MindHive

Technical Assessment

Documentation

&

Instruction

By

Wong Horng Woei

Table of Contents

[Introduction 3](#_Toc194678721)

[Frameworks and Libraries 3](#_Toc194678722)

[Prerequisite 4](#_Toc194678723)

[Part 1: Web Scraping and Database Storage 5](#_Toc194678724)

[Part 2: Geocoding 6](#_Toc194678725)

[Part 3: API Development 7](#_Toc194678726)

[Part 4: Frontend Development and Visualization 10](#_Toc194678727)

[Part 5: Chatbot Functionality 14](#_Toc194678728)

[Limitation and Future Enhancement 17](#_Toc194678729)

# Introduction

This project is a **McDonald's outlet locator and chatbot system** that enables users to:

* Find McDonald's stores in Malaysia based on location.
* Display outlets on an interactive map with a **5KM radius catchment area**.
* Highlight **intersections** between different store catchment areas.
* Query store information (e.g., **24-hour operation, birthday party availability**) using an **LLM-powered chatbot**.

# Frameworks and Libraries

|  |  |  |
| --- | --- | --- |
| **Component** | **Technology Used** | **Reasoning** |
| Web Scraping | requests, json, sqlite3 | Simple and effective for extracting and storing McDonald's data. |
| Backend | FastAPI (Python) | High performance, supports async operations, and easy to integrate with external APIs. |
| Frontend | React | React offers a modular frontend. |
| Database | SQLite | Lightweight and easy to set up, suitable for this use case. |
| Chatbot | Together AI (Llama-2-70B) | Provides natural language processing capabilities without requiring local LLM hosting. |
| Hosting | Local / Cloud Deployment (Optional) |  |

# Prerequisite

Ensure you have the following installed:

* Python 3.9+
* Node.js & npm (for frontend)
* SQLite
* pip (Python package manager)
* Virtual environment (recommended)

# Part 1: Web Scraping and Database Storage

**Objective**

To extract McDonald’s outlet data from the [official website](https://www.mcdonalds.com.my/locate-us) and store it in a structured SQLite database.

**Implementation Steps**

1. **Scrape Data**

* Send an HTTP request to the **McDonald's Malaysia Store Finder API**.
* Extract relevant store details:

- **Name, Address, Latitude, Longitude, Operating Hours, Waze Link Telephone, Email, and features such as birthday parties, wifi, McCafe, McDeliveries etc**.

2. **Store Data in SQLite**

* Create a table (stores) in mcdonalds\_stores.db.
* Insert data while **preventing duplicates**.

**Key Challenges & Solutions**

|  |  |
| --- | --- |
| **Issue** | **Solution** |
| API request failed | Used proper headers and payload to simulate a browser request. |
| Some stores lacked operating hours | Defaulted to “N/A” unless tagged as “24 Hours”. |
| Duplicate stores in DB | Used UNIQUE(name, address) to prevent duplication. |

# Part 2: Geocoding

**Objective**

For each McDonald's outlet, retrieve its **latitude and longitude** based on the **stored address** using the **Google Maps Geocoding API**.

**Implementation Steps**

1. **Send a request** to the **Google Maps API** with the store address.
2. **Parse the response** to extract **latitude & longitude**.
3. **Update the SQLite database** with the geocoded coordinates.

**Key Features & Reasoning**

|  |  |
| --- | --- |
| **Features** | **Reason** |
| Google Maps API | Ensures accurate geocoding compared to other open services. |
| Automated | Made it reproducible for future data updates. |

**Key Challenges & Solutions**

|  |  |
| --- | --- |
| **Issue** | **Solution** |
| Google Maps API rate limits | Added a delay between requests to avoid hitting limits |

**How to Run**

1. Install dependencies (requests, pandas, sqlite3).

2. Run the preprocessing script: python geocoding\_p2.py

# Part 3: API Development

**Objective**

To build a **FastAPI backend** that:

* Serves McDonald's store data from an SQLite database.
* Integrates **Google Maps Geocoding API** to fetch missing coordinates.
* Provides an API for the **React frontend & chatbot integration**.
* Enables **CORS** for frontend access.

**Implementation Steps**

**Step 1: Set Up the FastAPI Framework**

* Use **FastAPI** to create a **lightweight and high-performance API**.
* Enable **CORS (Cross-Origin Resource Sharing)** to allow the frontend to communicate with the backend.

**Step 2: Connect to the SQLite Database**

* Establish a connection with an **SQLite database** that stores McDonald's outlet information.
* Ensure the database can return **store details**, including **address, operating hours, features (WiFi, McCafé, cashless payments, etc.), and contact details**.

**Step 3: Implement Google Maps Geocoding API**

* Fetch **latitude and longitude** for stores that are missing location data.
* Store the geocoding results in the database to **minimize API requests**.
* Secure the **Google API key** using environment variables to **prevent exposure of sensitive credentials**.

**Step 4: Create API Endpoints**

* Build an **endpoint (/stores)** that returns a **list of all McDonald’s outlets** with details.
* Add a **root endpoint (/)** for basic server health checks.
* Integrate a **chatbot API endpoint** that allows users to query store details using natural language.

**Step 5: Structure Data for Better Readability**

* Organize store data into **separate sections**:
  + - **Basic information** (name, address, operating hours, Waze link).
    - **Location details** (latitude, longitude).
  + **Contact details** (telephone, email).
  + **Store features** (WiFi, McCafé, birthday parties, cashless payments, etc.).

**Step 6: Handle Common Issues & Edge Cases**

* Implement error handling for **missing or incorrect address data**.
* Ensure that stores without coordinates are **automatically updated with geolocation data**.
* Optimize database queries to ensure **efficient data retrieval**.

**Step 7: Run & Test the Backend**

* Start the FastAPI server and ensure the **API is accessible** via the frontend.
* Test API endpoints using **Postman or a web browser** to confirm the data is returned correctly.
* Verify chatbot integration and store data responses.

**API Endpoints**

|  |  |  |
| --- | --- | --- |
| **Endpoint** | **Method** | **Description** |
| /stores | GET | Retrieves all McDonald’s stores with details. |
| / | GET | Returns a welcome message  . |
| Chatbot API | POST | Queries store details (from chatbot\_p5.py) |

**Key Features & Reasoning**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Technology Used** | **Reasoning** |
| FastAPI | FastAPI (Python) | High-performance and easy API development |
| Database | SQLite | Lightweight and easy to manage local database |
| Geocoding | Google Maps API | Ensures accurate location data. |
| CORS Middleware | FastAPI Middleware | Allows frontend (React) to access API. |
| Environment Variables | Python-dotenv | Secures API keys and sensitive data. |
| Chatbot Integration | Together AI (Llama) | Enables user queries for store details. |

**Key Challenges & Solutions**

|  |  |
| --- | --- |
| **Issue** | **Solution** |
| Frontend CORS restriction issues. | Enabled CORS middleware in FastAPI |
| Store features were not structured well | Grouped store attributes into contact details and features. |

**Running the Backend**

Firstly, activate a virtual environment in Python on Windows, run: **.venv/Scripts/activate**

Then, to start the FastAPI server, run: **uvicorn fastapi\_p3:app --reload**

API will be available at: [**http://127.0.0.1:8000/**](http://127.0.0.1:8000/)

# Part 4: Frontend Development and Visualization

**Objective**

To develop a responsive, interactive frontend that displays all McDonald's outlets on a map, visualizes 5KM catchment areas around each outlet, and highlights intersection zones.

**Implementation Steps**

1. Set up React app and install @react-google-maps/api.
2. Fetch outlet data from FastAPI backend.
3. Plot each outlet as a marker on the map.
4. Draw a 5KM circle around each outlet.
5. Highlight intersecting catchment areas with a different color.

**Key Features & Reasoning**

|  |  |
| --- | --- |
| **Feature** | **Reasoning** |
| Used Google Maps with @react-google-maps/api | Switched from Leaflet to Google Maps for better performance, smoother user experience, and richer built-in geometry support (e.g., circle, polygon, marker). |
| Displayed store locations as map markers | To allow users to visually identify McDonald's outlet positions across the map. |
| Drew 5KM radius catchment circles around each outlet | To illustrate each store’s service reach and support catchment area analysis. Implemented using Circle components from the Google Maps API. |
| Calculated intersection areas between circles | To identify overlapping service areas. Geometry-based logic can be implemented using the Google Maps Geometry library. This aids in location planning and competitive analysis. |
| Displayed outlet info in clickable popups (InfoWindow) | Improves usability by allowing users to view outlet name, address, and operation details (24-hour, birthday support) when clicking a marker. |

**Key Challenges & Solutions**

|  |  |
| --- | --- |
| **Issue** | **Solution** |
| Drawing and detecting intersecting circles. | Used geometry utility functions from Google Maps API. |
| Performance issues with many circles. | Limited zoom and cluster rendering as needed. |

**Functionality**

* View McDonald’s locations on a map.
* See 5KM radius catchment around each outlet.
* Spot overlapping (intersecting) outlets at a glance.

**Running on Frontend**

1. Install frontend dependencies: **npm install**

2. Add your Google Maps API key to .env

3. Start the frontend server: **npm start**

**UI Preview**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

Figure 1:Main Page (Map with 5KM radius catchment circles)

**A screenshot of a map

AI-generated content may be incorrect.**

Figure 2: InfoWindow popup when clicking a marker.

**A screenshot of a chat

AI-generated content may be incorrect.**

Figure 3: Chatbot function on the frontend part 1.

**A screenshot of a computer

AI-generated content may be incorrect.**

Figure 4:Figure 3: Chatbot function on the frontend part 1.

# Part 5: Chatbot Functionality

**Objective**

To create an intelligent chatbot that allows users to query McDonald's outlet information using natural language (e.g., "Which stores are open 24 hours in KL?").

**Implementation Steps**

1. Environment setup and load API keys securely using dotenv.
2. Define a FastAPI router /chatbot to handle user queries.
3. Connect to mcdonalds\_stores.db and create the stores table if not present.
4. Normalize and preprocess user inputs for better SQL generation.
5. Use Llama-2 via Together API to generate an SQL query from the user's message.
6. Execute the LLM-generated query against the SQLite database.
7. Format store information for user-friendly display.
8. Add hardcoded responses for greetings, farewells, and gratitude.
9. Send back a structured response with the generated SQL, number of matches, and formatted store data.

**API Endpoint**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Endpoint** | **Query Parameters** | **Description** |
| GET | /chatbot/ | query(string) | Accepts a natural language query about outlets. |

**Key Features & Reasoning**

|  |  |
| --- | --- |
| **Feature** | **Reason** |
| LLM-Powered SQL Generation | Utilizes meta-llama/Llama-2-70b-hf via Together AI to intelligently convert natural language into executable SQL queries. |
| Conversational AI Enhancements | Recognizes greetings, farewells, and gratitude, returning polite, context-appropriate responses for better user experience. |
| Query Preprocessing | Replaces user shorthand (e.g., “KL” → “Kuala Lumpur”) to improve LLM accuracy in SQL generation. |
| Structured Result Formatting | Store results are formatted with relevant details like name, address, GPS coordinates, features (WiFi, McCafe, etc.), and contact info. |
| FastAPI Modular Routing | Chatbot logic is encapsulated in a router module under /chatbot, allowing for clean separation and scalability. |
| Hardcoded Rule Handling | Special cases (e.g., 24-hour in KL or birthday party availability) are handled manually to ensure precision and reliability. |

**Key Challenges & Solutions**

|  |  |
| --- | --- |
| **Issue** | **Solution** |
| LLM-generated SQL sometimes being too generic or incorrect | Used detailed prompt engineering with strict rules and added regex-based preprocessing |
| Inconsistent user terminology (KL vs. Kuala Lumpur) | Normalized inputs using .lower() and re.sub() substitutions |
| Ambiguous/non-query conversations (e.g., "thanks") | Handled with conversational keyword recognition and canned responses |
| Risk of unsupported SQL types (non-SELECT) | Enforced SELECT-only queries and validated before execution |

**Functionality**

1. **Natural Language Input**  
   The user sends a query like:
   * "Find all outlets with McCafé."
   * "Which branches in KL operate 24 hours?"
   * "Where can I host a birthday party?"
2. **LLM-Powered Query Generation**  
   LangChain interprets the user query and dynamically constructs a valid SQL query to extract relevant data from the stores table.
3. **Database Search & Response**  
   The generated SQL is executed on the mcdonalds\_stores.db file. Results are returned as structured JSON.
4. **Frontend Integration**  
   The chatbot response can be integrated into a dedicated chat UI or returned as plain text. (see Part 4 UI Preview)

**Running the Chatbot**

1. Install dependencies: **pip install fastapi uvicorn python-dotenv together sqlite3**
2. Set your Together AI API key in .env file: **TOGETHER\_API\_KEY=your\_actual\_key\_here**
3. Run the FastAPI server: **uvicorn fastapi:app –reload**
4. Access the chatbot endpoint at: **http://localhost:8000/chatbot/?query=Find 24 hour McDonald's in KL**

**OR**

Use the frontend to use the chatbot

# Limitation and Future Enhancement

**Limitations**

The current chatbot implementation, while functional and able to handle natural language queries, still presents several limitations. One of the primary issues is the **latency** introduced by using a large-scale language model like Llama-2 70B hosted via Together AI. Because the model processes complex prompts and generates SQL queries in real-time, users may experience a noticeable delay, especially during peak usage or when dealing with more intricate queries.

In addition, the chatbot's performance is **highly dependent on prompt engineering**. The SQL output is only as reliable as the structure and clarity of the prompts provided. This reliance **limits** the system’s flexibility when users pose ambiguous or unusual questions, leading to **unexpected or inaccurate** query generation. Although some common cases like “24-hour outlets in KL” or “birthday party availability” are hardcoded to ensure consistency, this approach **lacks scalability** and cannot support a broader range of user queries dynamically.

Another constraint lies in the **lack of conversational memory**. The chatbot treats every query as **a standalone input** without understanding or referencing the context of previous interactions. This prevents the system from handling follow-up questions such as "What about in Petaling Jaya?" or "Are those open late too?", which require multi-turn dialogue capability.

Security-wise, while the SQL queries are **limited** to SELECT statements, the application still relies on the output of the LLM with minimal post-validation. This introduces **potential risks** if malformed or unintended SQL is generated and executed without further checks.

**Future Enhancements**

To address the above limitations and expand the chatbot’s capabilities, several improvements are envisioned. One key enhancement would be the **integration of a conversational memory** module, allowing the system to retain and reference previous user inputs. This would enable more natural, multi-turn interactions and improve the overall user experience.

To ensure robustness and security, a **SQL validation layer** could be introduced to verify that all LLM-generated queries comply with expected syntax and schema constraints before execution. Alternatively, the system could adopt a parser or query generation framework that translates natural language into SQL through a more deterministic approach. As an optimization measure, fine-tuning a smaller LLM on domain-specific query examples would improve both response time and accuracy while reducing dependence on prompt engineering.

A **lightweight classifier** could also be introduced to distinguish between general conversation and query-related messages. This would streamline processing by avoiding unnecessary API calls to the LLM for simple greetings or casual exchanges.

For a more intelligent and grounded system, a **Retrieval-Augmented Generation (RAG)** approach could be adopted. This enhancement would allow the chatbot to fetch relevant data snippets or example queries from a curated knowledge base before generating responses. Finally, integrating the chatbot into a user-friendly frontend—such as a web-based chat interface or even a voice assistant—would make it more accessible and intuitive for end-users.

With these future enhancements, the chatbot can evolve into a more responsive, secure, and interactive system capable of handling a wide range of natural language queries about McDonald's outlet features and availability.