AI-Driven Medical Assistant: System Documentation

This document provides a comprehensive overview of the AI-driven medical assistance application, detailing its system architecture, agent interaction mechanisms, setup instructions, and the inference process.

1. System Architecture

The medical assistant application is built upon a modular architecture, integrating several key components to provide an interactive and intelligent health assistance experience.

```
graph TD
   User[User Interface (Gradio)] --> Input[User Input];
   Input --> AgentExecutor[LangChain AgentExecutor];
   AgentExecutor -- Invokes --> OpenAI_LLM[OpenAI LLM (gpt-4o-mini)];
   AgentExecutor -- Selects & Calls --> Tools[Custom Tools];
   Tools -- Queries --> FAISS_VectorStore[FAISS Vector Store];
   FAISS_VectorStore -- Contains --> Disease_Data[Disease Data (Symptoms,
Precautions, Descriptions)];
   OpenAI_LLM -- Generates Response --> AgentExecutor;
   AgentExecutor -- Stores/Retrieves --> ConversationBufferMemory[Conversation
Buffer Memory];
   AgentExecutor --> Output[Agent Output];
   Output --> User;
```

Components Breakdown:

- **User Interface (Gradio):** Provides a simple, web-based chat interface for users to interact with the assistant. It handles user input and displays the AI's responses.
- **LangChain AgentExecutor:** This is the core orchestrator. It receives user input, consults the OpenAI LLM, decides which tools to use (if any), executes them, and formats the final response.
- **OpenAI LLM (gpt-4o-mini):** The large language model responsible for understanding user queries, reasoning about which tools to use, and generating natural language responses.
- **Custom Tools:** A set of specialized Python functions designed to interact with the FAISS vector store to retrieve specific medical information (e.g., symptom checking, precaution advice).

- **FAISS Vector Store:** A highly efficient library for similarity search. It stores vectorized representations of disease information, allowing for rapid retrieval of relevant data based on symptom queries.
- **Conversation Buffer Memory:** A component within LangChain that maintains the history of the conversation, allowing the AI to remember previous turns and provide contextually relevant responses.
- **Disease Data:** The underlying knowledge base, likely comprising a collection of diseases with associated symptoms, descriptions, and precautions, which is indexed and stored in the FAISS vector store.

2. Agent Interactions

The LangChain agent is designed to be a "tool-using" agent, leveraging specialized functions to augment its knowledge and capabilities beyond its pre-trained data.

- **LLM (ChatOpenAI):**
    - llm = ChatOpenAI(model='gpt-4o-mini', temperature=0): This initializes the OpenAI language model. gpt-4o-mini is chosen for its balance of capability and cost-effectiveness, and temperature=0 makes its responses more deterministic and factual, which is desirable for a medical assistant.
- **Memory (ConversationBufferMemory):**
    - memory = ConversationBufferMemory(llm=llm, memory_key="chat_history", return_messages=True): This component stores the conversation history. When a new turn occurs, the entire chat_history (user messages and AI responses) is passed to the LLM, allowing it to understand the context and continuity of the conversation.
- **Prompt (ChatPromptTemplate):**
    - The custom_prompt defines the agent's persona and instructions:

      ```
      ChatPromptTemplate.from_messages([
         ("system", "You are a helpful health assistant. You MUST use the provided tools for health or symptom queries. If a tool returns no info, respond with: 'I cannot find information on that topic using my available tools.'"),
         MessagesPlaceholder("chat_history"),
         ("human", "{input}"),
         MessagesPlaceholder("agent_scratchpad")
      ])
      ```

        - **System Message:** Establishes the agent's role and explicitly instructs it to use tools for health-related queries and how to handle cases where tools return no information.

        - **MessagesPlaceholder("chat_history"):** This dynamically inserts the ongoing conversation history into the prompt, enabling the agent to maintain context.
        - **("human", "{input}"):** This is where the current user's query is inserted.
        - **MessagesPlaceholder("agent_scratchpad"):** This is crucial for the agent's internal thought process, where it plans and executes tool calls.
- **Custom Tools:** These are Python functions decorated with @tool, making them discoverable and executable by the LangChain agent. Each tool serves a specific purpose by querying the FAISS vector store.
    - **symptom_checker_tool(symptom_query: str) -> str:**
        - **Purpose:** To identify potential diseases based on a user's described symptoms.
        - **Mechanism:** Performs a similarity search in the FAISS vector store using the symptom_query to find the top 3 most relevant disease documents.
    - **precaution_advisor(symptom_query: str) -> str:**
        - **Purpose:** To provide precautionary advice related to described symptoms.
        - **Mechanism:** Retrieves relevant documents from the vector store and extracts associated 'precautions' from their metadata.
    - **severity_risk_estimator(symptom_list: str) -> str:**
        - **Purpose:** To estimate the severity risk based on a list of symptoms.
        - **Mechanism:** Parses a comma-separated list of symptoms, finds relevant documents, aggregates severity scores from metadata, and calculates an average severity to determine a "Low," "Medium," or "High" risk.
    - **symptom_to_disease_matcher(symptoms_query: str) -> str:**
        - **Purpose:** To find diseases that match multiple symptoms provided by the user.
        - **Mechanism:** Retrieves relevant documents and returns the disease name and description for each match.
    - **faq_disease_info(disease_name: str) -> str:**
        - **Purpose:** To retrieve detailed information (description, symptoms, precautions) about a specific disease.
        - **Mechanism:** Fetches documents relevant to the disease_name and extracts comprehensive metadata.
- **Agent Creation and Execution:**
    - agent = create_openai_functions_agent(llm=llm, tools=tools, prompt=custom_prompt): This constructs the agent, giving it access to the LLM, the defined tools, and the custom prompt. This type of agent is specifically designed to work with OpenAI's function-calling capabilities.
    - agent_executor = AgentExecutor(agent=agent, tools=tools, memory=memory,

verbose=True, handle_parsing_errors=True): This is the runtime for the agent. It manages the agent's decision-making process, tool execution, and interaction with memory. verbose=True is useful for debugging, showing the agent's "thought process." handle_parsing_errors=True helps the agent recover from malformed tool outputs.

3. Setup Guide

To run this application locally or deploy it, follow these steps:

Prerequisites

- Python 3.8+ installed.
- pip (Python package installer).
- Git (for cloning repositories, if applicable).

Step 1: Clone the Repository (if applicable)

If your application code is in a Git repository, clone it:

```
git clone <your-repository-url>
cd <your-repository-name>
```

Otherwise, navigate to the directory containing your app.py file.

Step 2: Create a Virtual Environment

It's highly recommended to use a virtual environment to manage dependencies:

```
python -m venv venv
# On macOS/Linux
source venv/bin/activate
# On Windows (Command Prompt)
venv\Scripts\activate.bat
# On Windows (PowerShell)
venv\Scripts\Activate.ps1
```

Step 3: Install Dependencies

Create a requirements.txt file in the same directory as your app.py with the following content:

```
gradio
```

```
langchain
langchain-openai
langchain-community
faiss-cpu # Or faiss-gpu if you have a compatible GPU
openai
```

Then, install them:

```
pip install -r requirements.txt
```

Step 4: Prepare the FAISS Vector Store (disease_faiss_store)

Crucial Step: The application relies on a pre-built FAISS vector store. The error you previously encountered (No such file or directory) indicates this store is missing or misplaced.

You need to have a separate script or process that generates this disease_faiss_store directory, containing index.faiss and index.pkl files, from your raw disease data.

Example of how disease_faiss_store would typically be created (This code is illustrative and assumes you have disease_data.json or similar):

```
# This is a conceptual example of how to create the FAISS store.
# You would run this *once* to generate the 'disease_faiss_store' directory.

import json
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings
from langchain_core.documents import Document
import os

# Ensure your OpenAI API key is set for embeddings generation
# os.environ["OPENAI_API_KEY"] = "YOUR_OPENAI_API_KEY" # Set this if not already
in env

# --- Dummy Data Structure (Replace with your actual data loading) ---
# Your actual disease data would come from a database, CSV, JSON file, etc.
# Each item should ideally have 'disease', 'description', 'symptoms', 'precautions',
'severity'
```

```
dummy_disease_data = [
   {"disease": "Common Cold", "description": "A viral infectious disease of the upper respiratory tract.",
     "symptoms": ["runny nose", "sore throat", "cough", "sneezing", "headache"], "precautions": ["rest", "fluids", "hand washing"], "severity": 1},
   {"disease": "Influenza (Flu)", "description": "A common viral infection that can be deadly.",
     "symptoms": ["fever", "body aches", "fatigue", "cough", "sore throat", "headache"], "precautions": ["vaccination", "rest", "fluids"], "severity": 3},
   {"disease": "Migraine", "description": "A headache of varying intensity, often accompanied by nausea and sensitivity to light and sound.",
     "symptoms": ["severe headache", "nausea", "vomiting", "light sensitivity", "sound sensitivity"], "precautions": ["avoid triggers", "medication", "dark quiet room"], "severity": 4},
   {"disease": "Allergies", "description": "An immune system reaction to a substance that's usually harmless.",
     "symptoms": ["sneezing", "itchy eyes", "runny nose", "skin rash"], "precautions": ["avoid allergens", "antihistamines"], "severity": 1},
   {"disease": "Strep Throat", "description": "A bacterial infection that can make your throat sore and scratchy.",
     "symptoms": ["sore throat", "difficulty swallowing", "fever", "tiny red spots on roof of mouth"], "precautions": ["antibiotics", "rest"], "severity": 2},
]

# Convert your data into LangChain Document objects
documents = []
for item in dummy_disease_data:
   # Combine relevant fields for the page_content to be embedded
   page_content = f"Disease: {item['disease']}. Description: {item['description']}. Symptoms: {', '.join(item['symptoms'])}. Precautions: {', '.join(item['precautions'])}."
   doc = Document(page_content=page_content, metadata=item)
   documents.append(doc)

# Initialize embeddings
embeddings = OpenAIEmbeddings()

# Create the FAISS vector store
print("Creating FAISS vector store...")
vectorstore = FAISS.from_documents(documents, embeddings)
```

```
# Save the vector store locally
store_path = './disease_faiss_store'
if not os.path.exists(store_path):
    os.makedirs(store_path)
vectorstore.save_local(store_path)
print(f"FAISS vector store saved to {store_path}")
```

Action Required:

- You need to have a script similar to the example above that processes your actual disease data and saves it to the disease_faiss_store directory.
- Ensure the disease_faiss_store directory (containing index.faiss and index.pkl) is located in the **same directory** as your app.py script.

Step 5: Set Your OpenAI API Key

Your application requires an OpenAI API key to interact with the ChatOpenAI and OpenAIEmbeddings models.

For Local Execution (Temporary for current terminal session):

- **macOS/Linux:**
  ```
  export OPENAI_API_KEY="YOUR_OPENAI_API_KEY_HERE"
  ```
- **Windows (Command Prompt):**
  ```
  set OPENAI_API_KEY=YOUR_OPENAI_API_KEY_HERE
  ```
- **Windows (PowerShell):**
  ```
  $env:OPENAI_API_KEY="YOUR_OPENAI_API_KEY_HERE"
  ```
  Replace YOUR_OPENAI_API_KEY_HERE with your actual key.

For Local Execution (Permanent):

- **macOS/Linux:** Add export OPENAI_API_KEY="YOUR_OPENAI_API_KEY_HERE" to your ~/.bashrc, ~/.zshrc, or ~/.bash_profile file and then run source ~/.bashrc (or your respective file).
- **Windows:** Set it as a user environment variable through System Properties (search "Environment Variables" in Start Menu).

Step 6: Run the Application

Once the FAISS store is in place and your API key is set, you can run your app.py script:

python app.py

Gradio will then provide a local URL (e.g., http://127.0.0.1:7860) and a public shareable URL if share=True is enabled in demo.launch().
[https://huggingface.co/spaces/sbaguma/HealthBot]

4. Inference Process Documentation (How the AI Processes a Query)

This section details the step-by-step process the AI-driven medical assistant follows from receiving a user query to generating a response.

1. **User Input:** The user types a query into the Gradio chat interface (e.g., "I have a runny nose and cough, what could it be?").
2. **Input to AgentExecutor:** The chat_with_agent function in app.py receives the user_input and the current chat_history. It then invokes the agent_executor with this information.
3. **Agent's Thought Process (LLM Reasoning):**
    - The AgentExecutor sends the custom_prompt (including the system message, chat_history, and the new user_input) to the ChatOpenAI LLM.
    - The LLM analyzes the query and the conversation context. Based on its training and the instructions in the system prompt (especially "You MUST use the provided tools for health or symptom queries"), it determines if a tool is needed.
    - If a tool is needed, the LLM decides *which* tool is most appropriate and *what arguments* to pass to that tool (e.g., for "I have a runny nose and cough", it might decide to call symptom_checker_tool with symptom_query="runny nose, cough"). This decision-making process is part of OpenAI's function-calling capability integrated with LangChain.
4. **Tool Execution:**
    - If the LLM decides to use a tool, the AgentExecutor executes the selected Python function (e.g., symptom_checker_tool).
    - This tool then interacts with the FAISS vector store. For symptom_checker_tool, it performs a similarity search on the vectorized disease data using the symptom_query.
5. **Tool Output:**

    - The tool returns its result (e.g., a list of matching diseases and their descriptions) back to the AgentExecutor.
6. **LLM Generates Final Response:**
    - The AgentExecutor takes the original user query, the conversation history, and the *output from the tool* and sends all of this back to the ChatOpenAI LLM.
    - The LLM then synthesizes this information into a natural, conversational response for the user. For instance, if symptom_checker_tool returned "Common Cold, Influenza", the LLM might generate: "Based on your symptoms, it could potentially be a Common Cold or Influenza. Would you like more information on either?"
7. **Update Chat History & Display:**
    - The chat_with_agent function appends the user_input and the result['output'] (the AI's response) to the chat_history state.
    - The Gradio chatbot component is updated to display the latest turn of the conversation.

This iterative process allows the AI to provide dynamic, informed, and context-aware medical assistance by combining the reasoning power of a large language model with the precise, retrievable knowledge stored in the FAISS vector store.