Building a Patient Information Collection and Advisory Chatbot Agent

1. Objectives

- Develop an AI-powered chatbot agent that interactively collects patient information such as name, age, and symptoms.
- Provide preliminary health advice based on the collected patient data.
- Utilize **LangGraph** to manage the conversational flow and state.
- Employ **AzureChatOpenAI** as the language model through **LangChain**.
- Optionally integrate **Tavily** to fetch real-time information to supplement chatbot responses.

2. Problem Statement

Patients often need quick, preliminary health assessments before seeing healthcare professionals. This exercise aims to build an AI-driven chatbot agent that:

- Engages users in a natural conversational manner to gather essential patient details.
- Analyzes the collected information to offer relevant, preliminary health advice or recommendations.
- Optionally enhances advice with real-time web information via Tavily.
- Improves user experience with interactive, guided dialogue flow controlled by LangGraph.

3. Inputs / Shared Artifacts

- Python 3.x environment (preferably create virtual enviroment).
- Installed packages: langchain, langgraph, openai, and optionally tavily.
- Azure OpenAI API access with deployment configured.
- (Optional) Tavily API key for real-time web search integration.
- Basic understanding of prompt engineering and conversational AI design.

4. Expected Outcomes

- A conversational chatbot agent that collects patient details interactively.
- The agent provides preliminary health advice based on user input.
- When integrated, it supplements advice with relevant real-time information.
- A documented Jupyter Notebook showcasing the full implementation and sample interactions.
- Hands-on experience with LangGraph for managing complex conversation flows.
- Practical skills using AzureChatOpenAI for conversational AI.

5. Concepts Covered

- **Conversational AI:** Building a chatbot that interacts naturally with users.
- **Conversation Flow Management:** Using LangGraph to design and control dialogue states.
- **Language Model Integration:** Leveraging AzureChatOpenAI through LangChain for generating context-aware responses.
- **Optional Real-Time Data Retrieval:** Using Tavily to provide dynamic, up-to-date information during the conversation.
- **Prompt Engineering:** Crafting effective prompts for health-related advice generation.

6. Example: Step-by-Step Instructions and Code Demo

Step 1: Imports

```
from langchain_openai import AzureChatOpenAI

from langgraph.graph import StateGraph, END

from typing import TypedDict, Annotated

import operator

import os
```

Step 2: Define State Schema

```
class ChatState(TypedDict):

      messages: Annotated[list, operator.add]

      name: str

      age: str

      symptoms: str

      current_step: str
```

Step 3: Initialize LLM

```
llm = AzureChatOpenAI(

      azure_endpoint="https://your-endpoint.openai.azure.com/",

      api_key="your-api-key",

      api_version="2024-02-15-preview",

      deployment_name="your-deployment",

      temperature=0.7)
```

Step 4: Define Node Functions

```
def ask_name(state: ChatState):

      return { "messages": [("assistant", "Hello I am a Healthcare AI
Assistant. May I know what is your name?")],

      "current_step": "waiting_name" }

def ask_age(state: ChatState):

      name = state.get("name", "bạn")

      return { "messages": [("assistant", f"Cảm ơn {name}! How old are
you   ?")], "current_step": "waiting_age" }

def ask_symptoms(state: ChatState):

      return { "messages": [("assistant", " Please describe the
      symptoms you are experiencing.")], "current_step":
      "waiting_symptoms" }

def provide_advice(state: ChatState):

      prompt = f"""Patient Information:
      - Name: {state['name']}
      - Age: {state['age']}
      - Symptoms: {state['symptoms']}


      Please provide preliminary health advice based on the above
      information.
      Note: This is only preliminary advice; the patient should consult
      a specialist doctor.
      """


      response = llm.invoke([{"role": "user", "content": prompt}])
      return {
          "messages": [("assistant", f"Based on the information you
      provided:\n\n{response.content}")],
          "current_step": "completed"
```

```
    }
```

Step 5: Build Graph

```
def create_chatbot():

    workflow = StateGraph(ChatState)

    # Add nodes
    workflow.add_node("ask_name", ask_name)
    workflow.add_node("ask_age", ask_age)
    workflow.add_node("ask_symptoms", ask_symptoms)
    workflow.add_node("provide_advice", provide_advice)


    # Define edges
    workflow.add_edge("ask_name", "ask_age")
    workflow.add_edge("ask_age", "ask_symptoms")
    workflow.add_edge("ask_symptoms", "provide_advice")
    workflow.add_edge("provide_advice", END)


    # Set entry point
    workflow.set_entry_point("ask_name")


    return workflow.compile()
```

Step 6: Usage Example

```
def run_chatbot(): app = create_chatbot()

# Initialize state
initial_state = {
    "messages": [],
    "name": "",
    "age": "",
    "symptoms": "",
    "current_step": "start"
}

# Run conversation
result = app.invoke(initial_state)

# Display conversation
for role, message in result["messages"]:
    print(f"{role.capitalize()}: {message}")
```

Step 7: Run the chatbot

```
if __name__ == "main":

      run_chatbot()
```

7. Final Submission Checklist

- Submit your Python script or notebook containing the full code
- Include sample inputs and outputs.