Health Hive - Hospital System Chatbot

Authors

1. Adhyantini Bogawat - NUID: 002766612 | LinkedIn
2. Chinmay Dharwad - NUID: 002771037 | LinkedIn
3. Kishor Channal - NUID: 002737089 | LinkedIn

Table of Contents

1. Introduction

The **Health Hive - Hospital System Chatbot** is developed as a final project for the **Prompt Engineering INFO7375** course at **Northeastern University** under the guidance of **Professor Nick Brown**. This project leverages advanced techniques in LangChain to build a Retrieval-Augmented Generation (RAG) model designed specifically for hospital systems, capable of interacting with both structured and unstructured data.

2. Project Overview

This project aims to build a custom chatbot using LangChain, Neo4j, and other technologies to interact with hospital system data. The key objectives are:

- **Build and deploy a chatbot** using LangChain for natural language processing.
- **Integrate Neo4j AuraDB** for graph database functionality.
- **Develop a RAG model** that retrieves both structured and unstructured data from Neo4j.
- **Deploy the chatbot** using FastAPI for backend services and Streamlit for the frontend interface.

Directory Structure

- **langchain_intro/**: A preliminary version of the chatbot to get familiar with LangChain.
- **data/**: Contains raw hospital system data stored as CSV files.
- **hospital_neo4j_etl/**: Script to load data into the Neo4j database.
- **chatbot_api/**: Core FastAPI application serving the chatbot as a REST endpoint.
- **tests/**: Scripts to test the chatbot's response speed.

- **chatbot_frontend/**: Streamlit application that serves as the UI for interacting with the chatbot.

3. Prerequisites

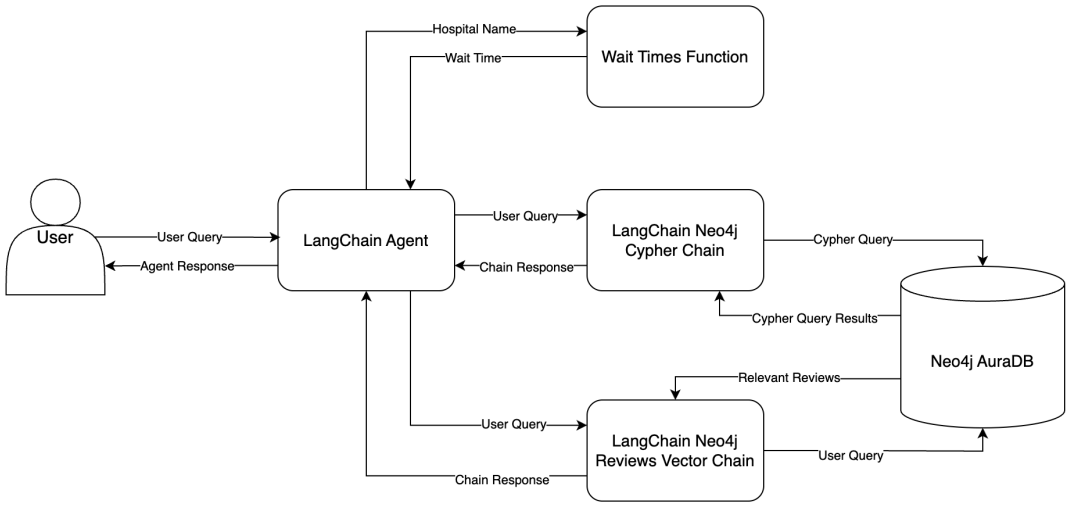
This project is intended for intermediate Python developers with the following skills and knowledge:

- Intermediate Python programming.
- Understanding of Large Language Models (LLMs) and prompt engineering.
- Familiarity with text embeddings, vector databases, and graph databases (Neo4j).
- Experience with the OpenAI developer ecosystem.
- Knowledge of REST APIs and FastAPI.
- Asynchronous programming concepts.
- Docker and Docker Compose for containerization.

4. System Architecture

Components

- **LangChain**: The core library used for building the chatbot.
- **Neo4j AuraDB**: A cloud-based graph database for storing and querying hospital data.
- **FastAPI**: The framework used to develop the RESTful API serving the chatbot.
- **Streamlit**: The framework used for creating the chatbot's frontend interface.
- **Docker**: For containerizing the application and its services.



Data Flow

1. **User Interaction**: Users interact with the chatbot via the Streamlit frontend.
2. **Chatbot Processing**: The frontend sends the user's query to the FastAPI backend.
3. **Data Retrieval**: The FastAPI backend uses LangChain to retrieve relevant data from Neo4j and generate a response.
4. **Response Delivery**: The chatbot returns the generated response to the frontend, which is then displayed to the user.

5. Implementation Steps

Step 1: Get Familiar With LangChain

Before building the full chatbot, it's essential to understand LangChain's main components and features. This step involves setting up a preliminary version of the chatbot to get familiar with the LangChain framework.

5.1.1 Setup

1. **Create a Python Project**: Set up a new Python project with a virtual environment.
2. **Install Dependencies**: Install necessary libraries, including LangChain, OpenAI, and python-dotenv for environment management.

```
(venv) $ python -m pip install langchain openai python-dotenv
```

5.1.2 Initial Configuration

- **Environment Variables**: Store your OpenAI API key in a `.env` file.
- **Create Directories and Files**:
    - **data/**: Store raw data files.
    - **langchain_intro/**: Contains initial scripts to build a preliminary chatbot.

5.1.3 Chat Models and Prompt Templates

- **Chat Models**: Learn to instantiate and interact with chat models using LangChain.
- **Prompt Templates**: Design modular prompts that guide the model's responses.

5.1.4 Chains and Retrieval Objects

- **Chains**: Create a sequence of calls between LangChain objects, known as chains.
- **Retrieval Objects**: Set up a ChromaDB instance to store and retrieve embeddings for patient reviews.

Step 2: Define Business Requirements and Analyze Data

Problem and Requirements

Objective: Create a chatbot for a large hospital system to answer queries about patients, visits, physicians, and insurance without needing SQL knowledge or waiting for reports.

Key Requirements:

- Handle both quantitative queries (e.g., total billing, patient counts) and qualitative ones (e.g., patient feedback).
- **Dynamic Query Generation**: The chatbot should generate accurate queries on the fly, addressing new or nuanced questions from stakeholders.

Data Overview

Datasets:

- **hospitals.csv**: Details 30 hospitals with hospital_id, hospital_name, and hospital_state.
- **physicians.csv**: Contains 500 physician records with physician_id, physician_name, medical_school, etc.

- **payers.csv**: Lists insurance companies with fields like payer_id and payer_name.
- **reviews.csv**: Includes 1005 patient reviews, used to answer subjective questions about experiences.
- **visits.csv**: Documents 9998 hospital visits, connecting hospitals, patients, physicians, and payers.

Data Gaps and API Needs

- **Wait Times**: The system lacks historical wait time data. The chatbot will need to call an API to retrieve real-time wait times.

Chatbot Design

Architecture:

- **LangChain Agent**: Manages queries, deciding which tool or data source to use.
- **Neo4j AuraDB**: Stores structured and unstructured hospital system data.
- **LangChain Cypher Chain**: Converts and executes queries in Neo4j.
- **LangChain Reviews Chain**: Retrieves relevant patient reviews based on query context.
- **Wait Times Function**: Fetches current wait times via an external API.

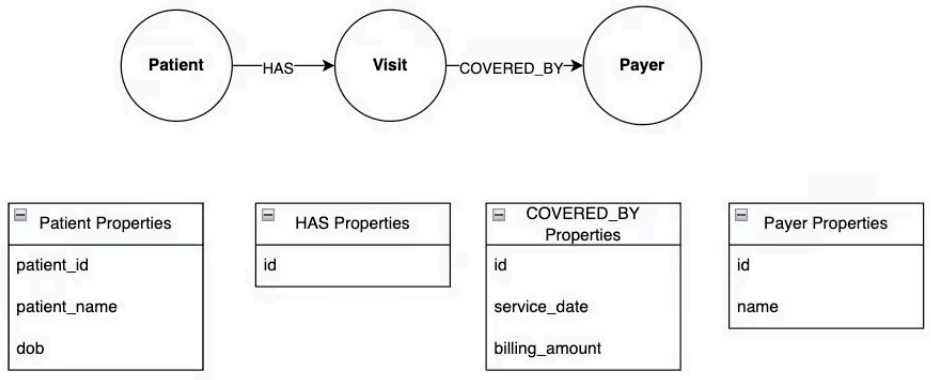
Step 3: Set Up a Neo4j Graph Database

5.3.1 Overview of Graph Databases

Graph Databases: Unlike relational databases, graph databases like Neo4j store data as nodes, relationships, and properties, making them ideal for modeling complex relationships.

Key Advantages:

- **Simplicity**: Easier to model real-world relationships.
- **Efficiency**: Fast retrieval of connected data.
- **Flexibility**: Schema-less, allowing easy adaptation to evolving data structures.
- **Pattern Matching**: Supports complex queries without complicated joins.



5.3.2 Create a Neo4j AuraDB Instance

Step-by-Step:

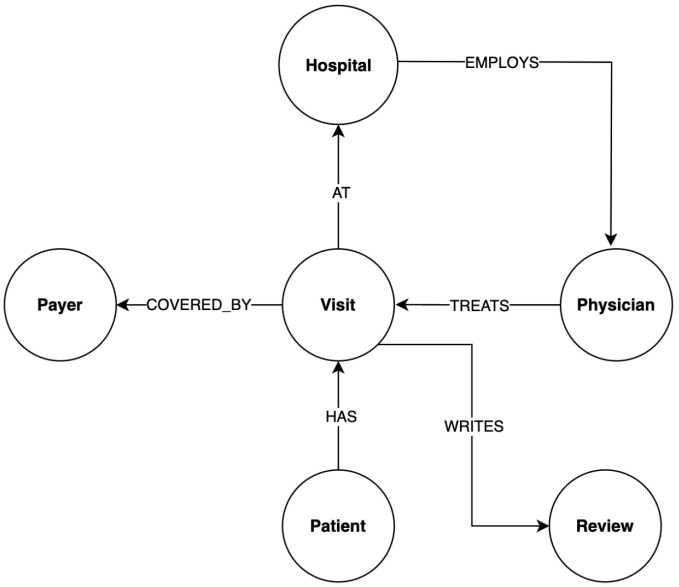
1. **Sign Up**: Create a free account on Neo4j AuraDB.
2. **Create Instance**: Set up a new Neo4j instance and download the credentials.

3. **Environment Setup**: Add `NEO4J_URI`, `NEO4J_USERNAME`, and `NEO4J_PASSWORD` to your `.env` file for secure access.

5.3.3 Design the Hospital System Graph Database

Graph Design:

- **Nodes**: Represent entities like Hospital, Patient, Physician, Payer, Visit, and Review.
- **Relationships**: Define connections like HAS, TREATS, COVERED_BY, linking nodes together.
- **Properties**: Store metadata, e.g., id, name, date_of_birth, billing_amount.



5.3.4 Upload Data to Neo4j

ETL Process:

1. **Docker Setup**: Use Docker to automate the extract, transform, load (ETL) process.
2. **Script Execution**: Create and run a Python script to load CSV data into Neo4j following your graph design.
3. **Docker Compose**: Use `docker-compose.yml` to orchestrate the ETL process.

5.3.5 Query the Hospital System Graph

Cypher Basics:

- **MATCH**: Used to find nodes and relationships.
- **WHERE**: Filters results based on node and relationship properties.
- **RETURN**: Specifies what data to retrieve.

Advanced Queries:

- **Relationships**: Query patterns involving multiple nodes and relationships.
- **Aggregations**: Perform operations like COUNT and SUM for complex analytics.

Step 4: Build the Graph RAG Chatbot with LangChain

Overview

With your design and data preparation complete, you are ready to build the chatbot using LangChain and Neo4j. The process is streamlined thanks to the setup you've done, and now it's about implementing the necessary components.

Project Structure

Create the following project structure:

```
./
├── chatbot_api/
│   ├── src/
│   │   ├── agents/
│   │   │   └── hospital_rag_agent.py
│   │   ├── chains/
│   │   │   ├── hospital_cypher_chain.py
│   │   │   └── hospital_review_chain.py
│   │   ├── tools/
│   │   │   └── wait_times.py
│   └── pyproject.toml
└── .env
```

Ensure your `.env` file contains necessary environment variables such as OpenAI API keys, Neo4j credentials, and paths to the datasets.

Install Dependencies

Add the required dependencies in `chatbot_api/pyproject.toml`:

```
[project]
dependencies = [
    "langchain==0.1.0",
    "openai==1.7.2",
    "neo4j==5.14.1",
]
```

Install the dependencies:

```
(venv) $ python -m pip install .
```

Implement the Chains

You will create two main chains for the chatbot: one for handling patient reviews using a vector store, and another for generating Cypher queries to interact with the hospital's structured data in Neo4j.

Reviews Vector Chain This chain will handle natural language queries related to patient reviews by using embeddings stored in Neo4j.

```python
# chatbot_api/src/chains/hospital_review_chain.py
import os
from langchain.vectorstores.neo4j_vector import Neo4jVector
from langchain.chains import RetrievalQA
from langchain_openai import ChatOpenAI

HOSPITAL_QA_MODEL = os.getenv("HOSPITAL_QA_MODEL")

neo4j_vector_index = Neo4jVector.from_existing_graph(
    embedding=OpenAIEmbeddings(),
    url=os.getenv("NEO4J_URI"),
    username=os.getenv("NEO4J_USERNAME"),
    password=os.getenv("NEO4J_PASSWORD"),
    index_name="reviews",
    node_label="Review",
    text_node_properties=["physician_name", "patient_name", "text", "hospital_name"],
)

reviews_vector_chain = RetrievalQA.from_chain_type(
    llm=ChatOpenAI(model=HOSPITAL_QA_MODEL, temperature=0),
    retriever=neo4j_vector_index.as_retriever(k=12),
)
```

Cypher Generation Chain This chain will convert natural language queries into Cypher queries for retrieving structured data from Neo4j.

```python
# chatbot_api/src/chains/hospital_cypher_chain.py
import os
from langchain_community.graphs import Neo4jGraph
from langchain.chains import GraphCypherQAChain
from langchain_openai import ChatOpenAI

HOSPITAL_CYPHER_MODEL = os.getenv("HOSPITAL_CYPHER_MODEL")

graph = Neo4jGraph(
    url=os.getenv("NEO4J_URI"),
    username=os.getenv("NEO4J_USERNAME"),
    password=os.getenv("NEO4J_PASSWORD"),
)

hospital_cypher_chain = GraphCypherQAChain.from_llm(
    cypher_llm=ChatOpenAI(model=HOSPITAL_CYPHER_MODEL, temperature=0),
    graph=graph,
    validate_cypher=True,
)
```

Create Wait Time Functions These functions simulate real-time wait times at hospitals.

```
# chatbot_api/src/tools/wait_times.py
import numpy as np

def get_current_wait_times(hospital):
    """Simulate fetching current wait times at a hospital."""
    wait time_in_minutes = np.random.randint(0, 600)
    hours, minutes = divmod(wait_time_in_minutes, 60)
    return f"{hours} hours {minutes} minutes" if hours > 0 else f"{minutes} minutes"

def get_most_available_hospital(_):
    """Simulate finding the hospital with the shortest wait time."""
    return {"example_hospital": 15}
```

Assemble the Chatbot Agent Finally, combine everything into a chatbot agent.

```
# chatbot_api/src/agents/hospital_rag_agent.py
import os
from langchain_openai import ChatOpenAI
from langchain.agents import create_openai_functions_agent, Tool

HOSPITAL_AGENT_MODEL = os.getenv("HOSPITAL_AGENT_MODEL")

tools = [
       Tool(name="Experiences", func=reviews_vector_chain.invoke, description="Answer
questions about patient experiences."),
          Tool(name="Graph", func=hospital_cypher_chain.invoke, description="Answer
structured data questions."),
      Tool(name="Waits", func=get_current_wait_times, description="Fetch current wait
times at a specific hospital."),
     Tool(name="Availability", func=get_most_available_hospital, description="Find the
hospital with the shortest wait time."),
]

chat_model = ChatOpenAI(model=HOSPITAL_AGENT_MODEL, temperature=0)
hospital_rag_agent = create_openai_functions_agent(llm=chat_model, tools=tools)
```

Step 5: Deploy the LangChain Agent

Overview

With the LangChain agent for your hospital system chatbot fully developed, the next step is deployment. You'll deploy the agent as a FastAPI service and create a Streamlit-based user interface to allow stakeholders to interact with the chatbot.

Serve the Agent with FastAPI

To expose the chatbot as a service, you'll create a FastAPI application.

Define Request and Response Models: Use Pydantic models to define the expected input and output for your API.

```
# chatbot_api/src/models/hospital_rag_query.py
from pydantic import BaseModel

class HospitalQueryInput(BaseModel):
    text: str

class HospitalQueryOutput(BaseModel):
    input: str
    output: str
    intermediate_steps: list[str]
```

Asynchronous API Implementation: Utilize FastAPI's asynchronous capabilities for efficient API request handling.

```
# chatbot_api/src/main.py
from fastapi import FastAPI
from agents.hospital_rag_agent import hospital_rag_agent_executor
from models.hospital_rag_query import HospitalQueryInput, HospitalQueryOutput

app = FastAPI(title="Hospital Chatbot")

@app.post("/hospital-rag-agent")
async def query_hospital_agent(query: HospitalQueryInput) -> HospitalQueryOutput:
    query_response = await hospital_rag_agent_executor.ainvoke({"input": query.text})
    query_response["intermediate_steps"] = [str(s) for s in
query_response["intermediate_steps"]]
    return query_response
```

Dockerize the API: Build the FastAPI service using Docker. Below is a simplified Dockerfile.

```
# chatbot_api/Dockerfile
FROM python:3.11-slim
WORKDIR /app
COPY ./src/ /app
COPY ./pyproject.toml /code/pyproject.toml
RUN pip install /code/.
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Orchestrate with Docker Compose: Add the FastAPI service to your `docker-compose.yml` file.

```
version: '3'
services:
  hospital_neo4j_etl:
    build: ./hospital_neo4j_etl
    env_file: .env

  chatbot_api:
    build: ./chatbot_api
    env_file: .env
    depends_on:
      - hospital_neo4j_etl
    ports:
      - "8000:8000"
```

Run the service:

```
$ docker-compose up --build
```

Create a User Interface with Streamlit

To make the chatbot accessible to stakeholders, build a Streamlit application that communicates with the FastAPI service.

Streamlit Application: Create a simple chat interface.

```
# chatbot_frontend/src/main.py
import os
import requests
import streamlit as st

CHATBOT_URL = os.getenv("CHATBOT_URL", "http://localhost:8000/hospital-rag-agent")

st.title("Hospital System Chatbot")

if prompt := st.text_input("What do you want to know?"):
    response = requests.post(CHATBOT_URL, json={"text": prompt})
    if response.status_code == 200:
        st.write(response.json()["output"])
    else:
        st.write("An error occurred. Please try again.")
```

Dockerize the Streamlit UI: Build the Streamlit UI using Docker.

```
# chatbot_frontend/Dockerfile
```

```
FROM python:3.11-slim
WORKDIR /app
COPY ./src/ /app
COPY ./pyproject.toml /code/pyproject.toml
RUN pip install /code/.
CMD ["streamlit", "run", "main.py"]
```

Update Docker Compose: Add the Streamlit UI to your `docker-compose.yml`.

```
services:
  chatbot_frontend:
    build: ./chatbot_frontend
    env_file: .env
    depends_on:
      - chatbot_api
    ports:
      - "8501:8501"
```

Start the entire setup:

```
$ docker-compose up --build
```

Conclusion

In this project, we built a powerful and flexible chatbot using LangChain, tailored specifically to the needs of a hypothetical hospital system. By aligning with business requirements and leveraging available data, we developed a sophisticated solution capable of handling both structured and unstructured data.

Key takeaways from this project include:

- **Utilizing LangChain**: We harnessed LangChain's capabilities to create a chatbot that interacts seamlessly with various data sources, offering a personalized and responsive user experience.
- **Hospital System Chatbot**: By focusing on the specific use case of a hospital system, we gathered and aligned business requirements with technical solutions, ensuring the chatbot meets the needs of its stakeholders.
- **Graph Databases Integration**: We integrated graph databases into the chatbot design, allowing for more complex queries and relationships to be efficiently managed, resulting in more accurate and relevant responses.
- **Neo4j AuraDB Setup**: We set up a Neo4j AuraDB instance, which was crucial in storing and querying data to meet the hospital system's needs.
- **RAG Chatbot Development**: We developed a Retrieval-Augmented Generation (RAG) chatbot capable of fetching both structured and unstructured data from Neo4j, showcasing how LangChain can enhance the capabilities of traditional chatbots.
- **Deployment with FastAPI and Streamlit**: Finally, we deployed the chatbot using FastAPI and Streamlit, making it accessible to stakeholders through a user-friendly interface. This deployment strategy ensures that the chatbot is not only functional but also scalable and easy to use.