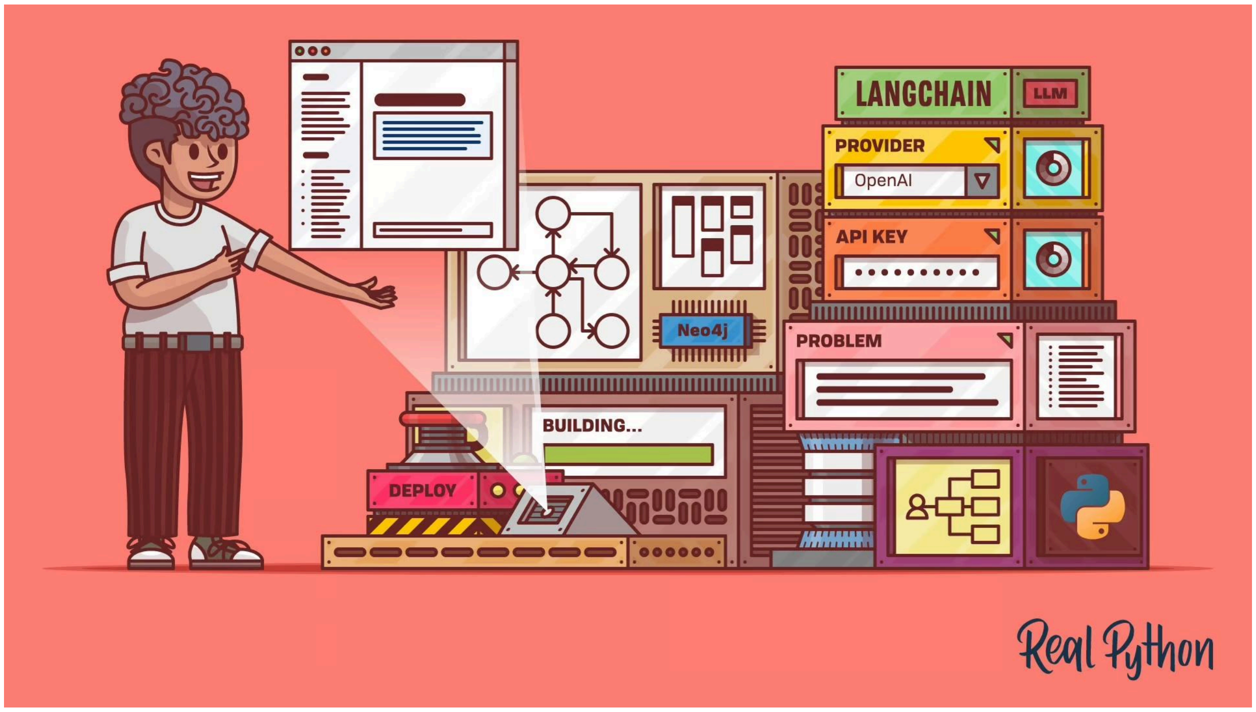
Spring 2024 Capstone Project
Mid-Term Progress Report
March 14, 2024

Data and Model Risk Management with RAG (Retrieval-Augmented Generation) Chatbot



Team Members
David Huang (th3061)
Numan Khan (nk3022)
Phillip Kim (ppk2003)
Jerry Wang (zw2888)
Tom Yu (ty2487)

Industry Mentors
Ben Carper & Jim Leach

DSI Capstone Mentor
Sining Chen

KPMG

COLUMBIA UNIVERSITY
Data Science Institute

Table of Contents

1. Introduction

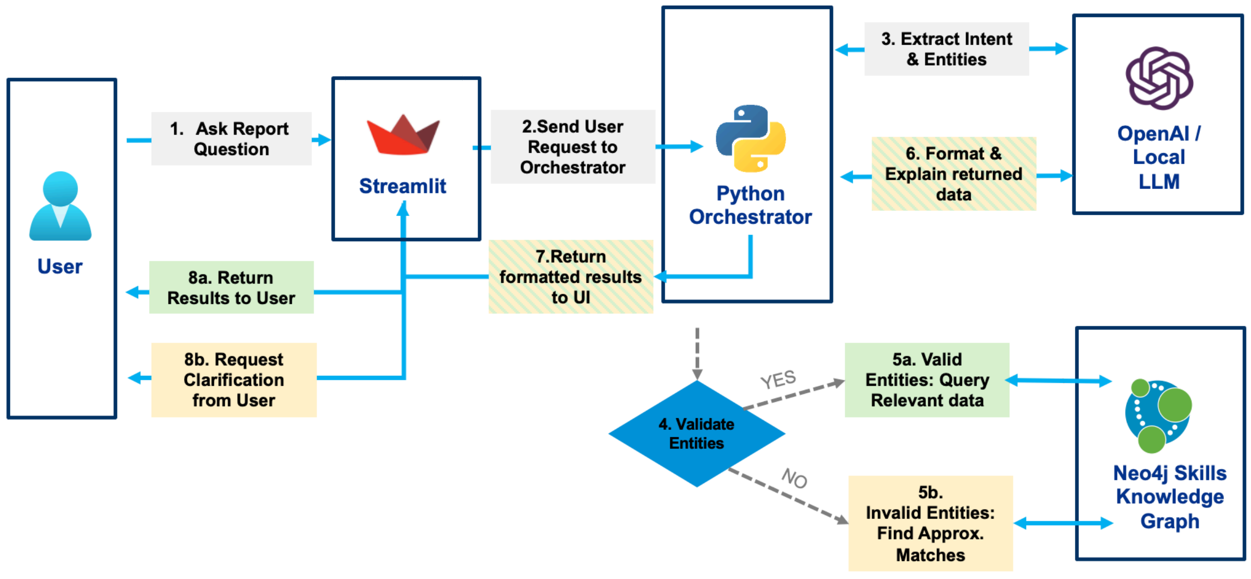
This capstone project is a collaboration between the Data Science Institute at Columbia and KPMG. It focuses on model risk management by increasing transparency of model metadata and data provenance by integrating it seamlessly into day-to-day business using a graph database and LLM to build a Retrieval Augmented Generation (RAG) chatbot.

1.1 Project Scope and Objectives

The capstone project is intended to provide hands-on experience with data science supported consulting work and provides a combination of technology and business skills. On the technical side, the main tasks include synthetic data generation, graph database data storage and retrieval, and large language model (LLM) integration including prompt engineering. On the business side, the main tasks include understanding and creating a business use case, creating written reports and presentations, and communicating with project and capstone mentors.
This project has two main parts. The first part is synthetic data generation and graph DB storage followed by the second part which is the creation of an orchestrator with a chatbot user interface that will integrate the user prompt, graph DB data, and LLM to return a natural language response to the user.

1.2 Project Concept

The mentors provided the following orchestrator flow to illustrate the concept of a RAG.

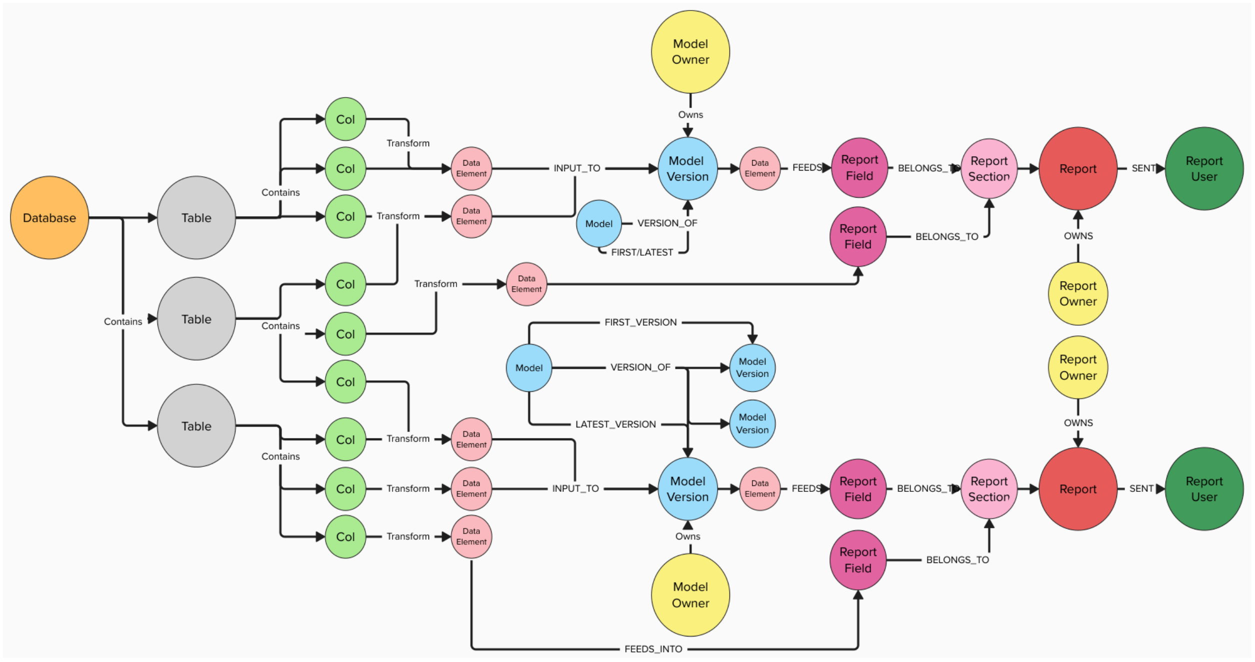

User
1. Ask Report Question
Streamlit
2.Send User Request to Orchestrator
Python Orchestrator
3. Extract Intent & Entities
OpenAI / Local LLM
6. Format & Explain returned data
7.Return formatted results to UI
8a. Return Results to User
8b. Request Clarification from User
4. Validate Entities
YES
NO
5a. Valid Entities: Query Relevant data
5b. Invalid Entities: Find Approx. Matches
Neo4j Skills Knowledge Graph

[Fig 1.1] Original Orchestrator Flow Diagram

2. Synthetic Data Generation

2.1 Synthetic Data Schema

Because this project requires a metadata schema to be created and populated, we had to generate the synthetic data from scratch with guidance from the mentors. From the examples that were given to us, it became clear that we needed to design a schema that included metadata for data sources, models and reports. The following is the initial schema that we designed.



[Fig 2.1] Data Schema Design

The data sources are represented by the Database, Table and Column nodes. The models are represented by Model and ModelVersion nodes, while the reports are represented by the Report, ReportSection, and ReportField nodes.

In terms of the data flow, the data from one or more of the columns are transformed into DataElement nodes. One or more of these data elements are then either passed into a model or directly fed into a report field for display in the report. If a data element is passed into a model, then the model will compute and output the result as a data element to be fed into a report field.

After designing the data schema, we needed to generate metadata for our hypothetical company. After researching different types of organizations, we decided to set up a company that sells various products across several geographic areas.

We primarily utilized the assistance of ChatGPT in generating the various parts of the overall schema which we will discuss further.

2.2 Datasources

We used ChatGPT to generate a database schema for our hypothetical company. After a few tries, we were able to generate something that looked quite realistic for our needs. We created ten databases each with three to four tables each with four to six columns. Fig 2.1 shows two of the databases. For example, the first database is named Executive Management and has three tables named Departments, Strategic Initiatives, and Performance Metrics. The Departments tables have columns named DepartmentID, DepartmentName, ManagerID, DepartmentBudget, Objectives, and DepartmentLocation.

Executive Management

- **Departments:** DepartmentID, DepartmentName, ManagerID, DepartmentBudget, Objectives, DepartmentLocation
- **Strategic Initiatives:** InitiativeID, InitiativeTitle, InitiativeDescription, InitiativeStartDate, InitiativeEndDate, InitiativeBudget, DepartmentID
- **Performance Metrics:** PerformanceMetricID, DepartmentID, PerformanceMetric, PerformanceTarget, PerformanceActual

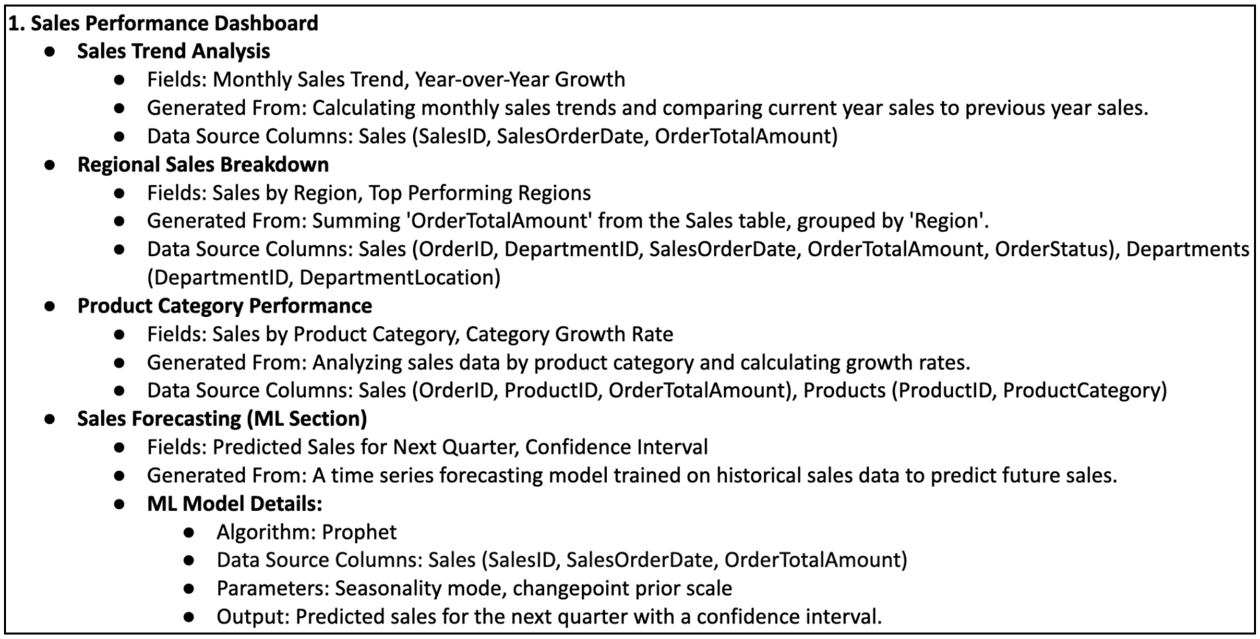
Finance and Accounting

- **Accounts:** AccountID, AccountType, AccountBalance, AccountDateOpened
- **Transactions:** TransactionID, BudgetID, AccountID, TransactionType, TransactionAmount, TransactionDate
- **Budgets:** BudgetID, DepartmentID, FiscalYear, BudgetAmount
- **Financial Reports:** ReportID, ReportType, ReportPeriod, ReportFile

[Fig 2.2] Sample Databases

2.3 Reports

After generating the database schema, we again utilized ChatGPT to generate hypothetical company reports and models based on the database schema. We asked it to include specific columns from the database tables that feed into a report and how each report field is calculated.



1. Sales Performance Dashboard

- **Sales Trend Analysis**
    - Fields: Monthly Sales Trend, Year-over-Year Growth
    - Generated From: Calculating monthly sales trends and comparing current year sales to previous year sales.
    - Data Source Columns: Sales (SalesID, SalesOrderDate, OrderTotalAmount)
- **Regional Sales Breakdown**
    - Fields: Sales by Region, Top Performing Regions
    - Generated From: Summing 'OrderTotalAmount' from the Sales table, grouped by 'Region'.
    - Data Source Columns: Sales (OrderID, DepartmentID, SalesOrderDate, OrderTotalAmount, OrderStatus), Departments (DepartmentID, DepartmentLocation)
- **Product Category Performance**
    - Fields: Sales by Product Category, Category Growth Rate
    - Generated From: Analyzing sales data by product category and calculating growth rates.
    - Data Source Columns: Sales (OrderID, ProductID, OrderTotalAmount), Products (ProductID, ProductCategory)
- **Sales Forecasting (ML Section)**
    - Fields: Predicted Sales for Next Quarter, Confidence Interval
    - Generated From: A time series forecasting model trained on historical sales data to predict future sales.
    - **ML Model Details:**
        - Algorithm: Prophet
        - Data Source Columns: Sales (SalesID, SalesOrderDate, OrderTotalAmount)
        - Parameters: Seasonality mode, changepoint prior scale
        - Output: Predicted sales for the next quarter with a confidence interval.

[Fig 2.3] Hypothetical Sales Report Details

2.4 Models

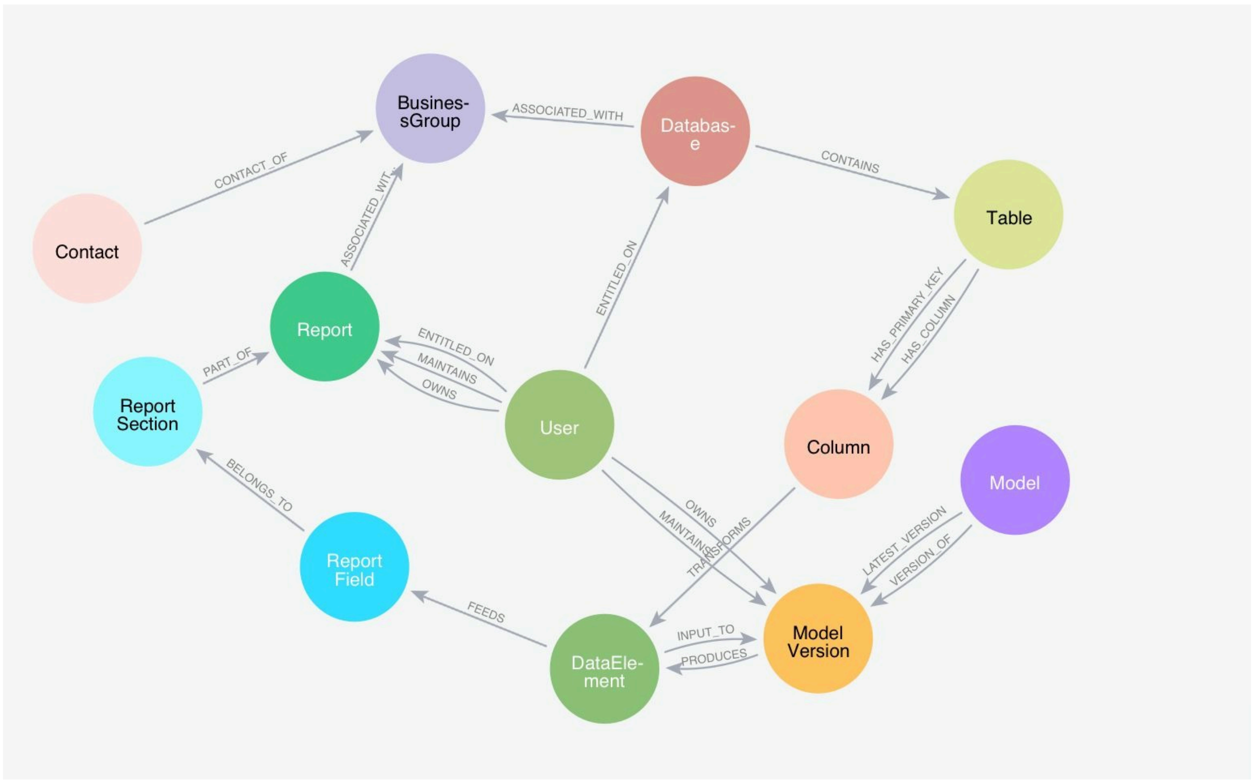
As can be seen in Fig 2.3, included in each of the hypothetical report details is a report field that is derived from a hypothetical model. These models are generally various types of statistical or machine learning prediction or forecasting models that take in various columns from the database as inputs and output values that feed into the report field. We again utilized ChatGPT to generate various model types along with its data sources, model parameters, and outputs.

2.5 Generation of JSON Files

To import our synthetic data into Neo4j, we needed to translate our database schema, models and reports into a format that could be parsed and then passed into Cypher queries (Neo4j's graph query language) that can create nodes and relationships along with their attributes. We chose to use JSON format for these files as recommended by our mentors. In looking at the overall graph DB schema, we decided to develop three separate JSON file types to represent the overall schema.

First, we generated Database JSON files to represent the databases, tables and columns, as well as ownership and user access to the databases. Second, we generated Report JSON files to represent the reports, report sections, and report fields. In addition, the Report JSON files also contained all information concerning the reports such as the data elements that feed into report fields and whether those data elements were derived directly from database columns or a model, as well as the ownership, maintainer, and distribution of the reports. The third type we generated were Model JSON files that contained information regarding models and model versions, as well as the input data elements and the columns that feed into them and output data elements that feed into report fields. In addition, model ownership, version information and various model metadata such as parameters, feature importance etc are also stored in the same file. For clarification, model nodes represent the model concept and model version nodes represent different iterations of such a model.

With these three types of JSON files (Databases, Reports and Models), we were able to capture the entire graph DB schema as illustrated in Fig 2.4 below.



[Fig 2.4] Graph DB Schema in Neo4j

3. Orchestrator

3.1 Overview

3.1.1 Service Dependencies

The RAG chatbot's orchestrator is built on the foundation of Streamlit, LangChain's Neo4j integration, and OpenAI chat completions API. Streamlit is an open-source Python library that simplifies the process of building and deploying chatbots. Streamlit's easy-to-use framework allows developers to quickly prototype and iterate on their chatbot designs, requiring minimal front-end coding. The library supports various interactive elements such as text inputs, buttons, and sliders, making it versatile for creating a dynamic chat experience.

LangChain's integration with Neo4j allows developers to connect to a Neo4j graph database and directly query it using natural language. This is achieved by initializing a Neo4jGraph object with the database credentials and then easily retrieving the graph schema. The GraphCypherQAChain module leverages language models to interpret natural language questions and translate them into Cypher queries. This module is initialized by wrapping a language model, such as ChatGPT, and linking it to the previously established Neo4jGraph. When the chain's run method is invoked with a question, it dynamically generates and executes the corresponding Cypher query against the Neo4j database, fetching and returning relevant information directly from the graph.

```python
graph = Neo4jGraph(
  url="bolt://localhost:7687", username="neo4j", password="pleaseletmein"
)
chain = GraphCypherQAChain.from_llm(
       ChatOpenAI(temperature=0), graph=graph, verbose=True
)
chain.run("Who acted in Top Gun?")
```

[Fig 3.1] LangChain's GraphCypherQAChain class which interacts with the Neo4j database

The OpenAI chat completions API offers an advanced framework for generating conversational text responses, requiring users to specify both a model and the role of each message in the interaction. By integrating this API, developers can harness state-of-the-art LLMs like gpt-3.5-turbo to understand context, engage in meaningful dialogue, and provide relevant answers. This system also requires the categorization of messages with roles such as "system," "user," and "assistant," guiding the conversation flow.

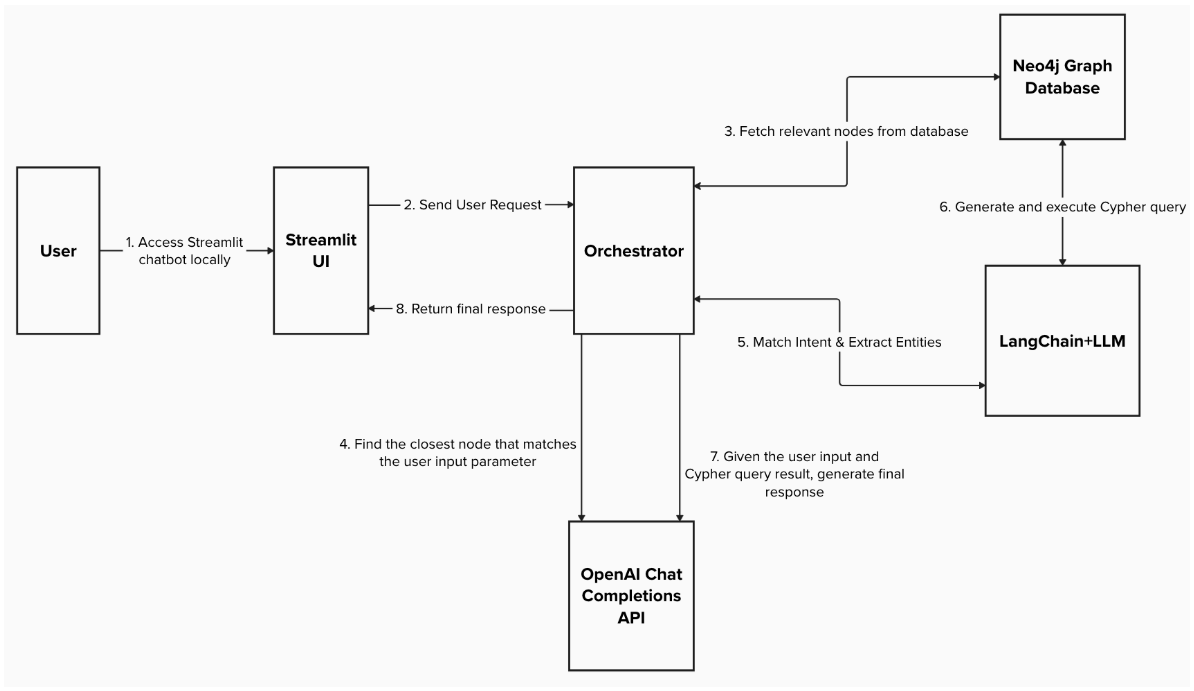
```python
from openai import OpenAI
client = OpenAI()

response = client.chat.completions.create(
 model="gpt-3.5-turbo",
 messages=[
       {"role": "system", "content": "You are a helpful assistant."},
       {"role": "user", "content": "Who won the world series in 2020?"},
       {"role": "assistant", "content": "The Los Angeles Dodgers won the World
Series in 2020."},
       {"role": "user", "content": "Where was it played?"}
 ]
)
```

[Fig 3.2] OpenAI’s Chat Completion API

3.1.2 Workflow

Figure 3.3 below depicts the chatbot’s orchestrator from the user submitting a request up until the chatbot returns a response to the user. After a user submits a request into a Streamlit-based UI, we fetch relevant data nodes in the Neo4j database to find the node name that best matches the database terminology that the user included in their request using OpenAI’s chat completion API. Then we pass the updated user request into a prompt template instructing LangChain and LLM to accurately identify the corresponding base intent request. Once the intent is determined, LangChain uses the LLM to formulate and execute a precise Cypher query based on this input. The outcome of this query, alongside the original user input, is then fed into the OpenAI’s API, which crafts the final response. This response is subsequently displayed on the Streamlit chatbot interface, providing the user with a coherent and contextually relevant answer to their query.


Neo4j Graph Database
3. Fetch relevant nodes from database
User
1. Access Streamlit chatbot locally
Streamlit UI
2. Send User Request
Orchestrator
8. Return final response
6. Generate and execute Cypher query
5. Match Intent & Extract Entities
LangChain+LLM
4. Find the closest node that matches the user input parameter
7. Given the user input and Cypher query result, generate final response
OpenAI Chat Completions API

[Fig 3.3] Orchestrator Flow Diagram

3.2 LangChain Integration

3.2.1 Defining Common User Queries

KPMG mentors recommended developing an end-to-end working prototype focused on addressing predefined base user requests. The initial phase involved brainstorming to identify the most common user requests, which covered a range of queries: performance metrics and data inputs of models, downstream and upstream information on report fields, and database permissions. As a starting point, this approach aimed at achieving an exact match between incoming user requests and the established base user requests.

3.2.2 Prompt Template

We considered the possibility of constructing a traditional classification model aimed at aligning user queries with a set of predefined common questions, thereby addressing the intent behind each query. Instead, we capitalized on LangChain's Cypher generation capabilities, specifically creating a prompt template designed to direct an LLM to craft Cypher queries pertinent to specific inquiries. This approach ensures the chatbot can be guided towards recognizing and matching the intent of user queries with designated questions. The methodology employs a comprehensive template, which encapsulates detailed instructions, the Neo4j database schema, an array of common questions, corresponding Cypher queries, and user input.

This template serves as a conduit for the GraphCypherQAChain module within LangChain's Neo4j toolkit, tasked with generating precise Cypher statements. The template outlines a clear task: to match a user's query with one of the predefined Cypher queries, modifying the latter based on the user's input to produce a final, accurate Cypher query. It meticulously details the steps, from using specific schema properties to checking if the user's intent corresponds with the base questions, and finally, updating the template Cypher query with relevant information extracted from the user's input.

3.2.3 Generating Cypher Queries

For our end-to-end prototype, our goal was to handle user requests that match exactly with single "parameter" questions, such as queries involving specific node names like a database, column, or report field name. For example, one question would be what are the performance metrics of the Customer Satisfaction Prediction Model? In this case, the single parameter provided would be the name of a model. We limited the scope of our user input to a single parameter to reduce the complexity of our chatbot as we were building an initial prototype.

The system is designed to prompt OpenAI to pinpoint and extract this parameter from the user's input. When users input the exact name of a parameter, corresponding to how it's recorded in our database, such as "IT_Database," our prototype performs flawlessly. However, our chatbot couldn't handle when a user states "IT database" instead, where the query generated would be looking for a database with the name "IT" instead of "IT_Database". To address this discrepancy, one approach we considered involves enhancing the template Cypher query using the "contains" keyword where we can find database names that contain the string "IT". While this works for some cases, we're still looking for an exact order of characters. Another effective solution entails retrieving all relevant nodes from our database and employing OpenAI's capabilities to determine the closest match to the user's input, thereby accommodating variations in how users might refer to specific database entities.

3.3 Obstacles Faced

During the development of our RAG chatbot using LangChain's GraphCypherQAChain, we encountered several challenges. Initially, while the system could retrieve responses from Cypher queries, converting these responses into final, human-readable answers proved problematic, sometimes resulting in empty outputs. To address this, we started directly accessing the intermediate steps of the LangChain execution to obtain the Cypher query responses. These were then passed alongside the user's request into the OpenAI API, bypassing the issue and ensuring reliable response generation.

Another obstacle stemmed from our use of negative keywords in our LangChain templates, which instructed the LLM on what information to exclude. Following advice from our mentors, we revised our approach to avoid such negative phrasings and instead provided examples of successful outcomes to improve clarity and effectiveness.

A significant part of our project involved identifying the most common user requests, which we determined to be queried about relevant upstream and downstream data for report fields. This realization compelled our team, initially inexperienced with Neo4j, to delve deeper into learning the Cypher query language to build effective template queries for these types of information.

Lastly, our initial prototype chatbot interface was simplistic, processing user requests and displaying single responses in a non-conversational format. Recognizing the importance of a conversational user experience, similar to that of ChatGPT and other advanced chatbots, we enhanced our chatbot by integrating session state management and chat history storage. This modification transformed our chatbot into a more interactive and engaging platform, better mimicking the natural flow of human conversation.

4. Conclusion & Next Steps

Throughout our journey in developing the RAG chatbot, we've made substantial progress in creating a system that can handle single-parameter requests with a significant degree of flexibility in parameter matching. This chatbot leverages the capabilities of LangChain and Neo4j to interpret user requests, match these with relevant database queries, and generate informative, human-readable responses. We've adopted creative strategies to overcome challenges, such as using OpenAI for intent matching, transforming Cypher query results into final responses, and finding the closest parameter by querying relevant nodes in our database.

Ethical implications should be minimal given that the RAG chatbot uses company specific data stored in a graph DB in conjunction with an LLM to provide natural language responses to user questions. As with any proprietary corporate data, as well as personally identifiable information contained in Human Resources files, should have appropriate access controls to protect corporate and employee privacy.

Several key initiatives stand out to further refine and enhance the capabilities of our chatbot. We aim to develop a more robust handling mechanism for invalid Cypher queries or when a request's intent doesn't match our predefined parameters, ensuring that users receive helpful responses even in complex scenarios. Implementing a new workflow that enables the chatbot to ask follow-up questions will facilitate a deeper understanding of user requests, particularly when initial inputs are ambiguous or incomplete.

Expanding the chatbot's knowledge to more thoroughly cover upstream and downstream data will provide users with richer, more comprehensive insights. To validate and improve our chatbot's performance, we plan to create a Python script that evaluates accuracy metrics by comparing known user inputs against expected Cypher query results.

Experimentation with various LLMs will explore how different models can optimize different aspects of the chatbot's functions, such as intent matching, Cypher query generation, and user response formulation. This tailored approach could significantly enhance the efficiency and accuracy of the chatbot's responses.

Integrating Neo4j visuals into the chatbot interface is another exciting direction, offering users a more intuitive and engaging way to understand complex database relationships. Additionally, experimenting with the inclusion of chatbot message history in subsequent user requests could provide a more contextually aware and personalized user experience.

Lastly, analyzing and reducing latency, especially after the first user request, remains a priority to ensure a smooth and responsive interaction for all users. By addressing these aspects, we aim to not only refine the chatbot's functionality but also to set a new standard for companies seeking to adopt RAG chatbots for their complex data systems.