[0]['count']  # Get the first count
    return counts.filter(F.col('count') != first_count).count() == 0  # Check
if any counts differ
```

```
# GET FILE FROM BIGQUERY
#####################################################
df = spark.read.format("bigquery") \
    .option("table", bigquery_dataset_input) \
    .option("viewsEnabled", "true") \
    .option("temporaryGcsBucket", temp_bucket) \
    .load() \
    .filter(F.col("timestamp") > F.current_timestamp() + F.expr("INTERVAL 454
MINUTES"))

# SORT AND CHECK the data
#############################################################
df = df.orderBy("CarParkID", "timestamp")

if not check_value_counts_equal_pyspark(df, 'CarParkID'):
    #print("CarParkID is not equal. Don't continue. Trigger an error")
    raise RuntimeError("CarParkID is not equal. Aborting the job.")
else:
    #print("CarParkEncoded is equal!")
    counts = df.groupBy('CarParkID').count()
    if counts.agg(F.max('count')).collect()[0][0] != SEQUENCE_LENGTH:
        #print("Don't have Sequence Length of Data Input. Don't continue.
Trigger an error")
        raise RuntimeError("Don't have Sequence Length of Data Input. Aborting
the job.")

#    else:
        #print("Have right amount of data to proceed. Data Preprocessing
Completed Successfully")


##### PART TWO: PREPROCESS
#########################################################################################
####################################

# FUNCTIONS USED
#####################################################
def create_sequences_per_location_pyspark(df, sequence_length=5):
    """
    Creates sequences from a PySpark DataFrame.
    This is a simplified version; efficient windowing in PySpark is complex.
    """
    # Define the columns to use for the 3D array
    selected_columns = ['CarParkEncoded_MM', 'Min_MM', 'Hour_MM',
'DayOfWeek_MM', 'AvailableLots_MM']
```

```
    # Group by 'CarParkEncoded' and collect into lists
#    grouped_df =
df.groupBy('CarParkEncoded').agg(F.collect_list(F.array(*selected_columns)).al
ias('sequences'))
    grouped_df =
df.groupBy('CarParkID').agg(F.collect_list(F.array(*selected_columns)).alias('
sequences'))

    def check_sequence_length(sequences):
        for seq in sequences:
            if len(seq) < sequence_length:
                return False
        return True

    # Create a UDF to check sequence lengths
    check_length_udf = F.udf(check_sequence_length, BooleanType())

    # Check if all sequences have the correct length
    sequence_done_flag = grouped_df.withColumn('all_sequences_valid',
check_length_udf(F.col('sequences'))).select(F.max('all_sequences_valid')).col
lect()[0][0]

    if not sequence_done_flag:
        return None, False
    else:
        return grouped_df, True

def load_pickles_to_driver(pkl_file_paths):
    """
    Loads pickle files to the driver node.
    WARNING: This is suitable for small pickle files only!
    For large files, distribute them or use Spark's broadcast.
    """
    loaded_objects = []
    from google.cloud import storage
    client = storage.Client()
    for pkl_path in pkl_file_paths:
        bucket_name = pkl_path.split('/')[2]
        blob_name = '/'.join(pkl_path.split('/')[3:])
        bucket = client.bucket(bucket_name)
        blob = bucket.blob(blob_name)
        blob_data = blob.download_as_bytes()
        loaded_objects.append(pickle.loads(blob_data))
    return loaded_objects

# OPEN MODEL SCALARS AND ENCODERS
#########################################################
```

```
le, scaler_AL, scalar_CP, scaler_temp = load_pickles_to_driver(pkl_file_paths)

# DATA PREPROCESSING
#########################################################
df = df.withColumn('timestamp', F.to_timestamp('timestamp'))

# FEATURE EXTRACTION/ ENCODING/ NORMALISATION
# Extract features
df = df.withColumn('Min', F.minute('timestamp')) \
       .withColumn('Hour', F.hour('timestamp')) \
       .withColumn('DayOfWeek', F.dayofweek('timestamp') - 1) # Adjust to
match Pandas (0-6)

# Encode CarParkID (UDF for complex logic)
def encode_carpark_udf(carpark_id):
    try:
        return int(le.transform([carpark_id])[0])
    except ValueError:
        return -1  # Or handle unknown IDs as needed

encode_carpark_spark_udf = F.udf(encode_carpark_udf, IntegerType())
df = df.withColumn('CarParkEncoded', encode_carpark_spark_udf('CarParkID'))

# Sort for time sequence modeling
df = df.orderBy('CarParkEncoded', 'timestamp')

# NORMALISE (UDFs for scaling)
def scale_al_udf(al):
    return float(scaler_AL.transform([[al]])[0][0])

scale_al_spark_udf = F.udf(scale_al_udf, FloatType())
df = df.withColumn('AvailableLots_MM', scale_al_spark_udf('AvailableLots'))

def scale_cp_udf(cp):
    return float(scalar_CP.transform([[cp]])[0][0])

scale_cp_spark_udf = F.udf(scale_cp_udf, FloatType())
df = df.withColumn('CarParkEncoded_MM', scale_cp_spark_udf('CarParkEncoded'))

def scale_time_udf(min, hour, dayofweek):
    scaled_values = scaler_temp.transform([[min, hour, dayofweek]])[0]
    return scaled_values.tolist()

scale_time_spark_udf = F.udf(scale_time_udf, ArrayType(FloatType()))
df = df.withColumn('scaled_time', scale_time_spark_udf('Min', 'Hour',
'DayOfWeek'))
df = df.withColumn('Min_MM', F.col('scaled_time').getItem(0)) \
```

```
        .withColumn('Hour_MM', F.col('scaled_time').getItem(1)) \
        .withColumn('DayOfWeek_MM', F.col('scaled_time').getItem(2))

# LSTM SEQUENCING
#########################################################
model_input, sequence_done_flag = create_sequences_per_location_pyspark(df,
SEQUENCE_LENGTH)

# Extra check. If fail don't proceed to predictions. Need to add some code
here
if not sequence_done_flag:
    raise RuntimeError("Not every Carpark has Sequence Length entries.
Aborting the job.")
#    print("ERROR! Not every Carpark has Sequence Length entries. Raise
Error")
#else:
#    print("Sequencing Done!")

model_input = model_input.orderBy("CarParkID")


##### PART THREE: PREDICT
#########################################################################################
###################################

# === Step 1: "Load from Parquet" ===
df_cp = model_input.select("CarParkID")
df_seq = model_input.select("sequences")

# LOAD MODEL
model = GBTRegressionModel.load(model_dir)

# PREPARE THE DATA
# Step 1: Flatten 5x5 array into 1D list (25 elements)
def flatten_array(arr):
    return [item for sublist in arr for item in sublist]

flatten_udf = udf(flatten_array, ArrayType(FloatType()))
df_flat = df_seq.withColumn("flat_features", flatten_udf("sequences"))

# Step 2: Convert to DenseVector
def to_vector(arr):
    return Vectors.dense(arr)

vector_udf = udf(to_vector, VectorUDT())
df_vectorized = df_flat.withColumn("features", vector_udf("flat_features"))
```

```
# GET PREDICTIONS
predictions = model.transform(df_vectorized)

# SCALE THE PREDCITIONS

# Step 1: Create a UDF that uses scaler_AL.inverse_transform
def inverse_prediction(pred):
    return float(scaler_AL.inverse_transform([[pred]])[0][0])

# Step 2: Register it as a UDF
inverse_udf = udf(inverse_prediction, FloatType())

# Step 3: Apply it to the prediction column
df_preds_all = predictions.withColumn("PredictedLots",
inverse_udf("prediction"))

# Show the result
#df_preds_all.select("prediction", "prediction_original_scale").show()

# GET LAST SET OF ORIGINAL DATA
# Step 1: Get the latest timestamp
max_ts = df.select(F.max("timestamp")).collect()[0][0]

# Step 2: Get the Min value corresponding to that timestamp
min_val_at_latest = df.filter(F.col("timestamp") == F.lit(max_ts)) \
                      .select(F.min("Min")) \
                      .collect()[0][0]

# Step 3: Filter all rows where Min equals that value and select only rows i
want
result_df = df.filter(F.col("Min") ==
F.lit(min_val_at_latest)).select("CarParkID", "AvailableLots", "timestamp",
"Location", "Development")

# ADD THE PREDICTIONS IN, CAST TO INT AND ROUND, AND ARRANGE FOR PREDICTION
BIGQUERY TABLE
# Add index to both dataframes
result_df_indexed = result_df.withColumn("row_id",
monotonically_increasing_id())
df_preds_indexed = df_preds_all.withColumn("row_id",
monotonically_increasing_id())

# Join on the row_id
joined_df = result_df_indexed.join(df_preds_indexed.select("row_id",
"PredictedLots"), on="row_id").drop("row_id")

final_df = joined_df.select(
```

```
    "CarParkID",
    "AvailableLots",
    round(col("PredictedLots")).cast(IntegerType()).alias("PredictedLots"),
    col("timestamp").alias("LatestTimestamp"),
    "Location",
    "Development"
)

# Show the final result
#final_df.show()

# WRITE TO BIG QUERY!
final_df.write \
    .format("bigquery") \
    .option("table", bigquery_dataset_output) \
    .option("temporaryGcsBucket", temp_bucket) \
    .mode("overwrite") \
    .save()

# END SESSION!
spark.stop()
```

20.4.2 CloudRun Code

```
# main.py
import functions_framework
from google.cloud import dataproc_v1 as dataproc

@functions_framework.http
def trigger_dataproc_job(request):  # <-- added request here
    PROJECT_ID = 'involuted-forge-456406-a1'
    REGION = 'us-central1'
    CLUSTER_NAME = 'cluster-53b3'

    job_client = dataproc.JobControllerClient(
        client_options={"api_endpoint": f"{REGION}-
dataproc.googleapis.com:443"}
    )

    job = {
        "placement": {"cluster_name": CLUSTER_NAME},
        "pyspark_job": {
            "main_python_file_uri":
"gs://dataproc_pythonfiles/dataproc_pysparkpipeline.py"
        },
    }
```

```
    operation = job_client.submit_job_as_operation(
        request={"project_id": PROJECT_ID, "region": REGION, "job": job}
    )

    response = operation.result()
    return f"Job finished with status: {response.status.state.name}", 200
```

20.5 Vertex AI Kubeflow Inference Pipeline

20.4.1 Vertex AI Kubeflow Inference Pipeline code

```
#from kfp.dsl import component, pipeline, Input, Output, Condition, OutputPath
#from kfp.dsl import Artifact, Dataset
from kfp import dsl
from kfp.dsl import Artifact, Dataset, Input, Output, Condition, OutputPath
from google_cloud_pipeline_components.types import artifact_types
#########PIPELINE LOAD AND CHECK
################################################################################
##########
#@component(packages_to_install=["google-cloud-bigquery", "pandas", "scikit-
learn"])
#def load_check_data(bigquery_dataset_input: str, my_project_id: str,
checked_parquet: str, checked_12_parquet: str) -> bool:
#@component(packages_to_install=["google-cloud-bigquery", "pandas", "db-
dtypes"])
@dsl.component(
    base_image="us-central1-docker.pkg.dev/involuted-forge-456406-a1/pipeline-
containers/my-custom-pipeline-image"
)
def load_check_data(bigquery_dataset_input: str, my_project_id: str,
checked_parquet: str, checked_12_parquet: str, run_id: str, success:
OutputPath(str)):

    from google.cloud import bigquery
    import pandas as pd
    import os
    import numpy as np
    from datetime import datetime, timedelta

    # Initialize BigQuery client
    client = bigquery.Client(project=my_project_id)

    #query = f"SELECT * FROM `{bigquery_dataset_input}`"
    #query = f"SELECT * FROM `{bigquery_dataset_input}` WHERE timestamp >
TIMESTAMP_ADD(CURRENT_TIMESTAMP(), INTERVAL '454' MINUTE)"
    query = f"SELECT * FROM `{bigquery_dataset_input}` WHERE timestamp >
DATETIME(TIMESTAMP_ADD(CURRENT_TIMESTAMP(), INTERVAL 454 MINUTE))"
```

```
    #query = f"SELECT * FROM `{bigquery_dataset}` WHERE timestamp >
TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL '25' MINUTE)"

    # Run query
    query_job = client.query(query)
    df = query_job.to_dataframe()

    #df

    #second query
    query = f"SELECT * FROM `{bigquery_dataset_input}` WHERE timestamp >
DATETIME(TIMESTAMP_ADD(CURRENT_TIMESTAMP(), INTERVAL 419 MINUTE))"

    # Run query
    query_job = client.query(query)
    df_12 = query_job.to_dataframe()

    # DECLARE SEQUENCE LENGTH CONSTANT
    ######################################################
    SEQUENCE_LENGTH = 5
    SEQUENCE_LENGTH_12 = 12

    # FUNCTIONS USED
    ######################################################
    def check_value_counts_equal(series):
        """
        Checks if all values in a Pandas Series' value_counts() are equal.

        Args:
            series: A Pandas Series.

        Returns:
            True if all value counts are equal, False otherwise.
        """
        value_counts = series.value_counts()
        if len(value_counts) <= 1: #if there is only one unique value, or
zero, then they are equal.
            return True
        else:
            return (value_counts == value_counts.iloc[0]).all() #compare all
to the first value.

    #Sort the data
    df.sort_values(['CarParkID', 'timestamp'], inplace=True)
    df_12.sort_values(['CarParkID', 'timestamp'], inplace=True)

    continue_flag = True
```

```
    if (check_value_counts_equal(df['CarParkID']) == False):
        #print("CarParkID is not equal. Don't continue. Trigger an error")
        continue_flag = False
    else:
        #print("CarParkEncoded is equal!")

        if (df['CarParkID'].value_counts().iloc[0] != SEQUENCE_LENGTH):
            #print("Don't have Sequence Length of Data Input. Don't continue.
Trigger an error")
            continue_flag = False
    #    else:
            #print("Have right amount of data to proceed. Data Preprocessing
Completed Successfully")

    if (check_value_counts_equal(df_12['CarParkID']) == False):
        #print("CarParkID is not equal. Don't continue. Trigger an error")
        continue_flag = False
    else:
        #print("CarParkEncoded is equal!")

        if (df_12['CarParkID'].value_counts().iloc[0] != SEQUENCE_LENGTH_12):
            #print("Don't have Sequence Length of Data Input. Don't continue.
Trigger an error")
            continue_flag = False

    # SAVE TO PARQUET FILE
    df.to_parquet(checked_parquet, index=False)
    df_12.to_parquet(checked_12_parquet, index=False)

    # PIPELINE COMPONENT OUTPUT
    with open(success, 'w') as f:
        f.write('true' if continue_flag else 'false')

#########PIPELINE
PREPROCESS##############################################################################
####################
#@component(packages_to_install=["google-cloud-bigquery", "pandas", "google-
cloud-storage", "pyarrow", "scikit-learn"])
@dsl.component(
    base_image="us-central1-docker.pkg.dev/involuted-forge-456406-a1/pipeline-
containers/my-custom-pipeline-image"
)
def preprocess_data(checked_parquet: str,
                    sequence_parquet: str,
                    pkl_file_path_1: str,
                    pkl_file_path_2: str,
```

```
                    pkl_file_path_3: str,
                    pkl_file_path_4: str,
                    run_id: str,
                    success: OutputPath(str)):

    import pandas as pd
    import numpy as np
    import os
    import pickle
    from datetime import datetime, timedelta
    from google.cloud import storage

    # DECLARE SEQUENCE LENGTH CONSTANT
    ######################################################
    SEQUENCE_LENGTH = 5

    # FUNCTIONS USED
    ######################################################
    def create_sequences_per_location(df, sequence_length=5):

        # Define the columns to use for the 3D array
        selected_columns = ['CarParkEncoded_MM', 'Min_MM', 'Hour_MM',
'DayOfWeek_MM', 'AvailableLots_MM']
        subset_df = df[selected_columns]

        # Convert the DataFrame subset to a NumPy array
        numpy_array = subset_df.values

        # Calculate the number of 2D arrays
        num_sequences  = numpy_array.shape[0] // sequence_length
        remainder = numpy_array.shape[0] % sequence_length

        if remainder != 0:
            #print("ERROR! Not every Carpark has Sequence Length entries")
            return None, False
        else:
            # Create the list to store the 3D arrays
            sequence = []

            # Reshape the array into 3D chunks
            sequence = numpy_array.reshape(num_sequences, sequence_length,
len(selected_columns))

            #print("Sequencing Done! ")
            return sequence, True

    def read_pkl_file(pkl_file_path):
```

```
        try:
            # Initialize the Google Cloud Storage client
            client = storage.Client()

            # Parse the GCS URI
            bucket_name = pkl_file_path.split('/')[2]
            blob_name = '/'.join(pkl_file_path.split('/')[3:])  # ERROR HERE:
Using pkl_file_path_1

            # Get the bucket and blob (file)
            bucket = client.bucket(bucket_name)
            blob = bucket.blob(blob_name)

            # Download the blob's content to a bytes object
            blob_data = blob.download_as_bytes()

            #print("Loaded successfully from GCS.")

            # Load the object from the bytes object
            return pickle.loads(blob_data)

        except Exception as e:
            #print(f"Error loading object from GCS: {e}")
            raise RuntimeError(f"Error loading object from GCS: {e}")
            return None

    # READ PARQUET FILE
    ###########################################################
    df = pd.read_parquet(checked_parquet)

    # OPEN MODEL SCALARS AND ENCODERS
    ###########################################################
    le = read_pkl_file(pkl_file_path_1)
    scaler_AL = read_pkl_file(pkl_file_path_2)
    scalar_CP = read_pkl_file(pkl_file_path_3)
    scaler_temp = read_pkl_file(pkl_file_path_4)


    # DATA PREPROCESSING
    ###########################################################
    df['timestamp'] = pd.to_datetime(df['timestamp'])

    # FEATURE EXTRACTION/ ENCODING/ NORMALISATION
    # Extract features
    df['Min'] = df['timestamp'].dt.minute
    df['Hour'] = df['timestamp'].dt.hour
    df['DayOfWeek'] = df['timestamp'].dt.dayofweek
```

```
    # Encode CarParkID
    df['CarParkEncoded'] = le.transform(df['CarParkID'])

    # Sort for time sequence modeling
    df.sort_values(['CarParkEncoded', 'timestamp'], inplace=True)

    # NORMALISE
    df['AvailableLots_MM'] = scaler_AL.transform(df[['AvailableLots']])
    df['CarParkEncoded_MM'] = scalar_CP.transform(df[['CarParkEncoded']])
    df[['Min_MM', 'Hour_MM', 'DayOfWeek_MM']] =
scaler_temp.transform(df[['Min','Hour', 'DayOfWeek']])

    # LSTM SEQUENCING
    ########################################################

    continue_flag = True
    model_input, sequence_done_flag = create_sequences_per_location(df,
SEQUENCE_LENGTH)

    # Extra check. If fail don't proceed to predictions. Need to add some code
here
    if (sequence_done_flag == False):
        #print("ERROR! Not every Carpark has Sequence Length entries. Raise
Error")
        continue_flag = False


    #model_input_flat = model_input.reshape(88, 25)
    model_input_flat = model_input.reshape(model_input.shape[0], -1)

    # Save as DataFrame and export to Parquet
    df_model_input = pd.DataFrame(model_input_flat)

    df_model_input.to_parquet(sequence_parquet, index=False)

    # PIPELINE COMPONENT OUTPUT
    with open(success, 'w') as f:
        f.write('true' if continue_flag else 'false')

##########PIPELINE PREDICTION AND UPDATE
TABLE#################################################################################
################

#@component(packages_to_install=["google-cloud-bigquery", "pandas","google-
cloud-storage","tensorflow","db-dtypes", "pyarrow", "scikit-learn"])
@dsl.component(
```

```
    base_image="us-central1-docker.pkg.dev/involuted-forge-456406-a1/pipeline-
containers/my-custom-pipeline-image"
)
def predict_with_lstm(model_keras_30: str,
                      model_keras_45: str,
                      model_keras_60: str,
                      checked_parquet: str,
                      checked_12_parquet: str,
                      sequence_parquet: str,
                      pkl_file_path_2: str,
                      my_project_id:str,
                      bigquery_dataset_output:str,
                      run_id: str):

    import tensorflow as tf
    import os
    import numpy as np
    import pandas as pd
    import pickle
    from google.cloud import storage
    from datetime import datetime, timedelta
    from google.cloud import bigquery

    df_model_input = pd.read_parquet(sequence_parquet)

    #df_model_input

    model_input = df_model_input.to_numpy().reshape(88, 5, 5)

    # Load the model
    #loaded_LSTM_Model = tf.keras.models.load_model(model_keras)
    LSTM_30 = tf.keras.models.load_model(model_keras_30)
    LSTM_45 = tf.keras.models.load_model(model_keras_45)
    LSTM_60 = tf.keras.models.load_model(model_keras_60)

    #preds = loaded_LSTM_Model.predict(model_input)

    preds_30 = LSTM_30.predict(model_input)
    preds_45 = LSTM_45.predict(model_input)
    preds_60 = LSTM_60.predict(model_input)



    def read_pkl_file(pkl_file_path):
        try:
            # Initialize the Google Cloud Storage client
            client = storage.Client()
```

```
            # Parse the GCS URI
            bucket_name = pkl_file_path.split('/')[2]
            blob_name = '/'.join(pkl_file_path.split('/')[3:])  # ERROR HERE:
Using pkl_file_path_1

            # Get the bucket and blob (file)
            bucket = client.bucket(bucket_name)
            blob = bucket.blob(blob_name)

            # Download the blob's content to a bytes object
            blob_data = blob.download_as_bytes()

            #print("Loaded successfully from GCS.")

            # Load the object from the bytes object
            return pickle.loads(blob_data)

        except Exception as e:
            #print(f"Error loading object from GCS: {e}")
            raise RuntimeError(f"Error loading object from GCS: {e}")
            return None

    scaler_AL = read_pkl_file(pkl_file_path_2)

    preds_t_30 = scaler_AL.inverse_transform(preds_30)
    preds_t_45 = scaler_AL.inverse_transform(preds_45)
    preds_t_60 = scaler_AL.inverse_transform(preds_60)



    # READ PARQUET FILE
    ########################################################
    df = pd.read_parquet(checked_parquet)

    df['timestamp'] = df['timestamp'].dt.floor('min')
    max_timestamp = df['timestamp'].max()
    df_lasttime = df[df['timestamp'] == max_timestamp]

    df_pred = df_lasttime.drop(['Area', 'Development', 'LotType', 'Agency'],
axis=1)
    df_pred = df_pred.rename(columns={'timestamp': 'LatestTimestamp'})

    df_pred['PredictedLots_30min'] = np.round(preds_t_30).astype(int)
    df_pred['PredictedLots_45min'] = np.round(preds_t_45).astype(int)
    df_pred['PredictedLots_1hour'] = np.round(preds_t_60).astype(int)
```

```
    df_12 = pd.read_parquet(checked_12_parquet)

    df_12['timestamp'] = df_12['timestamp'].dt.floor('min')

    pivot_df = df_12.pivot(index='CarParkID', columns='timestamp',
values='AvailableLots')

    # 4. Sort columns (timestamps) from earliest to latest
    pivot_df = pivot_df.sort_index(axis=1)

    # 5. Create new column names
    n_steps = pivot_df.shape[1]  # number of timestamps

    new_cols = [f"t-{5*(n_steps-i-1)}" if i != n_steps-1 else 't' for i in
range(n_steps)]

    # 6. Rename the columns
    pivot_df.columns = new_cols

    # 7. (Optional) Reset index if you want CarParkID as a column
    pivot_df = pivot_df.reset_index()

    #pivot_df

    df_final = df_pred.merge(pivot_df, on='CarParkID', how='inner')

    #df_final

    # Initialize client
    client = bigquery.Client(project=my_project_id)


    # Configure job to overwrite the table
    job_config = bigquery.LoadJobConfig(
        write_disposition=bigquery.WriteDisposition.WRITE_TRUNCATE
    )

    # Write DataFrame to BigQuery
    job = client.load_table_from_dataframe(df_final, bigquery_dataset_output,
job_config=job_config)
    job.result()  # Wait for the job to complete
```

```
# PIPELINE
#########################################################################################
#########################################################################################
#####
@dsl.pipeline(name="bigquery-preprocessing-pipeline")
def pipeline(my_project_id: str = "involuted-forge-456406-a1",
             bigquery_dataset_input: str = "smart-car-park-availability-
1.lta_data.view_carpark_availability_for_prediction",
             bigquery_dataset_output: str = "precise-tube-456807-
h5.externalData.predictions_new",
             checked_parquet: str =
"gs://carpark_predict_temp/df_checked.parquet",
             checked_12_parquet: str =
"gs://carpark_predict_temp/df12_checked.parquet",
             sequence_parquet: str =
"gs://carpark_predict_temp/sequenced.parquet",
             pkl_file_path_1: str =
"gs://carpark_predict_models/4april2025/label_encoder_Carparks.pkl",
             pkl_file_path_2: str =
"gs://carpark_predict_models/4april2025/min_max_scaler_AvailableLots.pkl",
             pkl_file_path_3: str =
"gs://carpark_predict_models/4april2025/min_max_scaler_Carparks.pkl",
             pkl_file_path_4: str =
"gs://carpark_predict_models/4april2025/min_max_scaler_Min_Hour_Day.pkl",
             model_keras_30: str =
"gs://carpark_predict_models/26APR/LSTM_Model_26APR_30mins.keras",
             model_keras_45: str =
"gs://carpark_predict_models/26APR/LSTM_Model_26APR_45mins.keras",
             model_keras_60: str =
"gs://carpark_predict_models/26APR/LSTM_Model_26APR_60mins.keras",
             run_id: str = "no_run_id" ):

    from kfp import dsl

    load_table_task = load_check_data(bigquery_dataset_input =
bigquery_dataset_input,
                      my_project_id = my_project_id,
                      checked_parquet = checked_parquet,
                      checked_12_parquet = checked_12_parquet,
                      run_id=run_id)

    load_table_task.set_caching_options(False) # Disable caching for this
component

    with dsl.Condition(load_table_task.outputs["success"] == 'true'):

        preprocess_task = preprocess_data(checked_parquet = checked_parquet,
```

```
                                        sequence_parquet = sequence_parquet,
                                        pkl_file_path_1 = pkl_file_path_1,
                                        pkl_file_path_2 = pkl_file_path_2,
                                        pkl_file_path_3 = pkl_file_path_3,
                                        pkl_file_path_4 = pkl_file_path_4,
                                        run_id=run_id)
        preprocess_task.set_caching_options(False) # Disable caching for this
component

        with dsl.Condition(preprocess_task.outputs["success"] == 'true'):

            prediction_task = predict_with_lstm(model_keras_30 =
model_keras_30,
                                                model_keras_45 =
model_keras_45,
                                                model_keras_60 =
model_keras_60,
                                                checked_parquet =
checked_parquet,
                                                checked_12_parquet =
checked_12_parquet,
                                                sequence_parquet =
sequence_parquet,
                                                pkl_file_path_2 =
pkl_file_path_2,
                                                my_project_id = my_project_id,
                                                bigquery_dataset_output =
bigquery_dataset_output,
                                                run_id=run_id)

            prediction_task.set_caching_options(False) # Disable caching for
this component


##############################################################################
################################################
```

20.5.2 Vertex AI Kubeflow Inference Pipeline compiler

```
from kfp import compiler
from LSTM_pipeline import pipeline

# Compile the pipeline
compiler.Compiler().compile(
    pipeline_func=pipeline,
    package_path="pipeline.json"  # output filename
)
```

20.5.3 Docker Image code

```
# Start from official Python image
FROM python:3.10


# Install all needed libraries
RUN pip install \
    google-cloud-bigquery \
    google-cloud-storage \
    pandas \
    scikit-learn \
    db-dtypes \
    pyarrow \
    tensorflow
```

20.6 Data Ingestion

20.6.1 CloudFunction Code

```
# app.py
import requests
import json
from google.cloud import pubsub_v1

PROJECT_ID = "smart-car-park-availability-1"
TOPIC_ID = "carpark-data-topic"
API_KEY = "7GQ4fcMqRTuEm4Tb681Y6A=="

publisher = pubsub_v1.PublisherClient()
topic_path = publisher.topic_path(PROJECT_ID, TOPIC_ID)

def fetch_and_publish(request):
    url = "http://datamall2.mytransport.sg/ltaodataservice/CarParkAvailabilityv2"
    headers = {"AccountKey": API_KEY, "accept": "application/json"}

    try:
        response = requests.get(url, headers=headers)
        response.raise_for_status()
        data = response.json().get("value", [])

        for record in data:
            message = json.dumps(record).encode("utf-8")
            publisher.publish(topic_path, message)

        return f"✅ Published {len(data)} records.", 200
    except Exception as e:
```

```
        return f"❌ Error occurred: {e}", 500
```

20.6.2 DataFlow Code

```
import json
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions, StandardOptions

PROJECT = "smart-car-park-availability-1"
BUCKET = "smart-car-park-availability-1-dataflow"
BQ_TABLE = "smart-car-park-availability-1:lta_data.carpark_availability"
TOPIC = f"projects/{PROJECT}/topics/carpark-data-topic"


class ParseJson(beam.DoFn):
    def process(self, element):
        import json
        record = json.loads(element.decode("utf-8"))

        yield {
            "CarParkID": record.get("CarParkID"),
            "Area": record.get("Area"),
            "Development": record.get("Development"),
            "Location": record.get("Location"),
            "AvailableLots": int(record.get("AvailableLots", 0)),
            "LotType": record.get("LotType"),
            "Agency": record.get("Agency")
        }

def run():
    options = PipelineOptions(
        streaming=True,
        project=PROJECT,
        temp_location=f"gs://{BUCKET}/tmp/",
        region="asia-southeast1"
    )
    options.view_as(StandardOptions).streaming = True

    with beam.Pipeline(options=options) as p:
        (
            p
            | "ReadFromPubSub" >> beam.io.ReadFromPubSub(topic=TOPIC)
            | "ParseJSON" >> beam.ParDo(ParseJson())
            | "WriteToBQ" >> beam.io.WriteToBigQuery(
                BQ_TABLE,
                schema={
                    "fields": [
                        {"name": "CarParkID", "type": "STRING"},
```

```
                {"name": "Area", "type": "STRING"},
                {"name": "Development", "type": "STRING"},
                {"name": "Location", "type": "STRING"},
                {"name": "AvailableLots", "type": "INTEGER"},
                {"name": "LotType", "type": "STRING"},
                {"name": "Agency", "type": "STRING"}
            ]
        },
        write_disposition="WRITE_APPEND",
        create_disposition="CREATE_IF_NEEDED",
    )
)

if __name__ == "__main__":
    run()
```

20.7 Flask Pipeline

20.7.1 CloudRun Code

```
# app.py
from flask import Flask, request, jsonify
import numpy as np
from tensorflow.keras.models import load_model
import joblib
import os
import pandas as pd

app = Flask(__name__)

# Load model from GCS mounted bucket or local file system
MODEL_PATH = "gs://prediction_models_ej/models/lstm_model_x1.keras"
model = load_model(MODEL_PATH)

# Predict route
@app.route("/predict", methods=["POST"])
def predict():
    try:
        data = request.get_json()
        sequences = data.get("sequences", [])  # Expect list of lists with 10 elements each
        ids = data.get("carpark_ids", [])

        predictions = []
        for seq, carpark_id in zip(sequences, ids):
            input_array = np.array(seq, dtype=np.float32).reshape(1, -1, 1)
            predicted = model.predict(input_array)[0][0]

            # Apply constraints
```

```
        predicted = max(0, round(predicted))

        predictions.append({
            "carpark_id": carpark_id,
            "predicted_lots": predicted
        })

    return jsonify(predictions)

  except Exception as e:
    return jsonify({"error": str(e)}), 500


if __name__ == "__main__":
  port = int(os.environ.get("PORT", 8080)) # ☑ Cloud Run expects 8080
  app.run(host="0.0.0.0", port=port)
```

20.8 Model Monitoring

20.8.1 Model Monitoring Code

```
WITH RankedActuals AS (
  SELECT
    a.CarParkID,
    a.timestamp AS actual_time,
    a.AvailableLots AS actual_lots,
    ROW_NUMBER() OVER (
      PARTITION BY a.CarParkID, TIMESTAMP_TRUNC(a.timestamp, MINUTE)
      ORDER BY a.timestamp ASC
    ) AS rn
  FROM
    `smart-car-park-availability-1.lta_data.carpark_availability` a
),
prediction_vs_actual AS (
  SELECT
    p.CarParkID,
    p.utctimestamp AS prediction_time,
    TIMESTAMP_ADD(
      TIMESTAMP_TRUNC(CAST(p.utctimestamp AS TIMESTAMP), MINUTE),
      INTERVAL CAST(
        30 + (FLOOR(EXTRACT(MINUTE FROM CAST(p.utctimestamp AS TIMESTAMP)) / 5) *
5)
        - EXTRACT(MINUTE FROM CAST(p.utctimestamp AS TIMESTAMP))
        AS INT64
      ) MINUTE
    ) AS expected_time,
    p.PredictedLots,
    ra.actual_time,
```

```
    ra.actual_lots
  FROM
    `smart-car-park-availability-1.lta_data.view_carpark_predictions_30min_1` p
  LEFT JOIN
    RankedActuals ra
  ON
    p.CarParkID = ra.CarParkID
    AND TIMESTAMP_TRUNC(ra.actual_time, MINUTE) = TIMESTAMP_TRUNC(
      TIMESTAMP_ADD(
        TIMESTAMP_TRUNC(CAST(p.utctimestamp AS TIMESTAMP), MINUTE),
        INTERVAL CAST(
          30 + (FLOOR(EXTRACT(MINUTE FROM CAST(p.utctimestamp AS TIMESTAMP)) / 5) *
5)
          - EXTRACT(MINUTE FROM CAST(p.utctimestamp AS TIMESTAMP))
          AS INT64
        ) MINUTE
      ),
      MINUTE
    )
    AND ra.rn = 1
)

SELECT
  TIMESTAMP("2025-04-22 19:00:00+08:00") AS monitoring_timestamp,  -- Required for
table schema
  COUNT(*) AS total_records,
  ROUND(AVG(ABS(PredictedLots - actual_lots)), 2) AS MAE,
  ROUND(SQRT(AVG(POW(PredictedLots - actual_lots, 2))), 2) AS RMSE,
  ROUND(
    AVG(
      CASE
        WHEN actual_lots IS NOT NULL AND actual_lots != 0 THEN ABS((actual_lots -
PredictedLots) / actual_lots) * 100
        ELSE NULL
      END
    ),
    2
  ) AS MAPE
FROM prediction_vs_actual;
```

20.9 Smart Routing

20.9.1 Smart Routing Code

```
# Install required packages
!pip install google-cloud-bigquery requests --quiet

# Authenticate GCP
from google.colab import auth
auth.authenticate_user()

# Imports
from google.cloud import bigquery
import re
from math import radians, cos, sin, asin, sqrt

# Initialize BigQuery Client
ej_project_id = 'protean-acrobat-456609-u8'
client = bigquery.Client(project=ej_project_id)

# Clean and convert coordinates
def parse_coordinate(coord_str):
    coord_str = coord_str.strip().upper().replace("°", "")
    direction = re.search(r"[NSEW]$", coord_str)
    number = float(re.sub(r"[NSEW]", "", coord_str))
    if not direction:
        return number
    if direction.group() in ['S', 'W']:
        return -abs(number)
    return abs(number)

# Haversine formula for distance calculation
def haversine(lat1, lon1, lat2, lon2):
    R = 6371  # Earth radius in km
    dlat = radians(lat2 - lat1)
    dlon = radians(lon2 - lon1)
    a = sin(dlat/2)**2 + cos(radians(lat1)) * cos(radians(lat2)) *
sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    return R * c

# Hardcoded Coordinates (user location) - replace if needed
user_lat = parse_coordinate("1.3479° N")
user_lon = parse_coordinate("103.9583° E")
user_location = (user_lat, user_lon)

# Fetch parking lots with slot and distance filtering
```

```
def fetch_nearby_parking(user_lat, user_lon, min_slots=5,
max_slots=150, max_distance_km=4):
    query = """
        SELECT latitude, longitude, Location, predicted_slots
        FROM `smart-car-park-
availability.lta_data.view_carpark_availability_realtime`
        WHERE predicted_slots BETWEEN @min_slots AND @max_slots
    """
    job_config = bigquery.QueryJobConfig(
        query_parameters=[
            bigquery.ScalarQueryParameter("min_slots", "INT64",
min_slots),
            bigquery.ScalarQueryParameter("max_slots", "INT64",
max_slots),
        ]
    )
    query_job = client.query(query, job_config=job_config)
    results = query_job.result()

    nearby = []
    for row in results:
        distance = haversine(user_lat, user_lon, row.latitude,
row.longitude)
        if distance <= max_distance_km:
            nearby.append({
                "location_name": row.Location,
                "carpark_id": row.Location,  # using Location as ID
until a real ID field is available
                "latitude": row.latitude,
                "longitude": row.longitude,
                "predicted_slots": row.predicted_slots,
                "distance_km": round(distance, 2)
            })

    nearby.sort(key=lambda x: x["distance_km"])
    return nearby

# Google Maps Directions Link
def generate_google_maps_link(start_lat, start_lon, end_lat, end_lon):
    return
f"https://www.google.com/maps/dir/{start_lat},{start_lon}/{end_lat},{en
d_lon}/"

# === Main Execution ===
try:
    candidates = fetch_nearby_parking(user_lat, user_lon)
```

```
    if candidates:
        best = candidates[0]
        print("\n☑ Recommended Parking Lot (Nearest):")
        print(f"Location Name: {best['location_name']}")
        print(f"Carpark Lot ID: {best['carpark_id']}")
        print(f"Coordinates: ({best['latitude']},
{best['longitude']})")
        print(f"Available Slots: {best['predicted_slots']}")
        print(f"Distance: {best['distance_km']} km")
        print(f"Google Maps Route: {generate_google_maps_link(user_lat,
user_lon, best['latitude'], best['longitude'])}")

        if len(candidates) > 1:
            print("\n💡 Other Nearby Suggestions:")
            for suggestion in candidates[1:4]:
                print(f"- Location Name:
{suggestion['location_name']}")
                print(f"  Carpark Lot ID: {suggestion['carpark_id']}")
                print(f"  Slots: {suggestion['predicted_slots']}")
                print(f"  Distance: {suggestion['distance_km']} km")
                print(f"  Google Maps:
{generate_google_maps_link(user_lat, user_lon, suggestion['latitude'],
suggestion['longitude'])}\n")

    else:
        print("❌ No parking lots found with 10-15 slots within 3 km
radius.")

except Exception as e:
    print(f"❌ Error: {e}
```