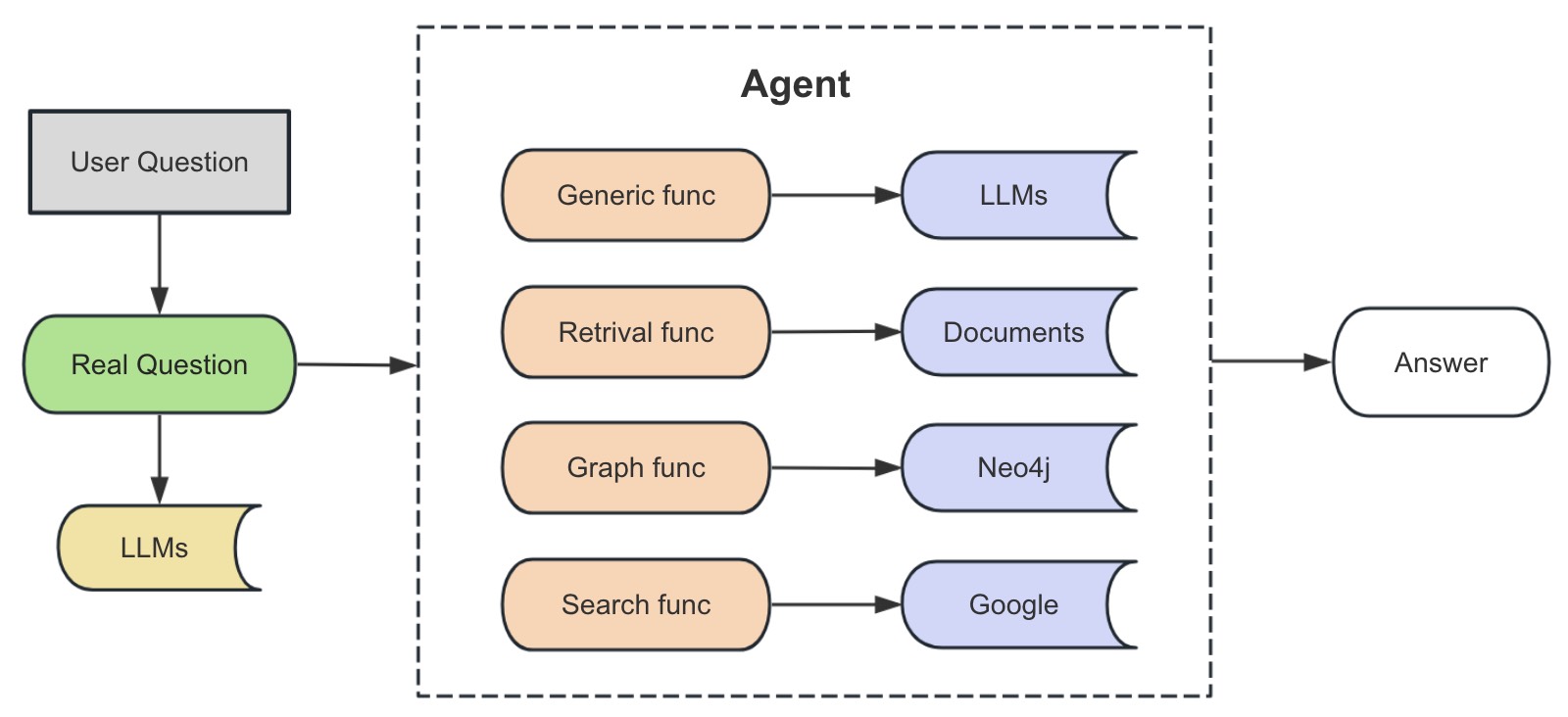
**Final report – Doctor Chatbot**

**Overview:**

The Doctor Chatbot is a chatbot that answers medical or disease related questions, such as ‘What causes a cold?’ “What medicines should I take then?” or generic conversations. Here's an overview of how such a chatbot could work:

* User Input: The chatbot receives user messages or questions related to diseases, symptoms, medications, and other health-related topics.
* Information Retrieval: The chatbot can have multiple sources for retrieving information. These sources can include local files such as PDFs, TXT, or CSV files containing medical literature or research. It can also utilize a knowledge graph like Neo4j, which can store relationships between various medical concepts. Additionally, the chatbot can perform Google searches to gather information from reputable medical websites.
* Disease-related Questions: When the user asks disease-related questions, such as "What causes a cold?" or "What are the symptoms of diabetes?" the chatbot can retrieve the appropriate information from its sources and present a summary of the relevant details.
* Medication Information: If the user asks about medications, such as "What medicines should I take for a cold?" or "Are there any side effects of a particular medication?" the chatbot can retrieve information about medications from its sources. It can provide details about the recommended medications, dosages, potential side effects, and any other relevant information.
* Generic Conversations: In addition to disease-related queries, the chatbot can engage in generic conversations. It can respond to greetings, provide general health tips, answer common medical questions, and even engage in small talk to provide a more conversational experience.
* Summarization: Once the relevant information is retrieved, the chatbot can employ natural language processing techniques to summarize the information in a concise and understandable manner. This helps in providing a clear response to the user's query.

**Structure:**  
The project's overall architecture follows the Agent model of LangChain and utilizes four different data sources for addressing questions related to specific scenarios. These include problem analysis, utilizing historical conversations to identify the user's true question while filtering out irrelevant information and facilitating named entity recognition.



* The Agent Generic Tool leverages the capabilities of a large model to answer general everyday communication questions.
* The Agent Retrieval Tool retrieves relevant fragments from documents to answer company-related questions, with data sourced from Xunyiwenyao.com and future plans to organize related documents.
* The Agent Graph Tool retrieves data from a knowledge graph to answer disease-related questions, reusing data and CQL templates from a knowledge graph question-answering project.
* The Agent Search Tool serves as a fallback solution, retrieving information from Google when other tools are unable to provide answers.

**Agent.py**

The provided .py file includes imports for various modules and libraries, such as `utils`, `config`, `prompt`, `os`, `langchain`, `py2neo`, `neo4j`, and `translate`. It defines a class named `Agent` with several methods. Here is a summary of the functionalities:

1. The `Agent` class initializes and utilizes different tools for answering user queries related to general questions, domain knowledge, medical-related questions, and using search engines.

2. The class uses the `Chroma` vector store for similarity search and the `FAISS` vector store for executing CQL queries.

3. It includes a translator to translate user input into Chinese (Since the knowledge graph is based on Chinese).

4. The class uses a language model chain (`LLMChain`) for generating responses based on predefined prompts and templates.

5. It interacts with a Neo4j graph database to retrieve information based on user queries.

6. The class implements memory functionality to store conversation history.

**App.py**

It sets up a Gradio-based user interface for the "Doctor Bot." Here's a breakdown of what the code does:

1. It imports the `gradio` library for creating the user interface and the `Service` class from the `service` module.

2. The `doctor\_bot` function is defined as a callback function for the Gradio interface. It takes a message and history as input, creates an instance of the `Service` class, calls the `answer` method of the service with the message and history, and returns the output from the service.

3. The `css` variable stores some custom CSS styles for the Gradio interface.

4. The `demo` variable defines the Gradio ChatInterface. It uses the `doctor\_bot` function as the callback function for generating responses. The interface includes a chatbot component, a textbox for user input, example queries, and buttons for submitting, clearing, retrying, and undoing inputs.

5. The `demo.launch()` function is called to start the Gradio interface.

When you run this script, it will launch a web-based interface where users can interact with the "Doctor Bot." Users can enter questions in the text box, submit them, and receive responses from the bot. The interface also provides example queries and buttons for clearing the conversation history.

**Config.py**

It defines a Python dictionary named `GRAPH\_TEMPLATE`, which contains templates for different types of queries related to a medical graph database. Each template is associated with a specific type of information and includes slots for dynamic data insertion.

Here's a detailed explanation of the template structure and the available query types:

1. `'desc'`: Retrieves the description of a disease.

2. `'cause'`: Retrieves the cause of a disease.

3. `'disease\_symptom'`: Retrieves the symptoms of a disease.

4. `'symptom'`: Retrieves the diseases associated with a symptom.

5. `'cure\_way'`: Retrieves the treatment methods, medications, and recommended foods for a disease.

6. `'cure\_department'`: Retrieves the medical departments related to the treatment of a disease.

7. `'prevent'`: Retrieves the prevention methods for a disease.

8. `'not\_eat'`: Retrieves the foods that should be avoided for a disease.

9. `'check'`: Retrieves the recommended medical examinations for a disease.

10. `'cured\_prob'`: Retrieves the cure probability of a disease.

11. `'acompany'`: Retrieves the complications associated with a disease.

12. `'indications'`: Retrieves the diseases that can be treated by a specific drug.

Each template includes the following components:

- `'slots'`: A list of slot names indicating the dynamic data that needs to be inserted into the template.

- `'question'`: The question template that will be presented to the user.

- `'cypher'`: The Cypher query template used to retrieve data from the graph database.

- `'answer'`: The template for the answer that will be presented to the user, including placeholders for the dynamic data.

The `%disease%` and `%symptom%` placeholders in the templates will be replaced with specific disease and symptom names, respectively, when generating the queries.

These templates provide a structured approach to query the medical graph database and generate responses based on user input.

**Data\_process.py**

It is a Python script that performs the following tasks:

1. Imports the necessary modules and functions from external libraries.

2. Defines a function named `doc2vec()` for converting documents to vectors.

3. Sets up a text splitter for breaking down text into smaller chunks.

4. Specifies a directory path where the documents are located.

5. Iterates over the files in the specified directory and loads each file using the appropriate loader based on its file extension (CSV, PDF, or TXT).

6. Uses the text splitter to split the loaded documents into smaller chunks.

7. Prints the resulting documents.

8. If there are documents available, creates a Chroma vector store from the documents using an embeddings model.

9. Persists the vector store in a specified directory.

The `doc2vec()` function converts the input documents into vector representations using the Chroma vector store and saves the vector store to disk for future use.

The script then calls the `doc2vec()` function when executed as the main entry point of the program.

**Prompt.py**

The provided code defines multiple string templates that are used in different parts of a conversational AI system. Here's a breakdown of each template:

1. `GENERIC\_PROMPT\_TPL`: A template used to generate a prompt for the conversation's generic context. It includes a placeholder for the user's query.

2. `RETRIVAL\_PROMPT\_TPL`: A template used to generate a prompt for retrieving information from a search result. It includes placeholders for the query, the search result, and the user's input.

3. `NER\_PROMPT\_TPL`: A template used to generate a prompt for named entity recognition (NER) tasks. It includes placeholders for format instructions and the user's query.

4. `GRAPH\_PROMPT\_TPL`: A template used to generate a prompt for querying a graph database. It includes placeholders for the query result and the user's input.

5. `SEARCH\_PROMPT\_TPL`: A template used to generate a prompt for answering user questions based on search results. It includes placeholders for the query result and the user's input.

6. `SUMMARY\_PROMPT\_TPL`: A template used to generate a prompt for summarizing a conversation. It includes placeholders for chat history and the user's query.

These templates provide a structured format for generating prompts in various conversational AI tasks, such as answering user questions, summarizing conversations, performing NER, retrieving information from search results, and querying a graph database. The specific use of these templates will depend on the implementation of the conversational AI system.

**Service.py**

It defines a `Service` class that acts as an interface for interacting with a conversational AI system. Here's an overview of the class and its methods:

1. `\_\_init\_\_(self)`: The constructor method initializes an instance of the `Service` class and creates an instance of the `Agent` class, which represents the conversational AI agent used to respond to user queries.

2. `get\_summary\_message(self, message, history)`: This method takes a `message` and a `history` of previous conversation turns as input. It uses a language model (LLM) and a template from `SUMMARY\_PROMPT\_TPL` to generate a summary message based on the historical conversation. The LLM model and the prompt are obtained from helper functions (`get\_llm\_model()` and `Prompt.from\_template()`, respectively). The method returns the generated summary message.

3. `answer(self, message, history)`: This method takes a `message` and a `history` of previous conversation turns as input. If there is a `history`, it calls the `get\_summary\_message()` method to generate a summary message based on the conversation history. Then, it passes the processed `message` to the `query()` method of the `Agent` class to obtain a response. The method returns the response generated by the agent.

Overall, the `Service` class provides a high-level interface for interacting with the conversational AI system. It leverages the `Agent` class to process user queries and generate responses, and it uses a language model and templates to facilitate tasks such as summarizing conversations.

**Utils.py**

It includes several utility functions and configurations for the conversational AI system. Here's an overview of the different components:

1. `get\_embeddings\_model()`: This function retrieves the embeddings model based on the value of the `EMBEDDINGS\_MODEL` environment variable. It returns an instance of the appropriate embeddings model, such as the `OpenAIEmbeddings` model.

2. `get\_llm\_model()`: This function retrieves the language model (LLM) for chat-based interactions based on the value of the `LLM\_MODEL` environment variable. It returns an instance of the appropriate LLM model, such as the `ChatOpenAI` model.

3. `structured\_output\_parser(response\_schemas)`: This function takes a list of response schemas as input and generates a text description of the schemas and their types. This is used to instruct the parser on how to extract entity information from text and output it in JSON format.

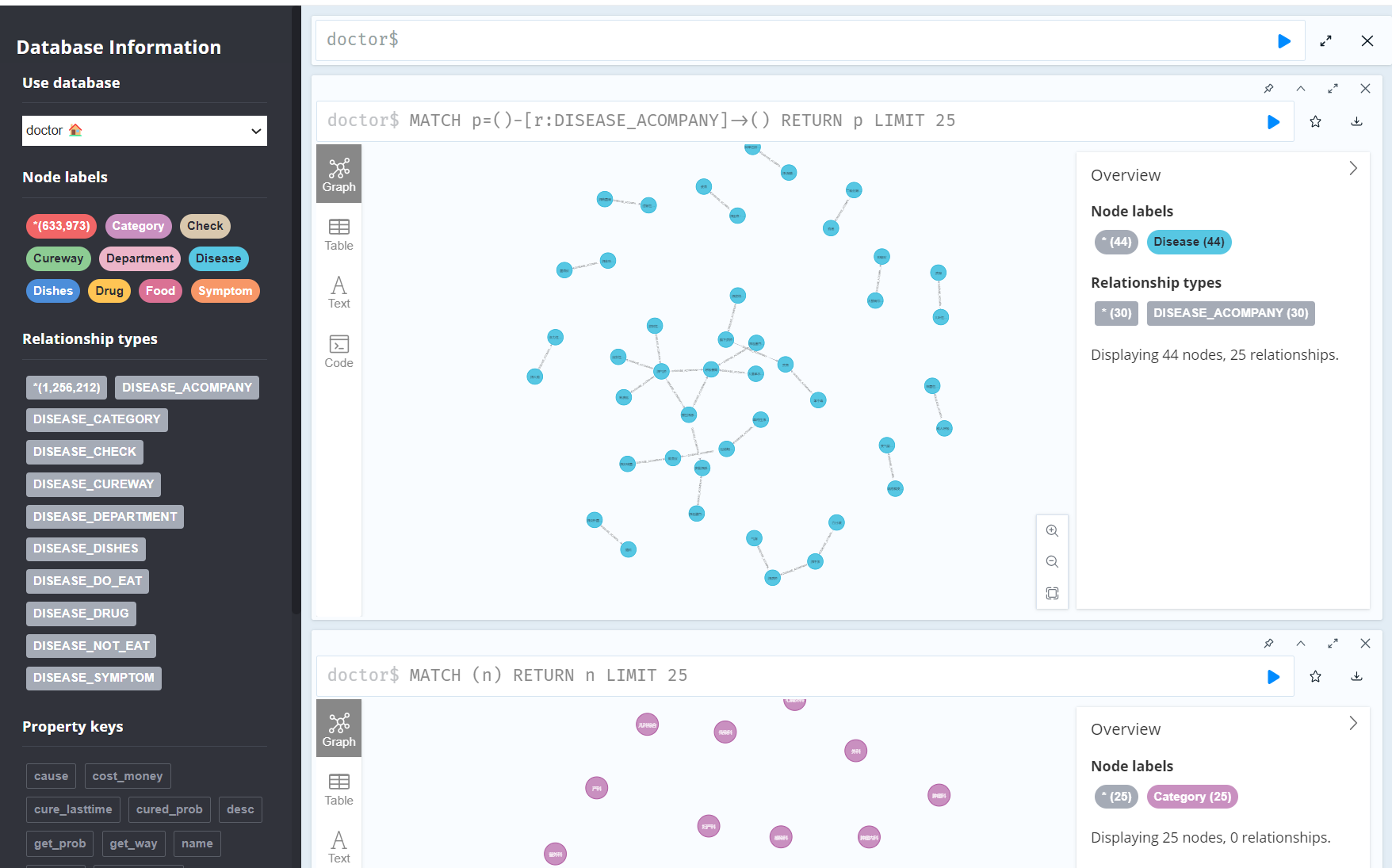
4. `replace\_token\_in\_string(string, slots)`: This function replaces tokens in a string with their corresponding values. It takes a `string` and a list of `slots` as input, where each slot is a key-value pair representing a token and its value. The function iterates over the slots and replaces the tokens in the string with their respective values.

5. `get\_neo4j\_conn()`: This function establishes a connection to a Neo4j database using the credentials and URI specified in the environment variables `NEO4J\_URI`, `NEO4J\_USERNAME`, and `NEO4J\_PASSWORD`. It returns a `Graph` object representing the connection to the Neo4j database.

Additionally, the code imports the necessary modules and libraries, such as `OpenAIEmbeddings`, `ChatOpenAI`, `Graph` from `py2neo`, and `load\_dotenv` from `dotenv`. It also loads environment variables from a `.env` file using `load\_dotenv()`.

Overall, these utility functions and configurations provide essential functionalities for the conversational AI system, including retrieving models, parsing structured outputs, replacing tokens in strings, and establishing connections to a Neo4j database.

**Neo4j Database**



The database takes in a list of disease, symptoms, drugs, etc. and automatically creates an relational graph. The Agent Graph Tool retrieves data from this database.

**Chatbot Interface**

