Carpark Availability System with AI Insights – Technical Documentation (Updated on 17th August 2025)

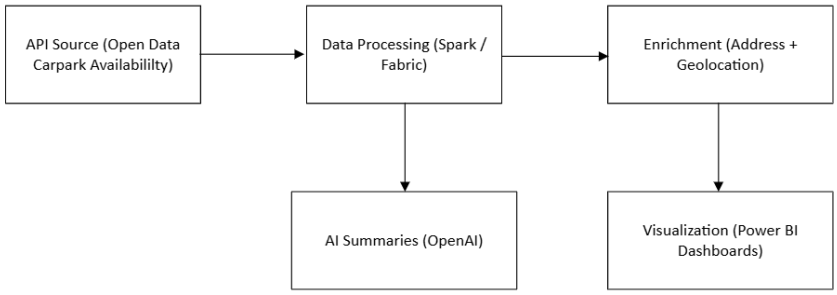
1. Project Overview

The **Carpark Availability System** provides real-time insights into Singapore's carpark occupancy using the Singapore Open Data API. The system combines **data engineering pipelines**, **real-time data ingestion**, **geospatial enrichment** and **AI summarisation** using OpenAI to generate human-friendly insights for operational and management decision-making.

Key Features:

- Real-time ingestion of carpark availability from the Singapore Open Data API
- Data enrichment with address and geolocation
- Parallel processing for coordinate fetching
- Storage and transformation in **Microsoft Fabric Lakehouse**
- AI-generated daily summaries using OpenAI API
- Integration with **Power BI** for dashboards and maps

2. System Architecture



3. Data Sources & Enrichment

Source	Purpose	Notes
https://api.data.gov.sg/v1/transport/carpark-availability	Real-time availability	Provides carpark_number, total_lots, lots_available, update_datetime
HDBCarparkInformation (CSV)	Enrich carpark with **address**	Merged on carpark_number
Geolocation API / get_coordinates function	Add **Latitude and Longitude**	Executed in parallel to speed up processing

4. Data Flow with Merging & Cleaning

a) **Ingest Raw Availability Data**
   a. Fetch JSON from Singapore Open Data API.
   b. Flatten JSON into tabular Spark DataFrame.

b) **Enrich with Address**
   a. Read HDBCarparkInformation file in Spark.
   b. Merge on carpark_number:
   c. carpark_final = carpark_availability.join(hdb_info, on="carpark_number", how="inner")

c) **Fetch Coordinates in Parallel**
   a. Extract address list from merged table
   b. Use **ThreadPoolExecutor** to parallelize API calls, which reduces execution time drastically for thousands of carparks

d) **Data Cleaning**
   i. Convert Latitude and Longitude to numeric types.
   ii. Handle missing values:
   iii. carpark_final = carpark_final.dropna(subset=['Latitude','Longitude'])
   iv. Ensure lots_available and total_lots are integers.
   v. Remove any duplicates on carpark_number + update_datetime

e) Save Cleaned & Enriched Table
    i. Store final enriched table in **Delta format** for AI analysis.

5. Deployment and Logging

a) Deployed as a scheduled Fabric Notebook pipeline.
b) Logging implemented in Spark and errors in API calls are retried with exponential backoff.

6. Data Quality Measure

- Removed duplicates based on carpark_number + update_datetime
- Dropped rows with missing geolocation
- Validated lots_available is less than or equal to total_lots
- Rows where availability percentage exceeds 100% are flagged as potential data quality issues.

7. AI Integration

- Aggregation of daily statistics (avg/min/max lots available)
- Send daily statistics to **OpenAI GPT-4o-mini**
- Generate human-readable summaries with enriched context:
    - Include address information if needed
    - Potentially use coordinates for **mapping or heatmap insights**

8. Sample Fabric Notebook Code Snippet

```
# Merge HDB information
hdb_info = spark.read.csv("Files/HDBCarparkInformation.csv", header=True)
carpark_final = carpark_df.join(hdb_info, on="carpark_number", how="inner")

# Prepare addresses for coordinates
addresslist = [row['address'] for row in carpark_final.collect()]

# Fetch coordinates in parallel
from concurrent.futures import ThreadPoolExecutor

max_workers = 10
with ThreadPoolExecutor(max_workers=max_workers) as executor:
  future = executor.submit(get_coordinates, addresslist)
  coordinates_list = future.result()

# Clean numeric fields
carpark_final = carpark_final.withColumn("lots_available", carpark_final["lots_available"].cast("int")) \
                .withColumn("total_lots", carpark_final["total_lots"].cast("int")) \
                .dropna(subset=["Latitude","Longitude"])

# Generate AI Summary
from openai import OpenAI
# Create OpenAI client
client = OpenAI(api_key=openai.api_key)

prompt = f"""
Summarise the following daily carpark availability statistic for Singapore:
{stats_str}
Generate a table showing a sample list of carparks with notable availability patterns, highlighting anomalies such as availability percentages exceeding 100%, indicating possible data quality issues. After the Availability column, include the fields: lots_available, total_lots, location, and datetime. Sort the table in ascending order by Availability.
"""
# Create completion
response = client.chat.completions.create(
  model="gpt-4o-mini",
  messages=[
```

```
        {"role": "system", "content": "You are a Data Analyst."},
        {"role": "user", "content": prompt}
    ],
    max_tokens=2000)

# Extract AI summary text
ai_summary = response.choices[0].message.content
print("\n🤖 AI Summary:")
print(ai_summary)
```

9. Key Notes on Parallel Processing

- **ThreadPoolExecutor** allows concurrent execution of I/O-bound tasks (like API calls for coordinates).
- max_workers=10 is adjustable based on system capability and API rate limits.
- Latency is significantly reduced compared to sequential coordinate fetching.

10. Downstream AI & Visualization

- The **enriched, cleaned table** feeds into:
  - Daily aggregation for **OpenAI summarisation**
  - Power BI dashboards with **map visualizations** using Latitude/Longitude
  - AI insights include **location-aware analysis** (busiest areas, trends by region)

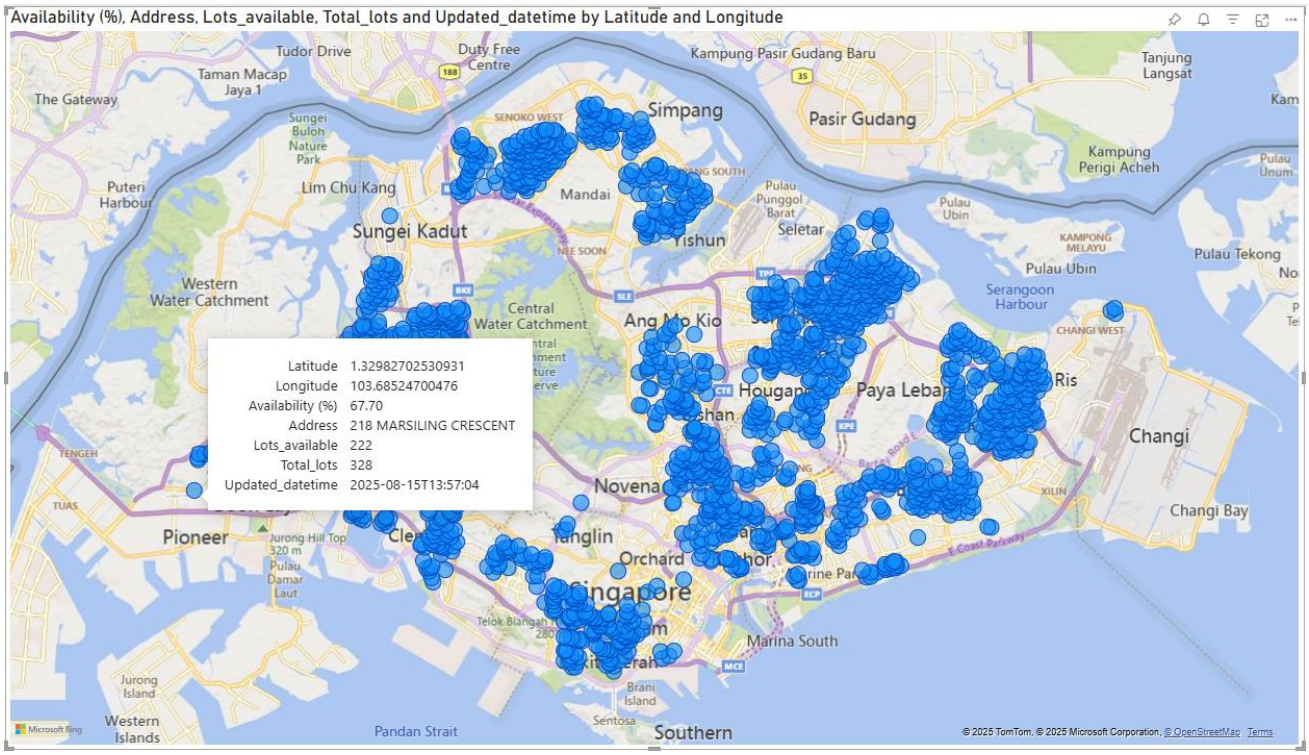
11. Outputs

🤖 AI Summary:
Here's a corrected and organized summary of selected carpark availability data for Singapore. The entries are sorted in ascending order by availability percentage. Each entry includes the requested data:

Carpark	Availability (%)	Lots Available	Total Lots	Location	Last Updated
A15	0.0	0	59	226H ANG MO KIO STREET 22	2025-08-15T11:09:04
A20	0.0	0	83	319 ANG MO KIO STREET 31	2025-08-15T11:08:58
A30	0.0	0	12	134 ANG MO KIO STREET 12	2025-08-15T11:08:50
A33	0.0	0	3	254 ANG MO KIO STREET 21	2025-08-15T11:09:58
A59	60.0	103	172	551 ANG MO KIO STREET 54	2025-08-15T11:08:14
B80	85.0	213	250	739A BEDOK RESERVOIR ROAD	2025-08-15T11:07:27
J7	97.0	134	138	203 JURONG EAST STREET 21	2025-08-15T11:05:32
A39	100.0	199	200	405 ANG MO KIO AVENUE 10	2025-08-15T11:08:50
HE12	162.9	171	105	81 REDHILL LANE	2025-08-15T11:09:10
SK61	500.0	50	10	406 FERNVALE ROAD	2025-08-15T11:03:55
TB6	183.0	244	133	33 TELOK BLANGAH WAY	2025-08-15T11:05:56

Issues Identified:
- The calculations for availability had inconsistencies and errors in the data entries for car parks like HE12 (171/105), SK61 (50/10) and TB6 (244/133).

Notes:
- For the entries with issues or incorrect values, it's essential to investigate the data source or correct entries needing review.



Further enhancements could be explored in the following areas:

a) Implement real-time alerts for carpark availability using a combination of Real-Time Analytics (KQL Database) + Power Automate.
b) Include embeddings or **RAG** (Retrieval-Augmented Generation) for OpenAI API implementation.

12. RAG (Retrieval-Augmented Generation) for OpenAI API implementation

Why RAG is needed for this carpark system?
The basic AI integration provides daily summaries, but RAG enables:

- Historical pattern recognition: "Show me carparks with similar availability patterns to Location X"
- Contextual insights: "What factors typically cause low availability in this area?"
- Semantic search: "Find carparks near shopping malls with consistent evening availability"

Comparison of RAG and the basic OpenAI integration of AI Summaries.

Aspect	RAG (Retrieval-Augmented Generation)	AI Summaries
Focus	Current data plus historical, seasonal, and comparative context	Snapshot of current data
Depth	Analytical (what is happening now, how it compares, why it matters)	Descriptive (what is happening now)
Context	Enriched with external knowledge, past data, and documents	Limited to the given dataset
Insights	Deeper insights, patterns, and trends over time	Quick highlights, surface-level
Comparisons	Benchmarks against history, peers, or averages	Usually absent
Predictive Value	High – can suggest expected trends or upcoming issues	None – static summary
Use Case Fit	Best for strategic decisions and forecasting reports	Good for daily/instant reports

RAG Workflow Description

Data Preparation Phase: Raw carpark availability data is first processed into **historical_stats**, which aggregates key metrics like average availability, peak usage times, and seasonal patterns for each carpark location.

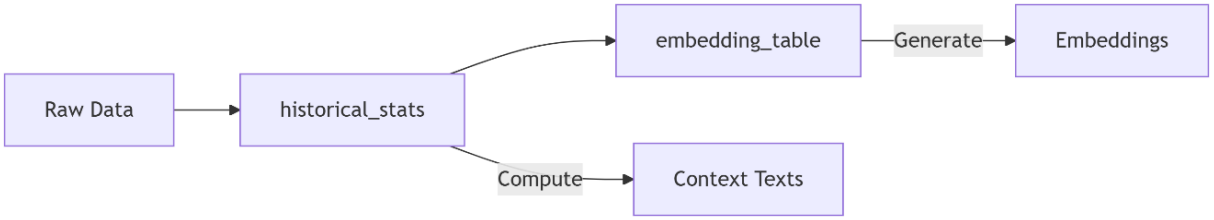
Context Generation: From the historical statistics, the system computes rich Context Texts that combine quantitative data with descriptive information - for example, "Carpark ABC123 at Orchard Road averages 65% occupancy on weekdays, with peak usage from 11 AM-2 PM during lunch hours, and shows 20% higher demand during shopping festivals."

Embedding Creation: These context texts are fed into the **embedding_table** process, which generates high-dimensional vector Embeddings using OpenAI's text-embedding models, creating numerical representations that capture semantic meaning and relationships between different carpark contexts.

Query-Time Retrieval: When a user asks a question about carpark availability, the system converts the query into an embedding, performs similarity search against the stored embeddings to find the most relevant historical context, and then combines this retrieved information with current real-time data to generate comprehensive, context-aware insights through the AI model.

Key Advantage: This workflow enables the AI to provide responses like "Current low availability in Jurong West is unusual - historically this area maintains 70% availability on Saturday afternoons, suggesting a special event or incident may be occurring" rather than just reporting raw numbers.

The workflow essentially transforms static historical data into searchable, contextual knowledge that enhances real-time analysis. See workflow below, and the differences between **historical_stats** and **embedding_table**.

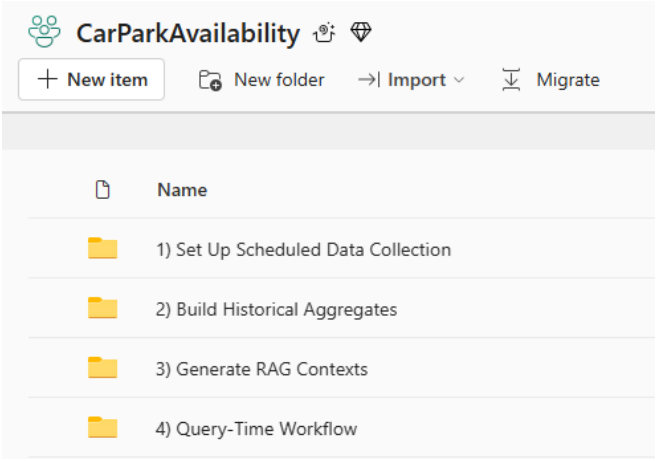


Feature	`historical_stats`	`embedding_table`
Data Type	Text summaries	Text + Vector embeddings
Columns	`[carpark_number, context]`	`[... + embedding]`
Purpose	Data preprocessing	Semantic search
Size	Smaller (text only)	Larger (vector arrays)

- `historical_stats` for business logic
- `embedding_table` for RAG queries

Embedding_table enables semantic search and stores **vector embeddings** (numeric representations) of the carpark context data.

There are basic 4 steps in the RAG workflow, as seen below.



Step 1: Raw ingestion (CarparkDataCollector) → CarparkAvailability

Step 2 Historical Aggregation (Carpark_Aggregator) → carpark_hourly_agg

- Prepares clean, summarized, manageable dataset

Step 3: RAG_Embedding_Generator → embed the hourly summaries from carpark_hourly_agg

Step 4: Query-Time Retrieval → retrieve embeddings + combine with current real-time snapshot

a) **Step 1**: The initial phase of **automated** data collection involves retrieving carpark data from the API and enriching it with address and geolocation information. This enriched data is stored in the Lakehouse Delta table called **CarparkAvailability**, with new records appended on every 5 mins.

```
# ==========================
# 2) EXTRACT DATA FROM API
# ==========================
url = "https://api.data.gov.sg/v1/transport/carpark-availability"
response = requests.get(url)
data = response.json()

# Flatten JSON
carparks = []
for item in data['items'][0]['carpark_data']:
    carparks.append({
        "carpark_number": item["carpark_number"],
        "total_lots": int(item["carpark_info"][0]["total_lots"]),
        "lots_available": int(item["carpark_info"][0]["lots_available"]),
        "update_datetime": item["update_datetime"]
    })

# Convert to Spark DataFrame
df = spark.createDataFrame(carparks)
```

```
# Save to Fabric Lakehouse (Delta format)
carpark_final.write.format("delta").mode("append").save("Tables/CarparkAvailability")

print(f"✅ Ingested {carpark_final.count()} carpark records into Lakehouse.")
```

- Command executed in 17 sec 66 ms by TAN JIA HUI, JOY on 3:44:48 PM, 8/16/25

Diagnostics 1

Ingested 1910 carpark records into Lakehouse.

b) **Step 2**: Based on the CarparkAvailability data, this step of **Historical Aggregation** is like a **bridge between raw API data and RAG-ready knowledge**. Instead of embedding millions of raw rows, the **condensed hourly summaries** are embedded which resulted in a much cheaper, faster, and higher signal-to-noise ratio.

- **CarparkAvailability** is updated **every 5 mins**, which is too granular (and noisy) for embeddings or user-facing insights.
- By computing **hourly averages**, the data is smoothed, as it captures the *trend* (e.g. "Carpark X usually has 30% availability at 6pm").
- The code to run (Carpark_Aggregator) is scheduled to run on an **hourly** basis, with **overwrite** mode.
- carpark_hourly_agg table has numbers (carpark_number, hour, avg_availability, samples), which LLMs do not work well with just numbers. They work best with **natural language summaries** that describe those numbers. For example, **"**On 2025-08-10 18:00, carpark J39 had an average availability of 35% based on 12 samples.". That is where the **context** column needs to come in, which explains the data point in plain English.

c) **Step 3**: In the RAG_Embedding_Generator, the **context** column (converts **numbers into text** that embeddings can capture meaning from) is added before it is passed to the RAG Embedding Generator.

```python
from pyspark.sql.functions import concat, lit, round, col, hour

# Load hourly stats + carpark info
hourly_agg = spark.table("carpark_hourly_agg")
carpark_info = spark.read.csv("Files/HDBCarparkInformation.csv", header=True, inferSchema=True)
carpark_info = carpark_info.withColumnRenamed('car_park_no', 'carpark_number')

# Create human-readable contexts
contexts = (hourly_agg
    .join(carpark_info, "carpark_number")
    .withColumn("context",
        concat(
            col("address"), lit(" averages "),
            round(col("avg_availability")*100, 1), lit("% availabililty at "),
            hour(col("hour")), lit(":00")
        )
    )
)

# Generate embeddings (using your existing UDF)
embeddings_df = contexts.withColumn("embedding", get_embeddings_udf(col("context")))

# Save for RAG
embeddings_df.write.format("delta").mode("overwrite").saveAsTable("carpark_embeddings")
```

ABC context
BLK 270/271 ALBERT CENTRE BASEMENT CAR PARK averages 38.5% availabililty at 22:00
BLK 98A ALJUNIED CRESCENT averages 59.8% availabililty at 22:00
BLK 101 JALAN DUSUN averages 99.6% availabililty at 22:00
BLK 227 ANG MO KIO STREET 23 averages 33.7% availabililty at 22:00
BLK 308C ANG MO KIO AVENUE 1 averages 29.0% availabililty at 22:00
BLK 260 ANG MO KIO STREET 21 averages 55.7% availabililty at 22:00
BLK 309B ANG MO KIO STREET 31 averages 69.1% availabililty at 22:00
BLK 316B ANG MO KIO STREET 31 averages 68.2% availabililty at 22:00
BLK 588 ANG MO KIO STREET 52 averages 27.5% availabililty at 22:00
BLK 700 ANG MO KIO AVENUE 6 averages 50.0% availabililty at 22:00
BLK 590 ANG MO KIO STREET 51 averages 64.7% availabililty at 22:00
BLK 596 ANG MO KIO STREET 52 averages 57.0% availabililty at 22:00
BLK 130A ANG MO KIO STREET 12 averages 38.7% availabililty at 22:00

d) **Step 4**: There are various components in the **Query-Time Retrieval.**

(i) Convert the user query into an embedding

```python
query_embedding = get_query_embedding(user_query)
```

- Use **OpenAI text-embedding-3-large** model.
- Convert the natural-language query into a numerical vector (length = 3072).
- This allows to measure *semantic similarity* between the query and stored historical summaries.

(ii) Retrieve the most relevant historical context (from embeddings)

```python
historical_context = search_embeddings(query_embedding, top_k=3)
```

- Loads the **carpark_embeddings** table from the Fabric Lakehouse.
- Computes **cosine similarity** between the query vector and each stored vector.
- Returns the **top 3 most relevant historical summaries** to feed into the LLM.

(iii) Fetch the latest real-time availability

```python
latest_data = get_latest_carpark_data(limit=10)
```

Read from the **CarparkAvailability** table in Fabric.
Get the **latest rows**, sorted by update_datetime.
Provide fields: carpark_number, lots_available, total_lots, address and update_datetime.
This ensures the final answer is not only historical but reflects the **current state** of carparks right now.

(iv) Combine context + real-time data into an LLM prompt

```python
prompt = f"""
You are an assistant analyzing Singapore carpark availability.

User question: {user_query}

Historical insights (retrieved from embeddings):
{historical_context}

Latest real-time availability:
{latest_data}

Provide a concise but comprehensive answer.
"""
```

- The retrieved **historical insights** (from embeddings) and **latest availability** (from raw table) are merged into one structured prompt.
- Sent to the **OpenAI gpt-4o-mini** model, which reasons over both sources and generates a **context-aware answer**.
- Output as below.

```
    Provide a concise but comprehensive answer.
    """

    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": "You are a carpark insights assistant."},
            {"role": "user", "content": prompt}
        ]
    )

    return response.choices[0].message.content


# Example usage inside Fabric notebook
answer = generate_answer("Which area has the highest and lowest Carpark Availability?")
print(answer)
```

✓ 36 sec - Command executed in 36 sec 994 ms by TAN JIA HUI, JOY on 5:33:54 PM, 8/17/25

> Spark jobs (8 of 8 succeeded) Resources Log

> Diagnostics 1

```
Based on the latest real-time availability data, the area with the highest carpark availability is:

- **HG3L** (Buangkok Link) with **43% availability** (408 out of 725 spaces).

The area with the lowest carpark availability is:

- **HG41** (648 Hougang Avenue 8) with **0% availability** (0 out of 461 spaces).

In summary:
- **Highest Availability**: HG3L (43%)
- **Lowest Availability**: HG41 (0%)
```

e. Performance Optimization

```
# cache frequently used data
embeddings_df.cache()
```

13. Business Value

- Provide live operational awareness for parking management.
- Support strategic planning with location-based usage trends.
- Reduce manual analysis time with automated AI summaries.
- Enable map-based insights for policy and resource allocation, hence helping decision-makers identify high-demand areas.

This project demonstrates a unique combination of data engineering, AI integration, and geospatial analytics. It showcases production-ready design patterns that can be adapted to various real-time analytics scenarios.

14. Improved System Architecture with LLM / RAG

- **API** → Fetches raw carpark availability data.
- **CarparkAvailability** → Stores/processes raw availability data.
- **Aggregation** → Aggregates data (e.g., by time, location, etc.).
- **CarparkHourlyAgg** → Stores hourly aggregated data.
- **RAG_Embedding_Generator** → Generates embeddings (for Retrieval-Augmented Generation).
- **Carpark_Embeddings** → Stores vectorized embeddings for retrieval.
- **Query Retrieval** → Retrieves relevant embeddings based on user queries.
- **AI Model** → Processes queries and generates insights.
- **Power BI** → Visualizes data and AI-generated insights.

Security Note ⚠ **Remove your API key** from the code! Use environment variables instead:

```
import os
from openai import OpenAI
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))  # Set key in your environment
```