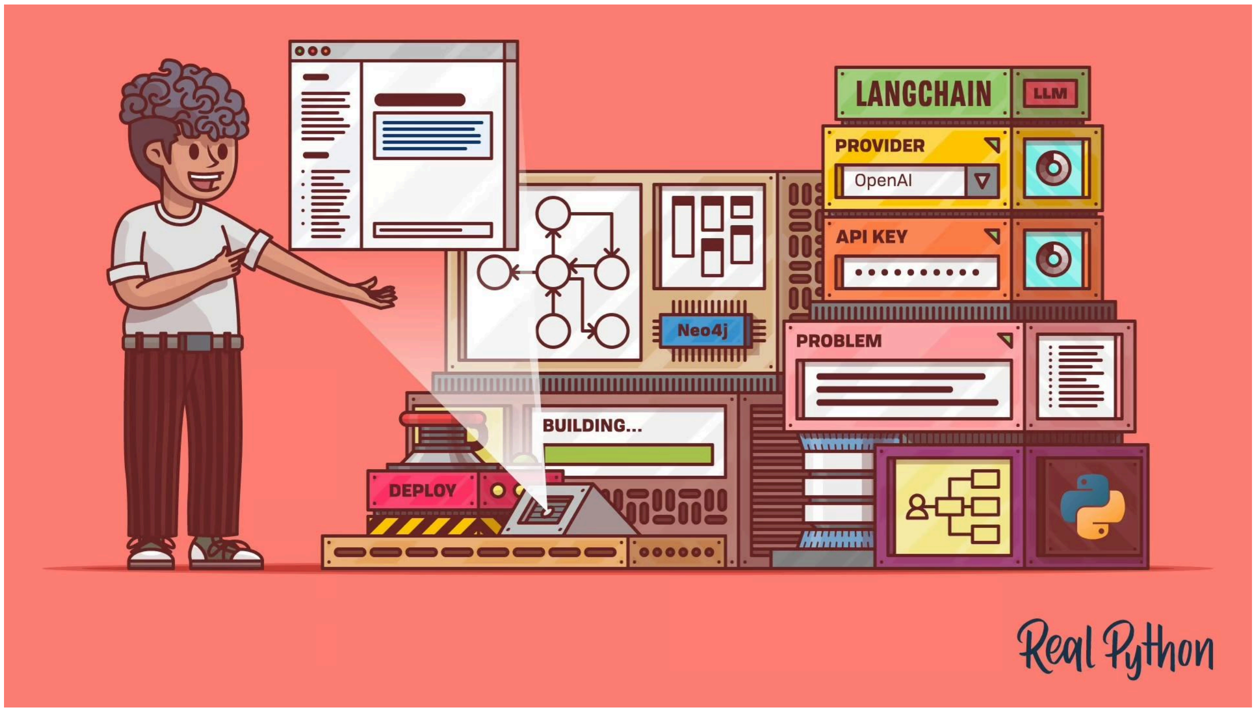
Spring 2024 Capstone Project
Final Report
May 10, 2024

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

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 AI transparency through the management of model metadata and data provenance and integrating it seamlessly into day-to-day business using a graph database and large language model (LLM) to build a Retrieval Augmented Generation (RAG) Chatbot.

1.1 Project Motivation

Companies are processing and storing an increasing amount of data every day which is used to generate reports and build predictive models. Consequently, these models help with forecasting and making vital business decisions. This reliance on an increasing amount of data and models is inherently risky and can potentially lead to operational losses. Consequently companies utilize Model Risk Management (MRM) to help identify, measure and mitigate risks from decisions based on incorrect or misused models. MRM is a holistic approach that includes, among other things, Model Validation, Model Monitoring, as well as Model Assumption Review. A Metadata RAG Chatbot can be a very useful tool in all of these components of MRM.

1.2 Why RAG Chatbot?

RAG Chatbots allows users to ask natural language questions and returns real-time answers that are more accurate and contextual without the need to learn a query language. Graph databases store data as a network of entities and relationships which can make storing metadata simpler and more intuitive for users. Metadata is especially suited to be stored in a graph database due to the flow of data from database tables, through models, and eventually ending up in reports. Graph databases also have a high level of flexibility in how data is stored, making it easier to adapt as an organization's needs evolve.

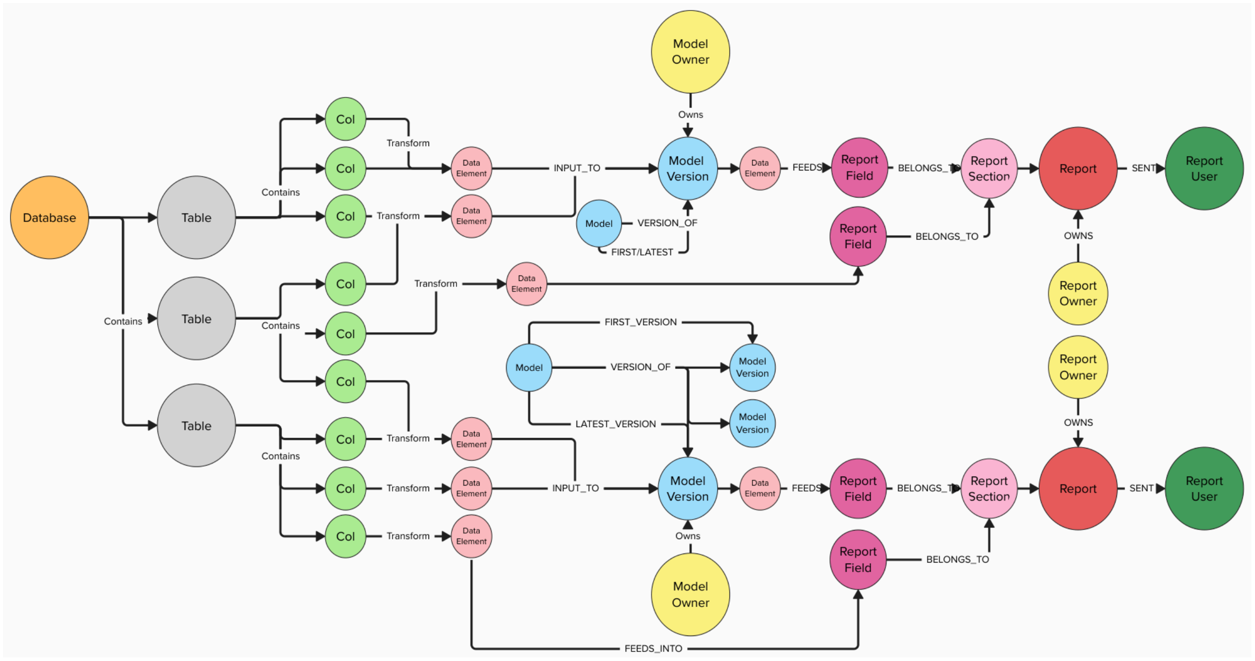
1.3 Project Scope and Objectives

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 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 database 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 database data, and LLM to return a natural language response to the user.

2. Synthetic Data Generation

2.1 Generation

To address the requirements of our project, which included the creation and population of a metadata schema, we embarked on generating synthetic data from scratch with guidance from our mentors. The design of our initial schema was carefully crafted to incorporate metadata for data sources, models, and reports.



[Fig 2.1] Data Schema Design

As outlined in Figure 2.1, our data sources were structured into Database, Table, and Column nodes, while the models and reports were organized into Model, ModelVersion, Report, ReportSection, and ReportField nodes, respectively.

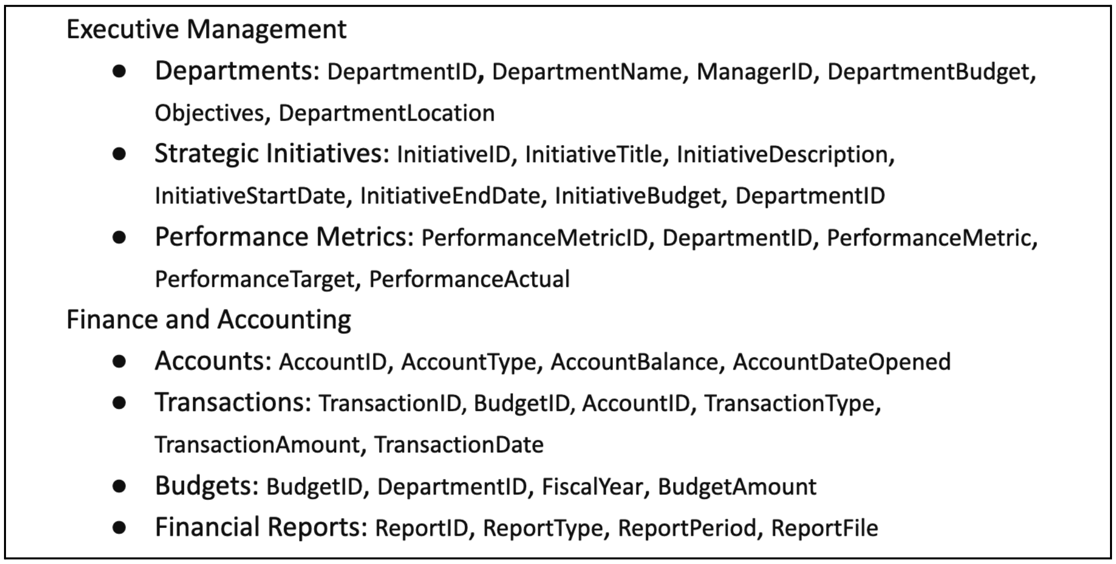
The synthetic data generation process involved mapping out the flow of data within our schema: data extracted from one or more columns were transformed into DataElement nodes. These elements could then be routed directly to report fields for display or passed through various models. When processed through a model, the output—again in the form of DataElement nodes—was used to populate corresponding fields in reports, ensuring a dynamic and interconnected data environment.

Our approach required a detailed and practical representation of a hypothetical company, crafted to simulate real-world business scenarios. This hypothetical company was conceptualized to engage in the sale of diverse products across multiple geographic regions. To populate the schema with realistic and coherent data, we leveraged the capabilities of ChatGPT, which assisted in generating the intricate parts of our schema. This included the development of various databases reflecting the company's operational and strategic layers, from executive management to sales, finance, and customer service, each tailored to specific business functions and needs.

The synthesized metadata and schema not only facilitated the testing of our RAG chatbot within a controlled environment but also allowed us to demonstrate the practical application of AI in navigating and managing complex data structures within a graph database. This foundational work was crucial for simulating realistic interactions and ensuring the robustness of our chatbot solution in handling real-life data challenges in model risk management.

2.2 Data Sources

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

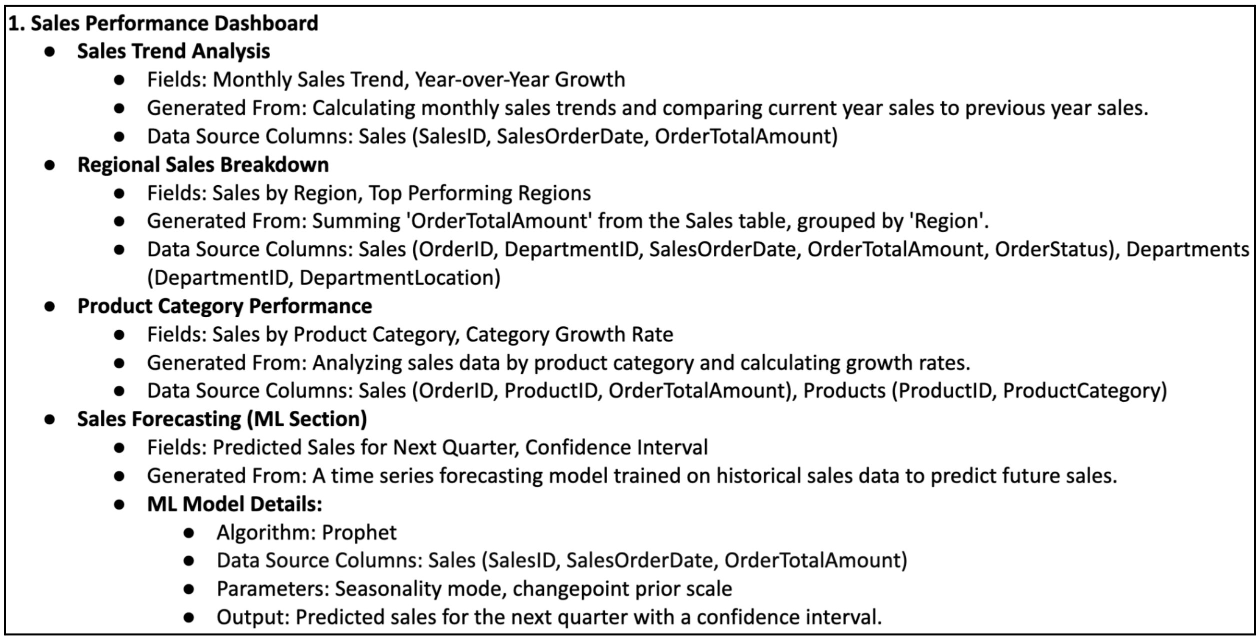
Finance and Accounting

- **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, 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

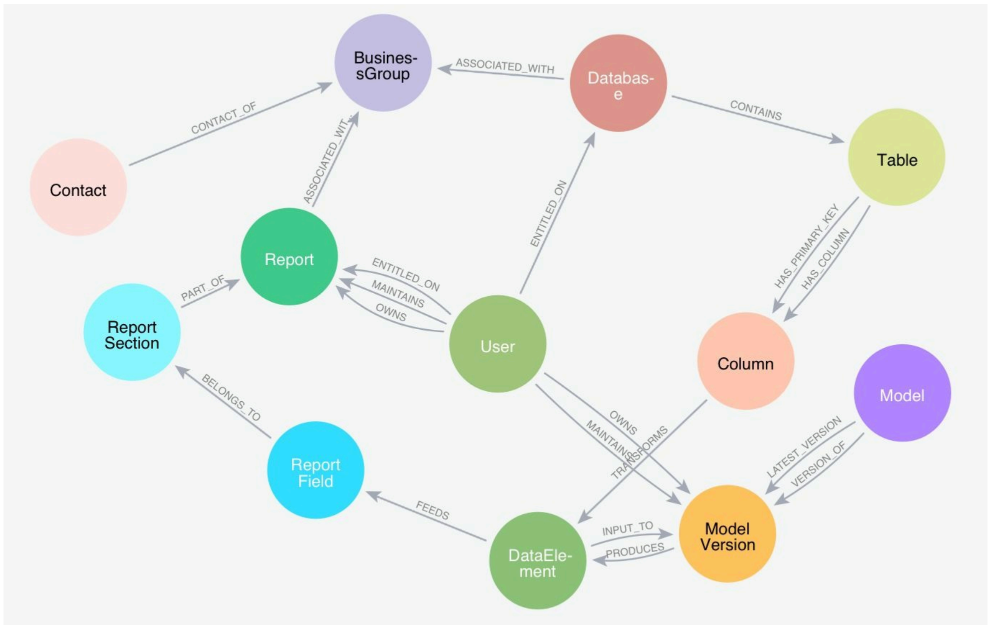
To import our synthetic data into Neo4j, we needed to translate our database schema, models and reports into a format that could be parsed and then passed into Cypher queries (Neo4j's graph query language) that can create nodes and relationships along with their attributes. We chose to use JSON format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

2.6 Data Import

Once we have JSON files containing synthetic data, the next step is to import them into the Neo4j Database. To achieve this, we utilize the 'neo4j' library in Python. After establishing a connection between Python and Neo4j, we can seamlessly execute Cypher Queries directly in Python. The JSON files allow for straightforward manipulation, enabling us to extract relevant parameters or values.

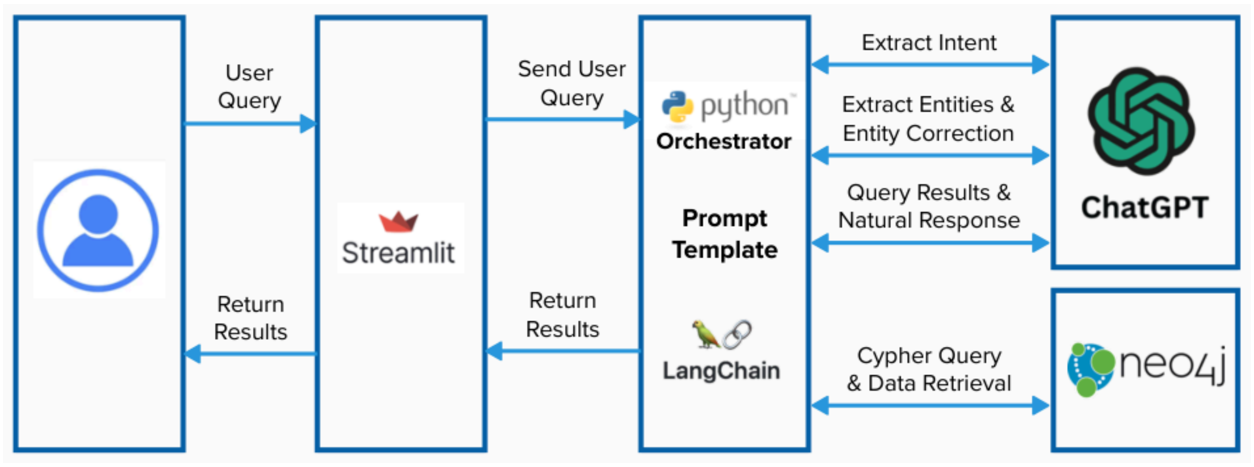
To facilitate the process, we create three essential functions: database, model, and report functions. These functions handle the execution of Cypher queries, allowing us to create nodes and edges within the Neo4j graph. By maintaining a consistent update mechanism based on the Python neo4j library, we ensured easy and efficient updates to our database.



[Fig 2.4] Graph DB Schema in Neo4j

3. Orchestrator

3.1 Service Components



[Fig 3.1] Chatbot Service Components

The RAG chatbot's orchestrator is built on the foundation of Streamlit, LangChain's Neo4j integration, and OpenAI's GPT 3.5 Turbo LLM. Streamlit is an open-source Python library that simplifies the process of building and deploying chatbot user-interfaces. 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 3.2] LangChain's GraphCypherQAChain class which interacts with the Neo4j database

OpenAI's chat completions API offers an advanced framework for generating conversational text responses, requiring users to specify both a large language 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. To avoid spending money, we leveraged GPT 3.5 Turbo and the free credits provided by OpenAI for all parts of our workflow.

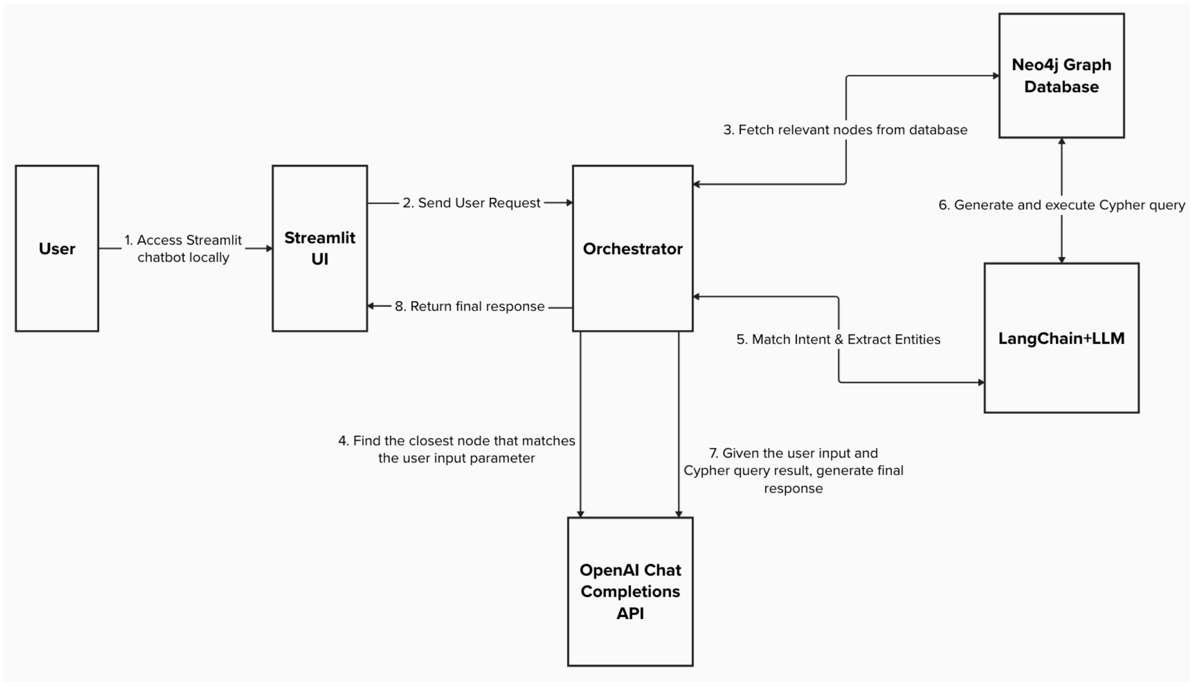
```
Python
from openai import OpenAI
client = OpenAI()

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 3.3] OpenAI's Chat Completion API

3.2 Workflow

3.2.1 Original Workflow

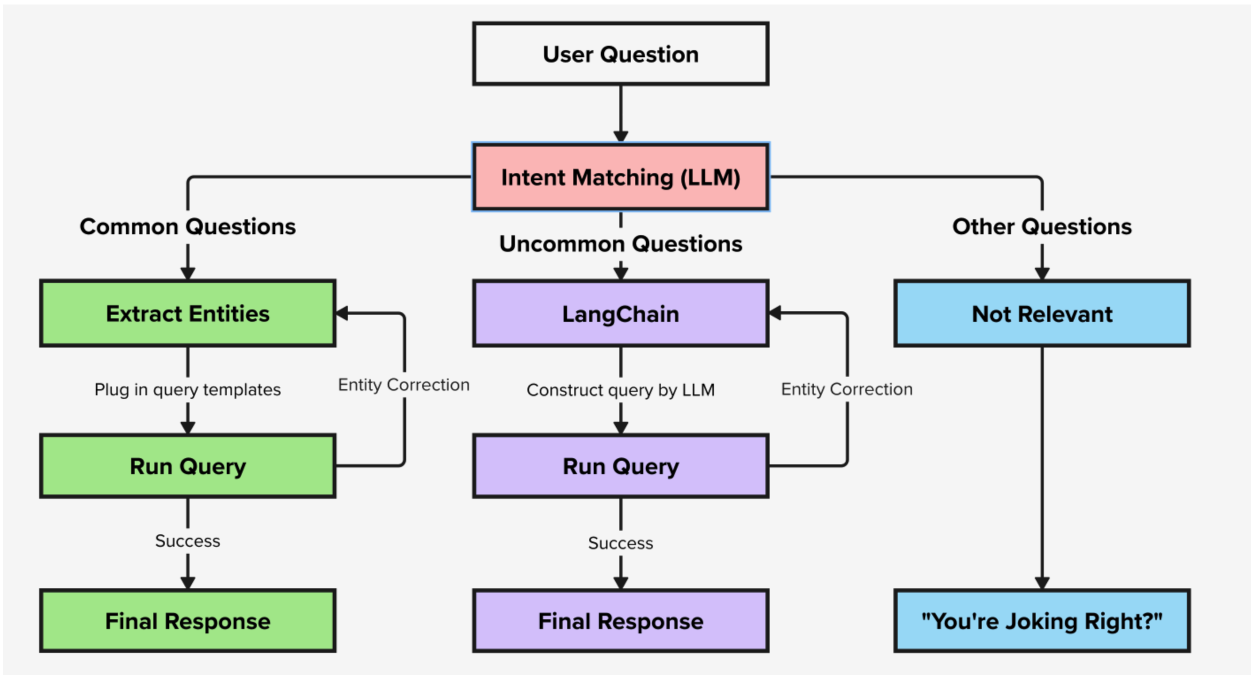
Figure 3.3 below depicts our chatbot's original orchestrator for the midterm deadline which details the user submitting a request up until the chatbot returns a response to the user. After a user submits a request into a Streamlit-based UI, we fetch relevant data nodes in the Neo4j database to find the 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 (GPT 3.5) 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
8. Return final response
Orchestrator
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] Original Orchestrator Flow Diagram

3.2.2 Final Workflow

After extensively testing the original workflow, we observed that after the first user request was fulfilled, subsequent calls to the LLM would take a long time. In addition, LangChain facilitates numerous steps when processing a single user question. Therefore, to reduce execution time, we decided to differentiate the types of questions a user could submit: common, uncommon, none. Common questions are eight questions we've identified as the most frequently asked questions by data analysts which can be seen in Section 8.1. An example of a common question is what report fields are downstream of the Performance Score column? Uncommon questions are relevant questions that require us to fetch data from our database but aren't one of the common questions. "None" or other questions are questions that are not related to the company and its business data.



[Fig 3.4] Final Orchestrator Flow Diagram

3.2.3 Common Questions

We defined separate workflows for each question type as Figure 3.4 depicts. The common question workflow involves extracting the single entity from the user question calling the LLM, plug-in the entity into predefined Cypher queries to be executed, then generating the final output using the fetched data by calling the LLM. Figure 3.5 shows an example of a predefined Cypher query that is provided with a model name parameter, and returns the performance metrics of the model's latest version. If there's a failure when executing the Cypher query, we'll attempt to correct it by looking up relevant node names in the database, update the query, and execute it again. An example of a question requiring parameter correction is "What data is upstream to a Top_Performing_Regions report field?" The orchestrator would check for any report field nodes similar to Top_Performing_Regions such as Top Performing Regions, return the corrected entity to the Cypher Query template, and then attempt to execute the query again. However, if the corrected Cypher query fails, we pass the corrected question to the uncommon workflow to ensure the chatbot does its best to answer the user's question.

```
Python
MATCH (m:Model)
WHERE m.name CONTAINS "{parameter1}"
MATCH (m)-[r1:LATEST_VERSION]->(mv1:ModelVersion)
RETURN mv1.performance_metrics AS performance_metrics
```

[Fig 3.5] Predefined Cypher Query for the Model Performance Metrics Common Question

3.2.4 Uncommon Questions

For any relevant questions that aren't one of the defined common questions, we'll pass the question to LangChain which will attempt to generate a Cypher query from scratch. The prompt template (Section 8.6) we used for these questions includes the user question, database schema, a list of user questions, and their respective Cypher query that would fetch data to answer the question. These examples will guide the LLM in generating the Cypher query. If the query isn't valid, we'll attempt to correct the entity similar to the common question workflow and then pass the corrected question to LangChain again.

3.2.5 None Questions

There are questions that are neither common questions nor relevant questions to our data in our database, which we classify as other questions or "none". We provide a static message to the user that we don't support their question and to ask another question. An example of an irrelevant question would be what is the record for the fastest marathon completion time?

4. Prompt Engineering

4.1 What is Prompt Engineering?

In today's fast-evolving landscape of artificial intelligence, users often pose a wide variety of questions and requests to LLMs. The challenge lies in steering these models toward producing structured and predictable responses tailored to specific needs. This is where prompts play a pivotal role. Prompt engineering involves creating and refining templates to direct LLMs in performing designated tasks. Through iterative refinement, prompt engineering enhances accuracy and achieves finely tuned results, making it a crucial aspect of LLM deployment.

4.2 Prompt Templates

In our system, multiple LLM calls are orchestrated through distinct prompt templates, each serving a unique purpose. These templates fall into several categories.

4.2.1 Intent Matching Template

The first category is the INTENT_MATCHING_TEMPLATE (shortened version found in Section 8.4), which functions as a filter. This template assesses whether the user's input is relevant based on a predefined schema. If the input is deemed irrelevant, the model returns [NONE, -1]. If the input is relevant, the template then matches the user's request to one of the predefined "common questions" and returns the corresponding question number such as [COMMON, 3]. If the input doesn't match any of the common questions, the template returns [UNCOMMON, 0].

4.2.2 Input Parameter Extraction Template

Next, we have the INPUT_PARAMETER_EXTRACTION_TEMPLATE (shortened version found in Section 8.5). This template, provided with a Neo4j schema and a question, extracts a single parameter from the question and returns it within square brackets, such as ['Sales Confidence Interval', 'ReportField']. The goal here is to ensure that we extract the correct parameter after identifying the user’s query as a common question in the previous step. This precision ensures that the LLM maintains consistency and relevance throughout its responses.

4.2.3 User Response Template

Finally, the USER_RESPONSE_TEMPLATE is more straightforward (shortened version found in Section 8.7). With the query result, the original user question, and some example results in hand, the LLM is tasked with generating a human-readable final response. This response is then returned to the front end for the user to see. The template’s simplicity allows the LLM to focus on creating a clear, concise, and informative response, enhancing the user experience.

4.3 Prompt Engineering Efforts

One of the significant challenges with LLMs is hallucination, where the model generates plausible-sounding but incorrect outputs. A well-crafted prompt template can mitigate hallucinations and significantly improve accuracy. Through testing and iteration, we have made several major improvements to achieve satisfactory and stable results.

4.3.1 Structuring the Template

Since LLMs are designed to interpret text like a human, it's beneficial to structure the templates accordingly. Structured snippets of text are generally easier for both humans and LLMs to comprehend compared to long, uninterrupted paragraphs. For example, the intent-matching template is divided into sections for common, uncommon, and other questions. This helps the LLM differentiate between different types of user inquiries and label them appropriately. This structured approach provides clarity and allows the LLM to focus on specific tasks.

4.3.2 Using Positive Language

LLMs tend to respond better to positive language. Based on feedback from our mentors, we've shifted to using more positive commands in our templates by telling it to do specific tasks, and avoiding negative commands such as "Don't do this or that". This change has improved accuracy, as LLMs seem to be more aligned with positive guidance. By creating a positive and constructive framework, the LLM is better positioned to interpret and address user queries effectively.

4.3.3 Specifying Desired Output

Clearly specifying the desired output is an effective way to prevent AI hallucination. By indicating preferences for tone, style, and specific elements like headings or subheadings, we can guide the LLM to produce output that aligns with our intentions. This level of specificity helps the LLM stay on track and deliver responses that are consistent with our expectations, reducing the likelihood of erroneous or misleading information.

4.3.4 Defining a Persona or Frame of Reference

Giving the LLM a persona or frame of reference is particularly useful in business contexts where domain-specific knowledge is crucial. This approach helps guide the LLM to use appropriate tone and terminology for the scenario. In our case, we've defined the LLM as a Neo4j and English language expert, capable of understanding user intent and extracting entity parameters. To reinforce this, we've even set a strict guideline that the LLM should follow or face simulated consequences, such as not getting paid if guidelines are not adhered to. This approach creates a realistic and relatable context for the LLM, enhancing its performance.

4.4 Cypher query construction

4.4.1 Streamlining Common Queries with Cypher Query Templates

After the intent matching process, our approach is to construct a Cypher query tailored to each common question. This method yields a standardized template for handling these queries. Instead of initiating additional calls to an LLM, we simply substitute parameters within the Cypher query. This optimization significantly accelerates the processing of common questions.

4.4.2 Cypher Query

Cypher serves as a query language for retrieving data from a graph database. It bears a striking resemblance to SQL, boasting high intuitiveness and ease of comprehension.

4.4.3 Optimizing Upstream Data Access with Conditional Routing

When dealing with upstream questions, we initially categorize them into two types due to the availability of two routes for accessing the upstream data source: one route passes through the model, while the other links directly to columns. However, our discovery of the beneficial function "OPTIONAL MATCH" alters our approach. This function enables the query to return the successful route identified after an "OPTIONAL MATCH" operation. Consequently, we can consolidate two separate upstream questions into a single query.

5. User Interface & Graphical Elements

5.1 Streamlit Interface: Empowering Interactive Data Exploration

The Streamlit interface serves as a dynamic platform for interactive data exploration and visualization. With its intuitive design and user-friendly features, Streamlit empowers users to seamlessly develop and deploy data-driven applications. Through Streamlit, users can effortlessly create custom dashboards, interactive visualizations, and insightful data analytics tools, all with just a few lines of Python code.

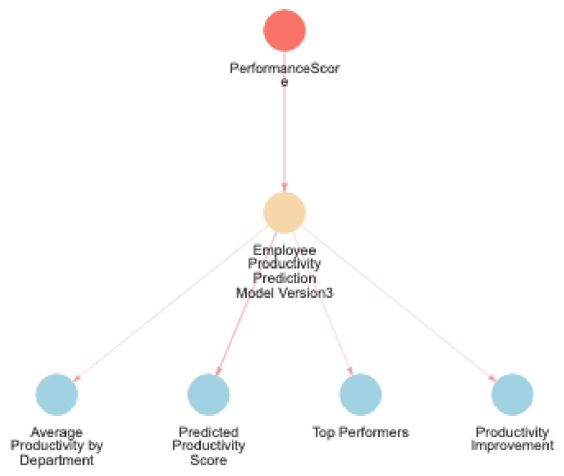
5.2 Enhancing User Understanding with Visualizations

We've decided to showcase visualizations for specific common questions, including inquiries into upstream data, affected downstream elements, and model version comparisons. The rationale behind this approach is that these types of questions can be readily understood by users through diagrams. For instance, consider the question: "What’s the upstream data source for the Sales Confidence Interval report field?" A diagram will illustrate that there's a model node upstream to the report field, indicating that the report field originates from the output of the model. When users examine further, they'll notice two columns at the top linked with the model, suggesting that these two columns serve as inputs to the model. Through this example, users gain a high-level overview of the relationship between each node before delving into the text response.

5.3 Leveraging Streamlit-Agraph Package for Network Visualization

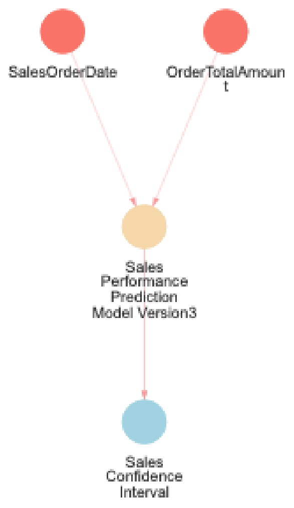
To present these diagrams, we've utilized the Streamlit-Agraph package. It offers a straightforward approach to building and creating network visualizations. The provided code showcases how users can input the nodes, edges, and configurations inside the brackets, then execute the "agraph" function. The screenshot below shows three types of common questions that the chatbot will present the diagrams.

What report fields are downstream of PerformanceScore column?


PerformanceScore
Employee Productivity Prediction Model Version3
Average Productivity by Department
Predicted Productivity Score
Top Performers
Productivity Improvement

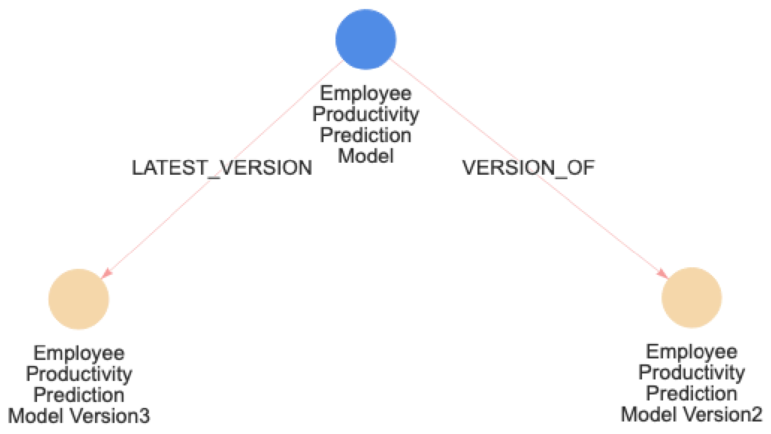
[Fig 5.1] downstream question diagram

What data is upstream to the Sales Confidence Interval report field?

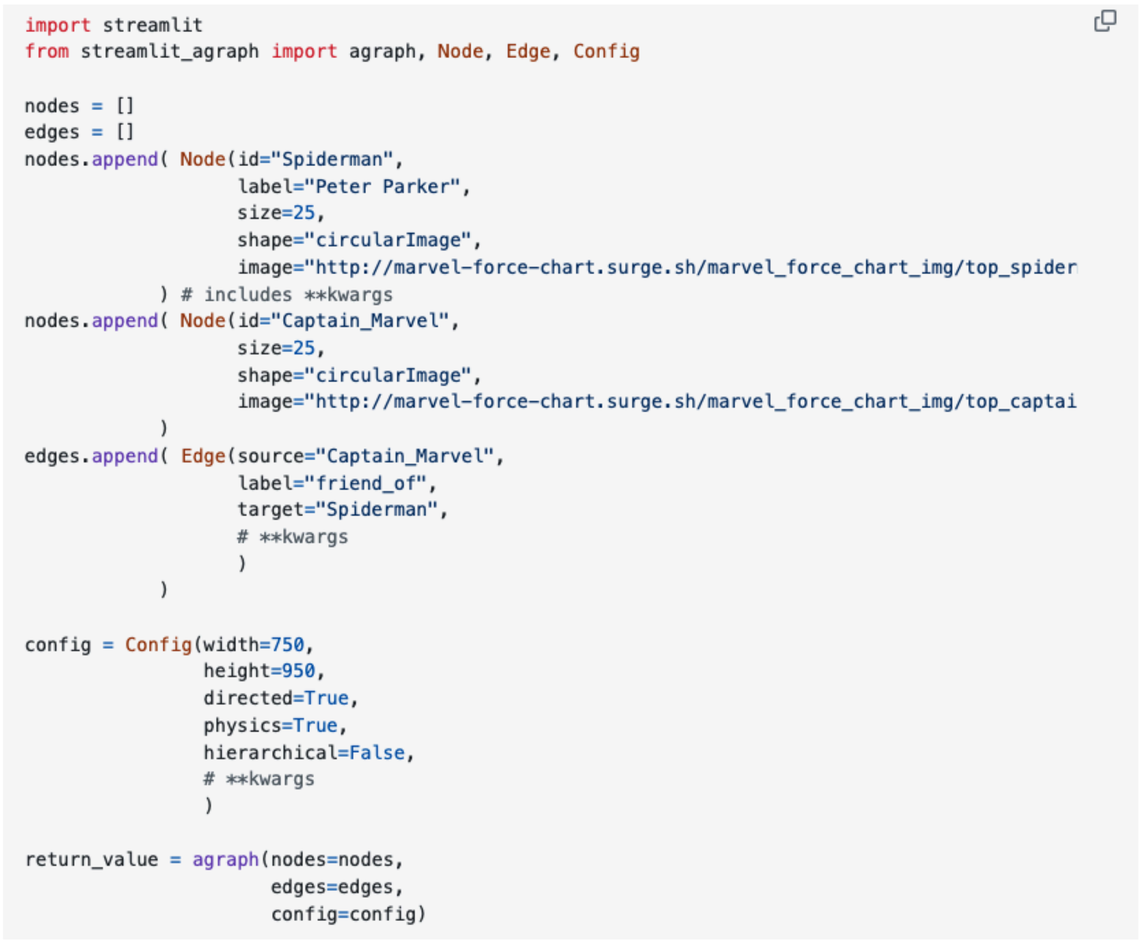

SalesOrderDate
OrderTotalAmount
Sales Performance Prediction Model Version3
Sales Confidence Interval

[Fig 5.2] upstream question diagram

What is the difference between the latest version and the previous version of the Employee Productivity Prediction Model?


Employee Productivity Prediction Model
LATEST_VERSION
VERSION_OF
Employee Productivity Prediction Model Version3
Employee Productivity Prediction Model Version2

[Fig 5.3] modelversion comparison diagram



```
import streamlit
from streamlit_agraph import agraph, Node, Edge, Config

nodes = []
edges = []
nodes.append( Node(id="Spiderman",
                   label="Peter Parker",
                   size=25,
                   shape="circularImage",
                   image="http://marvel-force-chart.surge.sh/marvel_force_chart_img/top_spider
            ) # includes **kwargs
nodes.append( Node(id="Captain_Marvel",
                   size=25,
                   shape="circularImage",
                   image="http://marvel-force-chart.surge.sh/marvel_force_chart_img/top_captai
            )
edges.append( Edge(source="Captain_Marvel",
                   label="friend_of",
                   target="Spiderman",
                   # **kwargs
                   )
            )

config = Config(width=750,
                height=950,
                directed=True,
                physics=True,
                hierarchical=False,
                # **kwargs
                )

return_value = agraph(nodes=nodes,
                      edges=edges,
                      config=config)
```

[Fig 5.4] agraph sample code

6. Testing and Issues

6.1 Test Questions

We initiated the question-generation process by defining specific categories that would allow us to thoroughly test various aspects of the system's capabilities. These categories, also known as the intent types, included "Common," "Uncommon," and "None," each tailored to assess different facets of the chatbot's performance. "Common" questions were directly tied to the primary operations and expected functionalities of the database, such as querying the impact of specific columns on report fields or understanding model outputs. These questions were designed to confirm that the core features of our system were functioning correctly.

For the "Uncommon" category, we created questions that, while relevant, addressed less frequent or more complex scenarios. These included queries about specific algorithmic details of models and intricate data dependencies, which helped us evaluate the system's depth of understanding and its ability to handle detailed informational requests.

The "None" category consisted of questions that were completely irrelevant to the project's context. This category was crucial for testing the system's robustness against irrelevant or out-of-scope queries, ensuring that it could effectively manage unexpected inputs without errors.

The generation of these questions was significantly aided by the use of ChatGPT 4, which helped create diverse and nuanced questions that simulated realistic user interactions. This approach not only populated our test suite with a wide range of queries but also enhanced our ability to rigorously evaluate the system under varied conditions, ensuring a comprehensive assessment of its capabilities.

All of the questions were compiled in one CSV file, along with each question's intent type, correct answer, common question ID, and parameter extracted. This approach for test question generation was instrumental in validating the effectiveness and reliability of our chatbot within the structured environment of a graph database, ensuring readiness for real-world application and user interactions. See Appendix 8.2 for some sample testing questions.

6.2 Test Methods and Results

We created three Python scripts that tested the three major parts of the chatbot: intent matching, parameter extraction, and the chatbot's final response. For the intent-matching script, we calculate the overall intent-matching accuracy and accuracy for each intent type by comparing the expected intent types versus what the chatbot returns. The actual accuracy scores met our expectations of accuracy where from highest to lowest would be irrelevant (none) questions, common questions, and then uncommon questions.

	Common	Uncommon	Other
Intent Matching	93%	78%	100%
Parameter Extraction	97%	-	-
Chatbot Response	99%	60%	-

[Fig 6.1] Testing Results

For common requests, we tested the parameter extraction functionality. Similar to intent matching, for each test question, we compared the expected parameter extracted to what the chatbot returns. Given how common questions are a small set of questions, we have a high accuracy for parameter extraction as well.

For the final output testing, we tested if the chatbot correctly fetched data from the database and returned a correct answer. The difficulty with validating this part is that the LLM could return the correct answer but could describe it differently every time a user asks a question. We first attempted to use the token set ratio metric from a Python library called fuzzywuzzy which calculates the similarity between two strings by first sorting the tokens in each string alphabetically and then comparing them. The token set ratio would be a number between 0 and 100. As a team, we experimented with setting a threshold for the token set ratio that we would deem as a "correct" answer such as 70. However, a major issue with this metric is say a user's question requires fetching numerical data and the number returned from the chatbot is slightly different but the answer "sounds" right with a high token set ratio. In other words, the metric could label this as a correct answer, but the overall answer is wrong because just one numeric value of the answer is wrong. Therefore, given the time constraints of the project, we decided to manually check the answers after running the test questions against the chatbot.

Ultimately, we ended up getting a significantly higher accuracy for the common questions compared to the uncommon questions. This is expected as we have provided the common questions template with a significant amount of guidance compared to the uncommon questions which we simply provided a list of example questions and their respective Cypher query.

6.3 Debugging and Unsolved Issues

When we manually tested the final chatbot output, we started with very poor accuracies for both common and uncommon questions. One of the issues we identified was that we were not providing enough context from the Cypher queries for common questions. One example is for the question: How many nodes upstream is the data source for the Training Hours report field? This question initially simply returned the number 2. However, providing the number 2 and the initial user question wouldn't be clear enough for the LLM to generate the final output.

Consequently, we updated the Cypher query to include additional information with the fetched data, so now the Cypher query returns “The number of nodes upstream to the data source for Monthly Sales Trend is 2” instead of simply the number 2.

Another issue we found was that for certain uncommon questions routed through LangChain, the LLM would return correct Cypher queries but the direction of the relationship might be incorrect. In some even worse cases, it sometimes would return Cypher queries that were completely wrong. To reiterate, for the uncommon question prompt template, we would provide the LLM with the database schema. So it was odd that even with the provided schema, it would incorrectly generate a Cypher query. Therefore, we experimented with adding more question examples in our uncommon question prompt template and saw a significant increase in accuracy for uncommon questions.

7. Conclusion & Next Steps

Our team has made substantial progress in developing a RAG chatbot that would provide more transparency to various companies' model metadata that they collect. The chatbot can quickly handle single-parameter requests with a significant degree of flexibility in parameter matching. This chatbot leverages the capabilities of LangChain, GPT-3.5 Turbo, and Neo4j to interpret user requests, match or generate Cypher queries, and generate informative, human-readable responses. We've overcome various challenges during the development of the chatbot such as defining separate workflows for each question type, experimenting with prompt templates for common and uncommon questions, and creating effective methods at testing our chatbot.

Several key initiatives stand out to further refine and enhance the capabilities of our chatbot. For Large Language Models, we plan to improve the intent matching's latency and save costs by implementing Aurelio Lab's semantic router, where we could classify question type by comparing it to an embedding for each question type. Additionally, experimenting with different open-source LLMs and their configurations across various stages of the chatbot workflow—such as intent matching, parameter extraction, Cypher query generation, and generating the final chatbot response—will allow us to fine-tune our system for optimal performance and accuracy.

From a user experience perspective, introducing follow-up questions when the orchestrator fails to fetch data will enhance the chatbot's interactivity and problem-solving capacity. Supporting multiple parameter requests in a single query will also significantly broaden the scope of user interactions, making the chatbot more versatile and user-friendly.

Other avenues for improvement include experimenting with a unified template for both intent matching and parameter extraction to streamline the development process. For testing improvements, research tools for automating the testing process especially for the final chatbot output. Additionally, hosting the Streamlit application will ensure that our chatbot is accessible and maintainable, providing a robust platform for user interactions.

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. This strategic enhancement and expansion will pave the way for more sophisticated, efficient, and user-centered chatbot systems that provide more transparency for companies' metadata and complex models in the near future.

KPMG COLUMBIA UNIVERSITY Data Science Institute

8. Appendix

8.1 List of Common Questions

- What report fields are downstream of a specific column?
- What are the performance metrics of a specific model?
- What data is upstream to a specific report field?
- How many nodes upstream is the datasource for a specific report field?
- How was this report field calculated?
- What is the difference between the latest version and the previous version of a specific model?
- What are the top features of a specific model?
- Tell me about the latest version of a specific model?

8.2 Database Schema Used in Prompt Templates

Node properties are the following:
Database {name: STRING, type: STRING},Table {name: STRING, primary_key: STRING},Column {name: STRING, type: STRING},BusinessGroup {name: STRING},Contact {name: STRING, email: STRING},User {name: STRING, account: STRING, entitlement: LIST, role: STRING},Report {name: STRING},ReportSection {name: STRING},ReportField {id: STRING, name: STRING},DataElement {name: STRING, source: STRING, generatedFrom: STRING},ModelVersion {name: STRING, version: INTEGER, latest_version: STRING, metadata: STRING, model_parameters: STRING, top_features: STRING, performance_metrics: STRING, model_id: STRING},Model {name: STRING, model_metadata: STRING, move_id: STRING}
Relationship properties are the following:

The relationships are the following:
(:Database)-[:CONTAINS]->(:Table),(:Database)-[:ASSOCIATED_WITH]->(:BusinessGroup),(:Table)-[:HAS_COLUMN]->(:Column),(:Table)-[:HAS_PRIMARY_KEY]->(:Column),(:Column)-[:TRANSFORMS]->(:DataElement),(:Contact)-[:CONTACT_OF]->(:BusinessGroup),(:User)-[:ENTITLED_ON]->(:Database),(:User)-[:ENTITLED_ON]->(:Report),(:User)-[:OWNS]->(:Report),(:User)-[:OWNS]->(:ModelVersion),(:User)-[:MAINTAINS]->(:Report),(:User)-[:MAINTAINS]->(:ModelVersion),(:Report)-[:ASSOCIATED_WITH]->(:BusinessGroup),(:ReportSection)-[:PART_OF]->(:Report),(:ReportField)-[:BELONGS_TO]->(:ReportSection),(:DataElement)-[:FEEDS]->(:ReportField),(:DataElement)-[:INPUT_TO]->(:ModelVersion),(:ModelVersion)-[:PRODUCES]->(:DataElement),(:Model)-[:VERSION_OF]->(:ModelVersion),(:Model)-[:LATEST_VERSION]->(:ModelVersion)

8.3 Test Question Snippet

	Questions	**Answers**
1	Which downstream fields are influenced by the AccountBalance column? (**COMMON**)	AccountBalance' influences the 'Cash Flow Trends' field in the Financial Health Dashboard.
2	What are the performance metrics of the latest version for the Employee Productivity Prediction Model? (**COMMON**)	Mean Absolute Error: 0.70, Mean Percentage Error: 0.55, Root Mean Squared Error: 0.60
3	What data contributes to the Monthly Sales Trend field in the Sales Performance Dashboard? (**COMMON**)	"SalesOrderDate" and "OrderTotalAmount" columns are upstream data for the "monthly_sales_trend" field in the Sales Performance Dashboard.
4	How many nodes upstream is the datasource for the Monthly Sales Trend field in the Sales Performance Dashboard? (**COMMON**)	2
5	How was the Monthly Sales Trend field in the Sales Performance Dashboard calculated? (**COMMON**)	The Monthly Sales Trend field is calculated by aggregating 'OrderTotalAmount' by month based on 'SalesOrderDate'.
6	What are the differences in algorithm and top_features between the latest vesion and previous version of Employee Productivity Prediction Model? (**COMMON**)	Employee Productivity Prediction Model Version3: Random Forest, PerformanceScore (0.40), PerformanceReviewDate (0.30), PerformanceComments (0.20), EmployeeID (0.10). Previous Version (Version2): Decision Tree, PerformanceScore (0.55), PerformanceReviewDate (0.25), EmployeeID (0.20).
7	What are the top features of the Financial Health	TransactionAmount,

	Prediction Model? (**COMMON**)	TransactionDate, TransactionType, AccountID
8	Tell me about the latest version of the Employee Productivity Prediction Model? (**COMMON**)	The latest version of the Employee Productivity Prediction Model incorporates a Random Forest algorithm, emphasizing features such as PerformanceScore (0.40), PerformanceReviewDate (0.30), PerformanceComments (0.20), and EmployeeID (0.10).
9	What machine learning technique is utilized in Employee Productivity Prediction Model Version1? (**UNCOMMON**)	Neural Network
10	What is the neuron configuration in the second layer of the Inventory Management Prediction Model Version2? (**UNCOMMON**)	64 neurons
11	What is the max depth setting in the Random Forest model of Customer Satisfaction Prediction Model Version3? (**UNCOMMON**)	10
12	Who is the owner of Customer Satisfaction Survey Analysis Report? (**UNCOMMON**)	Customer Service Director
13	What is the capital of France? (**NONE**)	Paris
14	Who won the Nobel Prize in Physics in 2020? (**NONE**)	Roger Penrose, Reinhard Genzel, and Andrea Ghez
15	What is the tallest mountain in the world? (**NONE**)	Mount Everest
16	What is the chemical formula for water? (**NONE**)	H2O

8.4 Intent Matching Prompt Template

Python

```
Task:
   Step 1: determine if the user input is relevant or not based on whether it
uses any words mentioned in the schema
   If it is NOT relevant, return [NONE,-1]

   If it is relevant, step 2, match user request intent to one of the following
"common questions" and return the question number.

   And if it doesn't match any of the following 5 questions, return
[UNCOMMON,0]

   Make sure ONLY return [COMMON,Integer], [UNCOMMON,0] or [NONE,-1]!!!!!

   Common Questions:
   - 1. What report fields are downstream of a specific column?
   - 2. What are the performance metrics of a specific model?
   - 3. What data is upstream to a specific report field?
   - 4. How many nodes upstream is the datasource for a specific report field?
   - 5. How was this report field calculated?
   - 6. What is the difference between the latest version and the previous
version of a specific model?
   - 7. What are the top features of a specific model?
   - 8. Tell me about the latest version of a specific model?

   Some examples of uncommon Questions:
   - 0. How many report fields are there?
   - 0. What is the database type of a specific database?
   - 0. What are the columns in a specific table of a database?
   - 0. Which model versions have an accuracy metric above 85%?
```

```
    Example:
    - Question: What is fastest animal in the world?
    - Answer: [NONE,-1]

    Example:
    - Question: What are the SARIMA model parameters in Inventory Management
Model Version 1?
    - Answer: [UNCOMMON,0]

    Example:
    - Question: Which business group is linked to the Employee Productivity
Report?
    - Answer: [UNCOMMON,0]

    Example:
    - Question: What are the performance metrics of Customer Satisfaction
Prediction Model?
    - Answer: [COMMON,2]

    Schema:
    {schema}

    User input is:
    {question}
```

8.5 Parameter Extraction Prompt Template

Python
```
Task: Given a Neo4j schema and a question, extract the single parameter from
the question and return it within square brackets []
   Only return the input parameter and its type within the square brackets

   Schema:
   {schema}

   User input is:
   {question}

    Example:
   - Question: What report fields are downstream of the FeedbackComments
column?
   - Return [FeedbackComments,Column]

   Example:
   - Question: What are the performance metrics of Customer Satisfaction
Prediction Model?
   - Return [Customer Satisfaction Prediction Model,Model]

   Example:
   - Question: What data is upstream to the Sales Confidence Interval report
field?
   - Return [Sales Confidence Interval,ReportField]

   Example:
   - Question: How many nodes upstream is the datasource for Training Hours
report field?
   - Return [Training Hours,ReportField]
```

```
    Example:
    - Question: How was the Sales Confidence Interval report field calculated?
    - Return [Sales Confidence Interval,ReportField]

    Example:
    - Question: What is the difference between the latest version and the
previous version of the Employee Productivity Prediction Model?
    - Return [Employee Productivity Prediction Model,Model]
```

8.6 Uncommon Question Prompt Template

```
Unset
Given these examples of questions and their associated Cypher query, and
schema, generate the Cypher query for the user input.
    Only the Cypher query should be returned.

    Question: What report fields are downstream of the CustomerID column?
    Cypher Query:
    MATCH (col:Column)
    WHERE col.name CONTAINS "CustomerID"
    OPTIONAL MATCH (col)-[r1]->(de1:DataElement)-[r2]->(rf1:ReportField)
    WITH col, collect(distinct rf1) AS rf1s
    OPTIONAL MATCH
(col)-[r3]->(de2_1:DataElement)-[r4]->(mv:ModelVersion)-[r5]->(de2_2:DataElemen
t)-[r6]->(rf2:ReportField)
    WITH col, rf1s, collect(distinct rf2) AS rf2s
    WITH col, rf1s + rf2s AS allRfs
    UNWIND allRfs AS rf
    WITH col, rf
    RETURN col.name AS column, collect(distinct rf.name) AS AffectedReportFields
```

```
   Question: What are the performance metrics of Employee Productivity
Prediction Model?
   Cypher Query:
   MATCH (m:Model)
   WHERE m.name CONTAINS "Employee Productivity Prediction Model"
   MATCH (m)-[r1:LATEST_VERSION]->(mv1:ModelVersion)
   RETURN mv1.performance_metrics AS performance_metrics

   Question: What is the difference between the latest version and the previous
version of the Employee Productivity Prediction Model?
   Cypher Query:
   MATCH (m:Model)
   WHERE m.name CONTAINS "Employee Productivity Prediction Model"
   MATCH (m)-[r1:LATEST_VERSION]->(mv1:ModelVersion)
   MATCH (m)-[r2]->(mv2:ModelVersion)
   WHERE mv2.version = mv1.version-1
   RETURN
   mv1.name AS LatestVersion_v1,
   mv2.name AS PreviousVersion_v2,
   {{
   model_parameters_v1: mv1.model_parameters,
   model_parameters_v2: mv2.model_parameters
   }} AS ModelParameters,
   {{
   top_features_v1: mv1.top_features,
   top_features_v2: mv2.top_features
   }} AS TopFeatures

   Question: How was the Sales Confidence Interval report field calculated?
   Cypher Query:
```

```
   MATCH (rf:ReportField {{name: "Sales Confidence
Interval"}})<-[:FEEDS]-(de:DataElement)
   RETURN de.generatedFrom AS GeneratedFrom

   Question: What forecasting method is implemented in Inventory Management
Prediction Model Version1 model version?
   Cypher Query:
   MATCH (mv:ModelVersion {{name: "Inventory Management Prediction Model
Version1"}})
   RETURN mv.model_parameters

   User input:
   {question}

   Schema:
   {schema}
```

8.7 Chatbot Final Response Prompt Template

Unset

```
Given this user question: {query}
   And data from the Neo4j database: {cypher_query_response}

   Task: Answer the user question using only the data from the Neo4j database.
Use nested bullet points to summarize the answer if longer than one sentence.

   Example short answer response: The datasource for the Monthly Sales Trend
field is 2 nodes upstream.

   Example of long answer response:
```

```
   The main differences between the latest version (Employee Productivity Model
Version3) and the previous version (Employee Productivity Model Version2) of
the Employee Productivity Prediction Model are as follows:

   1. Model Parameters:
       - Version3: Decision Tree algorithm with a maximum depth of 8 and a
minimum samples split of 4.
       - Version2: Random Forest algorithm with 100 trees, a maximum depth of
10, and a minimum samples split of 2.

   2. Top Features:
       - Version3: The top features considered in Version3 are PerformanceScore
(0.55), PerformanceReviewDate (0.25), and EmployeeID (0.2).
       - Version2: The top features considered in Version2 are PerformanceScore
(0.4), PerformanceReviewDate (0.3), PerformanceComments (0.2), and EmployeeID
(0.1).

   Overall, the key differences between the two versions lie in the choice of
algorithm used, the parameters of the algorithm, and the weightage assigned to
the top features in the model.
```