Integrate LLM for Employee Activity Tracking Report

Xinge Zhang

April 2025

1 Initial System Design and Implementation

I designed a system that integrates a structured SQL database with a large language model (LLM) to support natural language queries. The database stores weekly employee activity records, including fields such as hours worked, meetings attended, sales performance, job title, department, and activity descriptions.

I implemented a function that generates SQL queries from user questions using an LLM. These queries are executed on the database, and the results are summarized again using the same LLM. If the query returns numeric data, the system also generates a visualization to support understanding. For this purpose, I used libraries like pandas, matplotlib, and sqlite3 in Python.

Below is a sample of the employee activity data that the system uses:

Table 1: Sample of Employee Activity Data (Vertical Format)

Field	**Value**
EmployeeID	1
WeekNumber	1
NumMeetings	8
TotalSales	10281.70
HoursWorked	44.2
Activities	Worked on sales calls. Faced tech issues.
Department	Sales
HireDate	2022-09-05
Email	richard.dawson@example.com
JobTitle	Sales Manager
Name	Richard Dawson

1.1 LLM Integration and API Setup

I used the `sqlite3` library to build the employee activity database and store the weekly records in a table called `EmployeeActivity`. I used `pandas` to load and clean the data, and `matplotlib` to create simple charts when needed.

To connect the database with a language model, I used the OpenAI API. I created a function that takes the user's question, builds a prompt that explains the table structure, and sends it to the model. The model replies with an SQL query, which I run on the database. If the result includes numbers, the system tries to plot a chart. After getting the query result, I ask the model again to summarize it in natural language.

```
import openai

client = openai.OpenAI(
    api_key="sk-xxxxxxxxxxx",
    project="proj_xxxxxxxxxx"
```

```
)

def ask_llm(prompt):
    response = client.chat.completions.create(
        model="gpt-4",
        messages=[{"role": "user", "content": prompt}]
    )
    return response.choices[0].message.content
```

I also tried using a local language model with Ollama. I tested small models like `llama2` and `mistral`, but they could not understand the prompt well or return correct SQL queries. I also tested larger models like `qwen:7b`, but my computer could not run them because they use too much memory.

Because of this, I decided to use the OpenAI GPT-4 model for this project. It is fast, accurate, and easy to use with the API. It gives good results for both SQL generation and summary writing.

2 Benchmark 1: Initial System Performance

To evaluate the initial performance of the system, I prepared a set of 20 diverse natural language queries covering different types of information needs, including totals, filtering, conditions, text search, and aggregation. These queries were passed through the LLM-based SQL generation pipeline to observe how well the model could interpret intent and produce correct queries.

Each query was handled by the `process_query` function, which prompts the LLM to generate a corresponding SQL query based on the database schema, executes the SQL query using `sqlite3`, and sends the query result back to the LLM for summarization. Below is the main logic:

```
def process_query(user_question):
    prompt_sql = "...SQL prompt with schema..."
    generated_sql = ask_llm(prompt_sql).strip()
    result_df = pd.read_sql_query(generated_sql, conn)
    prompt_summary = "...natural language summary prompt..."
    return ask_llm(prompt_summary)
```

Evaluation Result

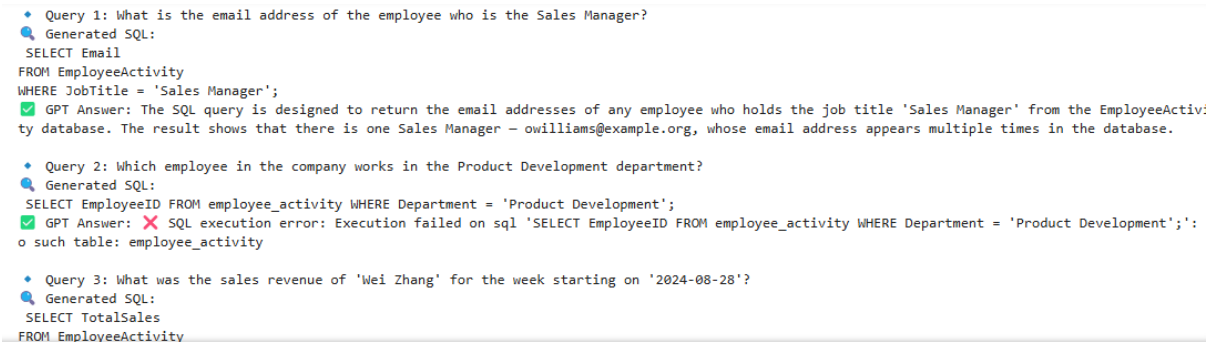

Figure 1: Benchmark 1 - Screencut of part of the result

Out of 20 test cases:

- 11 queries returned correct SQL queries and accurate summaries.
- 4 queries failed because the LLM used incorrect table names (e.g., `employee` or `employee_activity` instead of `EmployeeActivity`).
- 3 queries resulted in empty outputs due to missing data or logic gaps (e.g., invalid date logic or no match).
- 2 queries used functions like `DATEPART()` or unclear user intent (e.g., industry recession period), leading to invalid SQL or incomplete logic.

Here are a few representative examples:

Query 1: "What is the email address of the employee who is the Sales Manager?"
Generated SQL: `SELECT Email FROM EmployeeActivity WHERE JobTitle = 'Sales Manager';`
Result: Correct, returned expected email addresses.

Query 2: "Which employee works in the Product Development department?"
Result: SQL failed due to incorrect table name: `employee_activity`.

Query 4: "Who are the employees working in the Finance department?"
Result: Correct. Returned detailed weekly activity summaries for a valid employee.

Query 13: "Which employees were hired during a time of industry recession?"
Result: Failed due to lack of contextual timeframe for recession. Model responded with a generic SQL template using placeholder dates.

Query 18: "Who are the top 3 employees by total hours worked during the last 4 weeks?"
Result: Failed. Incorrect use of `EmployeeTable` as table name.

Observation

The system performed well for questions that asked for simple filters, aggregations, and summaries. However, it struggled with:

- Recognizing the correct table name (`EmployeeActivity`).
- Handling unclear or open-ended questions.
- Using SQL functions not supported in SQLite (e.g., `DATEPART`).
- Managing empty results or providing useful fallback answers.

These results indicate that although the model can produce reasonable outputs, improvements are necessary to ensure robustness and SQL validity across more complex or ambiguous queries.

3 System Improvement

After completing Benchmark 1, I identified several issues in the system that affected the performance of the LLM-generated answers. Some queries returned incorrect results, while others failed due to SQL errors or missing information. These problems were mainly caused by limitations in the initial prompt design and the structure of the database.

3.1 Schema Modification

In the first version of the database, the `Name` column was not included. Many user questions referenced employee names, which caused the LLM to generate SQL queries using unavailable fields like `Name` or to fall back to less meaningful fields like `Email`. To solve this, I added a new `Name` column to the `EmployeeActivity` table. Each row now includes the full name of the employee, making it easier for the LLM to generate human-readable and accurate queries.

3.2 Prompt Engineering

I redesigned the SQL prompt to provide more structure and clearer instructions to the LLM. In the first version, the prompt only contained a simple schema and a request to write SQL. This led to many problems such as:

- Using incorrect table names like `employee` or `EmployeeTable`.
- Using unsupported SQL functions like `DATEPART()`.
- Forgetting to use `DISTINCT` in queries for employee identity.
- Missing `GROUP BY` when using aggregation.

To address these issues, I updated the prompt with specific rules and examples. For example:

- I clearly defined the table name and column names.

- I instructed the LLM to always use `Name` when identifying employees.
- I added a rule to use `SELECT DISTINCT` when querying identity-related fields.
- I included a warning not to use SQL Server-specific functions.
- I added working examples to help the model understand query format and logic.
- I introduced a fallback response in case the question required unavailable external knowledge.

3.3 Local Model Evaluation (Optional Attempt)

I also tested a few local open-source LLMs as part of the improvement process. I attempted to run these models on my own machine to avoid API usage limits. However, the performance was not ideal. Smaller models were fast but could not follow SQL instructions well. Larger models required more computing power than my system could provide. Therefore, I decided to continue using the OpenAI GPT model via API for this project.

These changes greatly improved the performance of the system, as shown in the results of Benchmark 2.

4 Benchmark 2

After improving the database schema and refining the prompt instructions, I re-evaluated the system using the same set of 20 benchmark queries.

This version of the system includes the addition of a `Name` column and updated prompt rules that guide the language model more effectively. These changes enabled the LLM to consistently use correct table names, reference valid columns, and produce more accurate SQL syntax.

Below are several representative examples that demonstrate improvements in the LLM's SQL generation and overall answer quality.

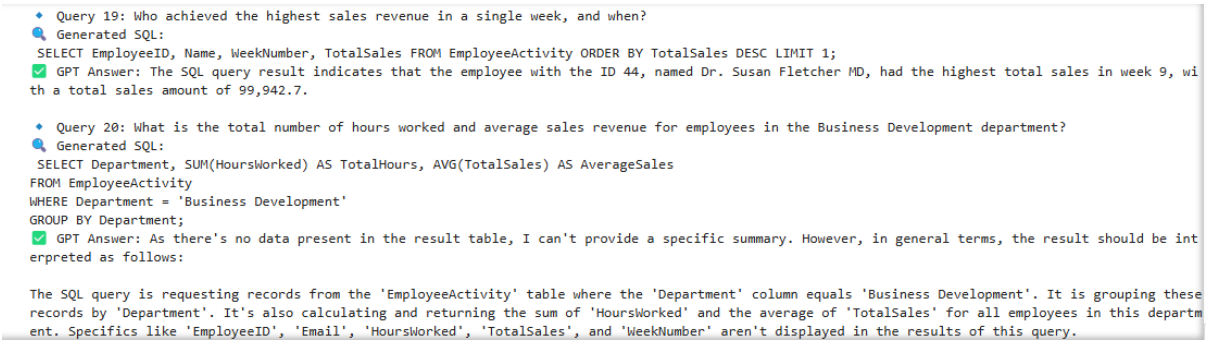

Figure 2: Benchmark 2 - Screencut of part of the result

- **Query 1:** The query correctly used `SELECT DISTINCT` and included relevant fields such as `Name` and `Email`. The result listed all employees with the job title "Sales Manager", along with accurate contact information.
- **Query 2:** The query for employees in the Product Development department returned 10 unique entries, successfully using the `Name` column to improve readability.
- **Query 3:** The sales performance of "Wei Zhang" after a specific date was retrieved accurately. This was made possible by referencing the `Name` field and filtering by `HireDate`.
- **Query 6:** The system identified all employees who worked more than 40 hours during week 1, including their names and hours worked. In Benchmark 1, this query only returned IDs.
- **Query 12:** The system successfully identified the employee who attended the most meetings in week 2 using `ORDER BY` and `LIMIT`. The result included the employee's name, which was previously missing.

- **Query 18:** The top three employees by total hours worked over the last four weeks were identified using `GROUP BY` and `SUM`, showing the LLM correctly applied SQL aggregation.

Compared to Benchmark 1, the updated system avoided previous errors such as:

- Using incorrect table names like `employee` or `EmployeeTable`.
- Calling unsupported SQL functions like `DATEPART()`.
- Failing to group results when performing aggregation.

Additionally, the LLM was able to follow prompt instructions to return cleaner queries, use aggregation correctly, and refer to employees by their names.

Out of the 20 benchmark queries:

- 17 queries executed successfully with meaningful answers.
- 2 queries were identified as unanswerable due to missing external context.
- 1 query failed to return data due to absence of matching records in the database.

These results indicate that the improvements made to both the prompt design and the schema led to a noticeable increase in answer correctness and completeness.

5 Data Visualization

To enhance the interpretability of query results, I implemented a visualization module that automatically determines and generates appropriate charts based on the structure of the SQL result table. This allows users to visually understand patterns, trends, or distributions without requiring any manual chart configuration.

5.1 Design of the Visualization Function

The function `auto_plot(df)` is responsible for selecting the most suitable chart type from three options: line chart, pie chart, and bar chart. The function begins by identifying the numeric and non-numeric columns in the result DataFrame.

5.2 Chart Selection Logic

The following logic is used to decide which type of chart to render:

Line Chart

A line chart is generated when the result contains a `WeekNumber` column. This column typically represents time and is used as the x-axis to show how a numeric value changes over time.

- **X-axis:** `WeekNumber`
- **Y-axis:** First available numeric column such as `HoursWorked`, `TotalSales`, or `NumMeetings`.
- The data is sorted by `WeekNumber` before plotting.

Pie Chart

A pie chart is used when the result contains:

- Exactly one numeric column.
- Exactly one non-numeric column.
- The numeric column name includes terms such as "`total`", "`percentage`", or "`share`".

This combination typically indicates a breakdown of a total across categories, which suits pie chart visualization. The following two figures (Figure 4 and figure 5) demonstrate the improvement after adjusting the pie chart logic. Since my OpenAI API usage has reached its limit, I was unable to programmatically regenerate the results, so the figures were captured using a phone.

```
summary, plot_path = process_query_with_plot("What is the weekly total sales trend for all employees combined?")
print(summary)

from IPython.display import Image, display
if plot_path:
    display(Image(plot_path))
```

```
🔍 Generated SQL:
 SELECT WeekNumber, SUM(TotalSales) AS WeeklySales FROM EmployeeActivity GROUP BY WeekNumber ORDER BY WeekNumber;
The SQL query results show the total weekly sales for a 10-week period. The sales for the first week were around 3.09 million. The next highest weekly sa
les came in the ninth week, at about 2.97 million. The sales for weeks 2 and 4 were also relatively high, with approximately 2.81 million and 2.85 millio
n respectively. The sales dropped to their lowest in the eighth week, which was approximately 2.49 million. For the rest of the weeks, the total sales we
re between 2.60 million and 2.77 million.
```

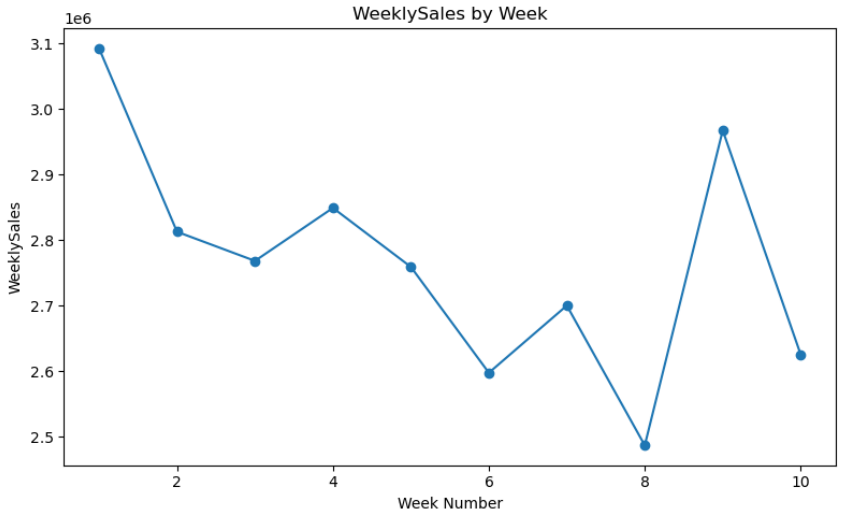

Figure 3: Line chart example

Bar Chart

A bar chart is used as the default option when the other two conditions are not met.

- **X-axis:** First non-numeric column (or inferred label column).
- **Y-axis:** First numeric column.

This is applicable for categorical comparisons where multiple groups are compared across a single numeric value.

5.3 Implementation Details

The `auto_plot` function is implemented using `matplotlib`. It performs the following steps:

1. Identifies numeric and non-numeric columns using `pandas`.
2. Applies chart selection rules based on column structure.
3. Handles missing or invalid values using type coercion and row filtering.
4. Saves the figure as `output_plot.png`.

The function returns the file path of the saved chart for later use.

5.4 Summary

This automatic charting approach simplifies the user experience and ensures that each query result is represented in the most informative format. By dynamically adapting to result structure, the system provides meaningful insights through appropriate visualizations without any manual intervention.

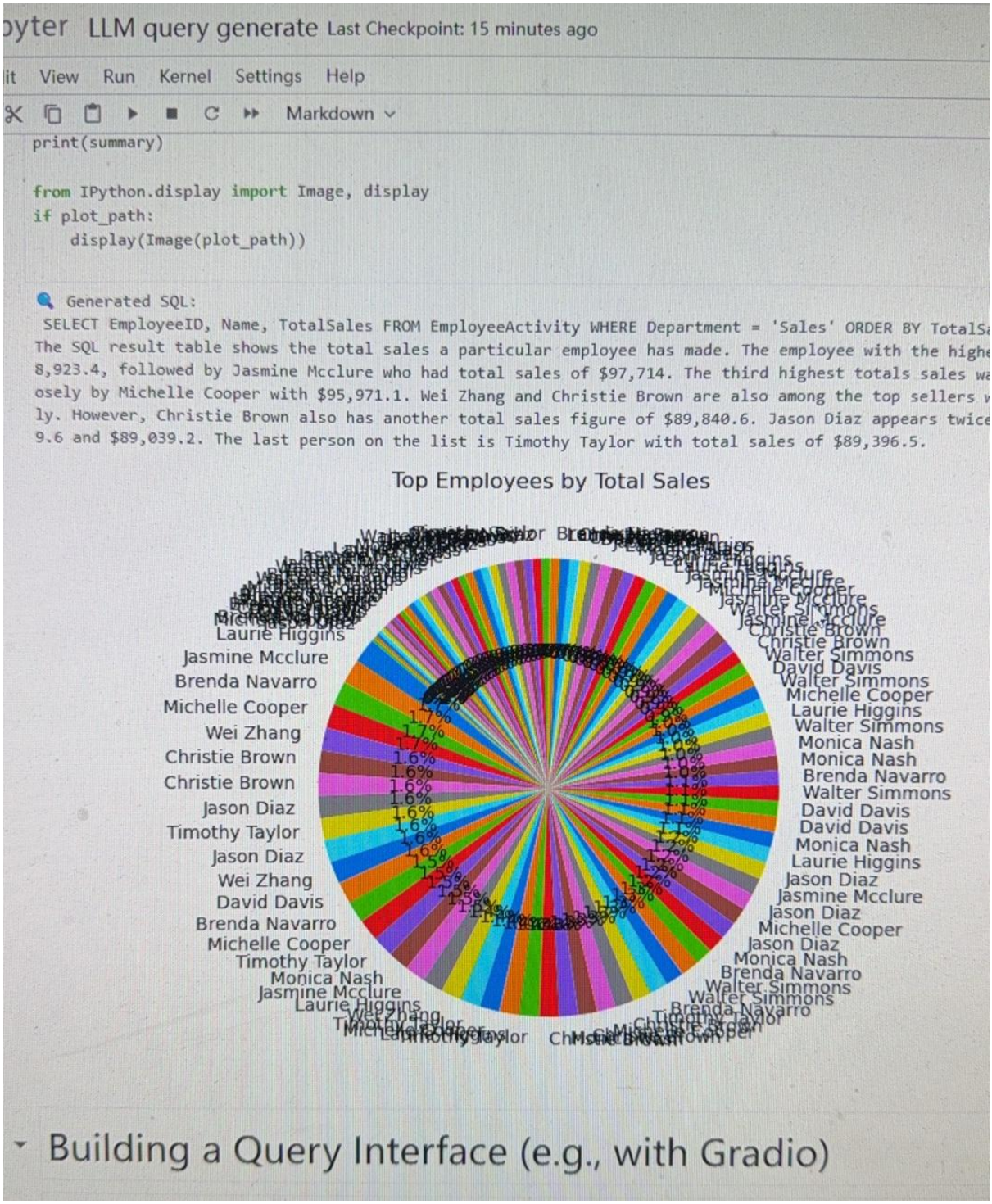

Figure 4: Pie chart - Before Adjustment

6 Building a Query Interface

To make the system more interactive, I built a query interface using the Gradio library. This allows users to ask natural language questions about the employee activity database through a simple web-based interface.

When the user enters a question, the system performs the following steps:

- Calls the language model to generate the corresponding SQL query.
- Executes the query on the SQLite database.

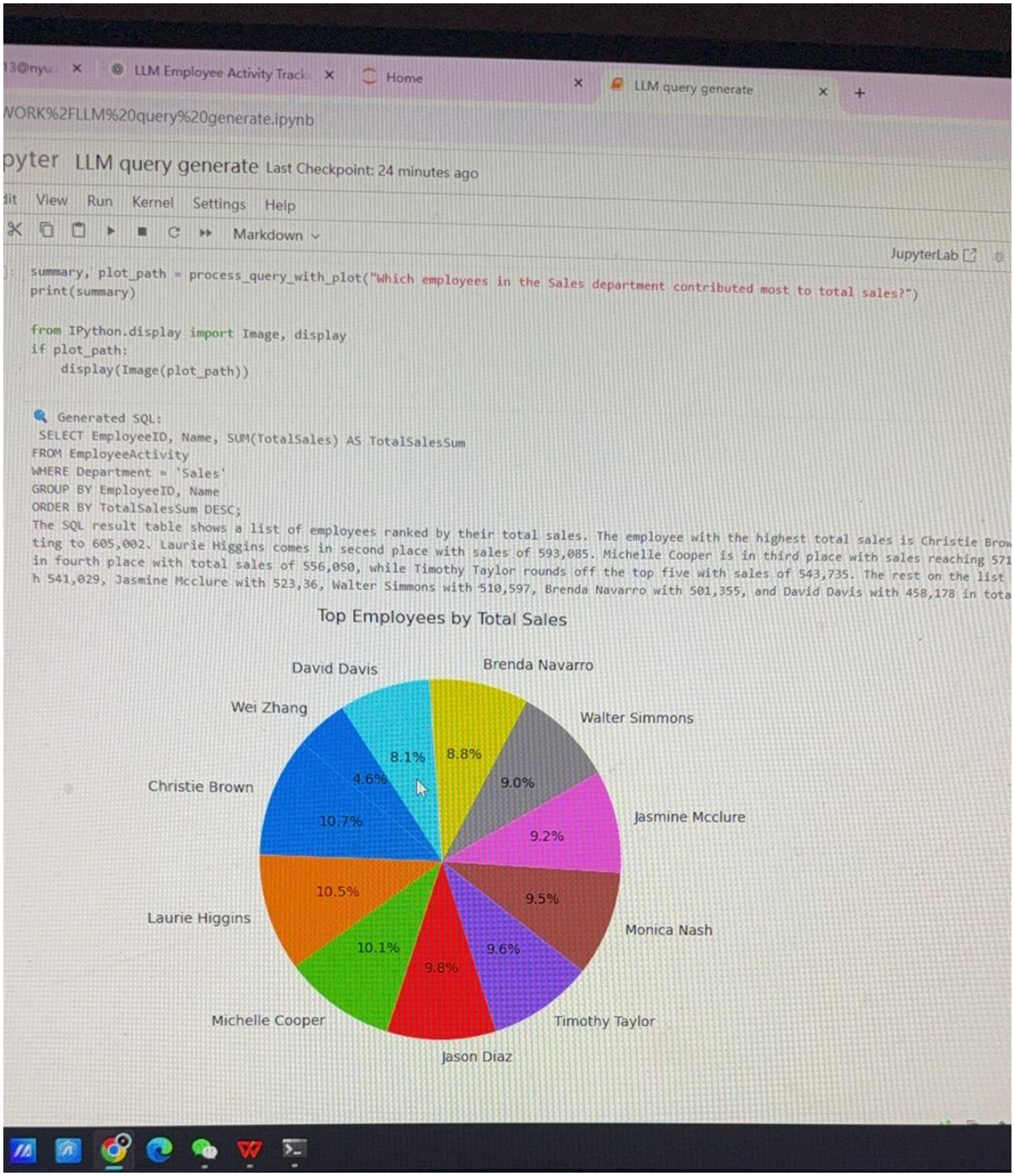

Figure 5: Pie chart - After Adjustment

- Summarizes the result using the same language model.
- Generates a chart if the result contains numeric values.

The interface includes a text input box for the question and two outputs: one text box for the LLM answer and one image area for the generated chart. The final interface is launched using `demo.launch(share=True)`, which enables public access.

Below is a screenshot of the query interface:

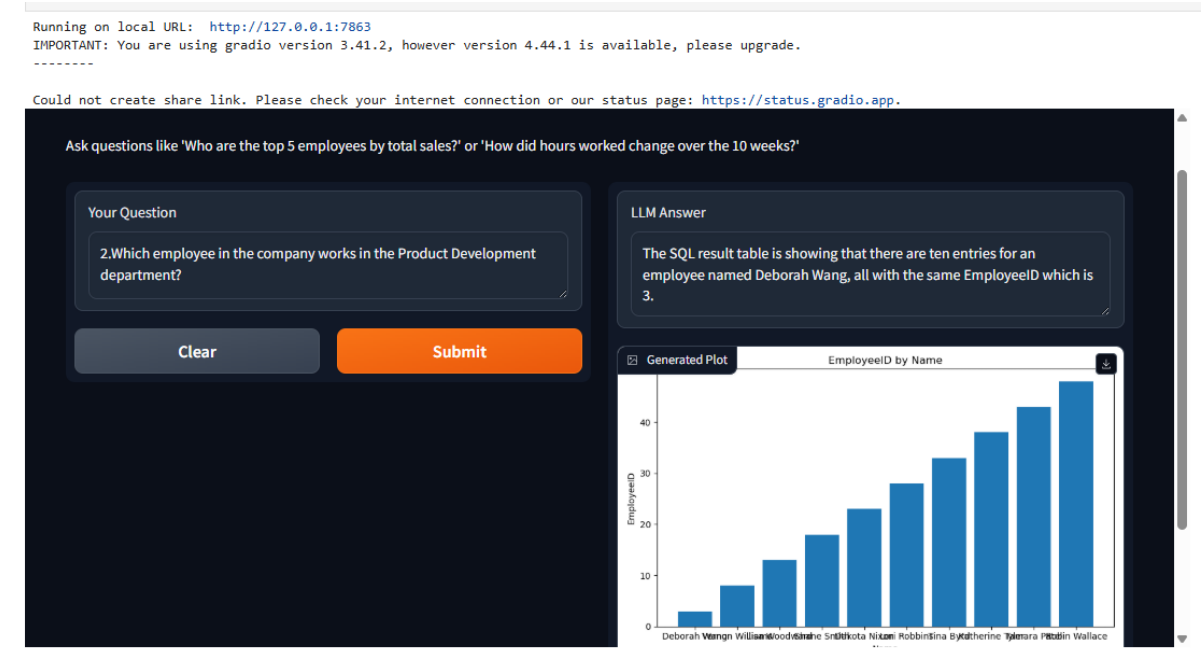

Figure 6: Gradio-powered SQL query interface

7 Analysis and Conclusion

In the initial version of the system, the database schema lacked the `Name` column, and the prompt for SQL generation was under-specified. These limitations caused frequent query failures and imprecise answers in Benchmark 1.

After identifying the main issues, I updated the database structure and designed a more detailed prompt to guide the language model. I also added rules to avoid duplicate results and adjusted the chart generation logic to better handle different data types. Benchmark 2 showed clear improvements in accuracy, completeness, and relevance of answers.

However, some potential limitations remain. For instance, when the API quota is exceeded (as shown in the followed figure), the system becomes unusable. Additionally, handling vague or ambiguous questions is still a challenge for the LLM.

Figure 7: Gradio-powered SQL query interface

Overall, this project gave me a valuable opportunity to design, debug, and refine a full-stack prototype that combines databases, LLMs, visualization, and interfaces. I appreciate the chance to explore the technical and design trade-offs of such a system.