Lesson 3 Demo 3

Configure Multi-Step Reasoning in AutoGen for AI-Powered Troubleshooting

Objective: To build an AI-powered chatbot using AutoGen that implements multi-step reasoning for structured troubleshooting, improving issue resolution efficiency

You are developing an AI assistant that helps users troubleshoot technical issues in a structured, step-by-step manner. Traditional AI chatbots provide single-response answers without confirming if the problem is resolved, leading to inefficiencies and frustration. By implementing multi-step reasoning, your chatbot will analyze the issue, suggest possible causes, and guide users through an interactive troubleshooting process, adapting responses based on their inputs.

Prerequisites:

1. Create a virtual environment
2. Install dependencies

Steps to be followed:

Step 1: Set up the environment

Step 2: Set Up Libraries and Configure OpenAI Client

Step 2: Define the IT Support Chatbot Class

Step 3: Streamlit Web App Setup for Interaction

Step 4: Handle User Input and Display Responses

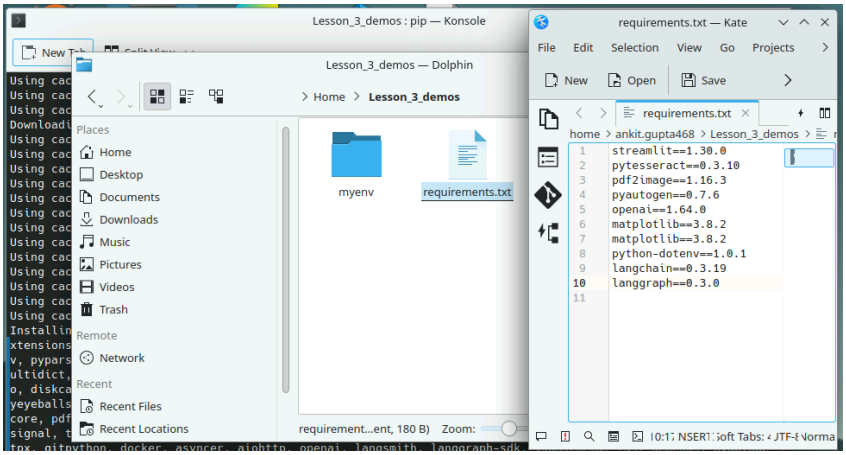
Step 5: Run the code

Step 1: Set up the environment

1.1 Open command prompt and go to the "**Lesson_3_demos**" folder (which we created in Demo_1) using the command given below:
mkdir Lesson_3_demos (not needed if the folder is already created in Demo1)
cd Lesson_3_demos

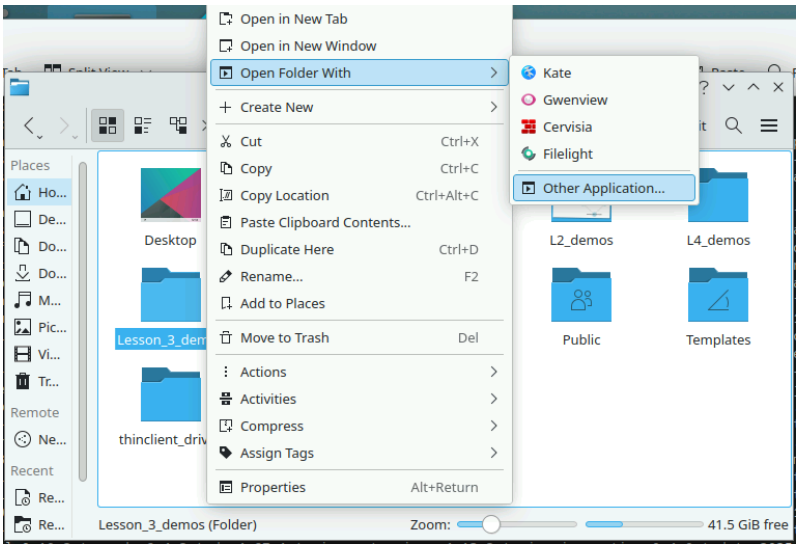
1.2 After this, activate the virtual environment using the command below:
python3 -m venv venv (not needed if the virtual env. is already created in Demo1)
source venv/bin/activate

1.3 Now, create a `requirements.txt` file inside the folder with required libraries (not needed if already done in Demo1):



1.4 Install all the required libraries using the command below:
pip install -r requirements.txt (not needed if already done in Demo1)

1.5 Now, open the folder using VS Code editor:



After this, open a new Python file using the "**New File**" option and name it as "**Demo3**".

Step 2: Set Up Libraries and Configure OpenAI Client

2.1 Start by importing necessary libraries such as **autogen** for creating the chatbot agent, **openai** for interacting with OpenAI models, and **streamlit** for the web interface.

2.2 Set up the **OpenAI client** with the appropriate API key and endpoint to connect to the **Azure GPT model**.

```
import streamlit as st

import autogen

import openai

from dotenv import load_dotenv

import os


client = openai.AzureOpenAI(


api_key="2ABecnfxzhRg4M5D6pBKiqxXVhmGB2WvQ0aYKkbTCPsj0JLKsZPfJQQJ99BDAC77bzfXJ3w3AAABACOGi
3sC",

    api_version="2023-12-01-preview",

    # azure_endpoint="https://openai-api-management-gw.azure-api.net/"

    azure_endpoint="https://openai-api-management-gw.azure-api.net/deployments/gpt-4o-
mini/chat/completions?api-version=2023-12-01-preview"

)
```

Step 3: Define the IT Support Chatbot Class

3.1 Define the **MultiStepITSupportBot** class, which will manage the IT support process.

3.2 The class uses **multi-step reasoning**, meaning it will guide the user through a structured troubleshooting process.

3.3 The chatbot generates responses by calling **OpenAI's GPT** with a prompt containing the user's issue and previous steps.

```
# Step 2: Define an IT Support chatbot with customizable behavior

class CustomBehaviorITSupportBot(autogen.AssistantAgent):

    def __init__(self, name, model="gpt-4o-mini", response_style="detailed",
troubleshooting_priority="basic"):

        """

        Initializes the chatbot with:

        - A name for identification.

        - A selected GPT model (default: GPT-3.5 Turbo).

        - A response style that can be modified (e.g., 'detailed', 'concise', 'formal',
'casual').

        - A troubleshooting priority that determines whether to start with 'basic' or
'advanced' solutions.

        """

        super().__init__(name=name)

        self.model = model

        self.response_style = response_style  # Customize how the bot replies

        self.troubleshooting_priority = troubleshooting_priority  # Determines problem-
solving approach


    def generate_reply(self, message):

        """

        Step 3: Handles user queries while considering customized behavior.

        - Adjusts response style (detailed, concise, formal, casual).

        - Prioritizes troubleshooting solutions based on user preferences.

        - Dynamically adapts responses based on issue complexity.

        """

        response = self._get_gpt_response(message)

        return response
```

```
    def _get_gpt_response(self, message):
        """
        Step 4: Uses OpenAI's GPT to generate a response based on:
        - Response style (how information is presented).
        - Troubleshooting priority (whether to start with basic or advanced solutions).
        - Adaptability to user feedback.
        """

        prompt = f"""
        The user reported an IT issue: "{message}".
        You are an IT support assistant with the following behavior settings:
        - Response style: {self.response_style}
        - Troubleshooting priority: {self.troubleshooting_priority}

        Follow these guidelines:
        1. If troubleshooting priority is 'basic', suggest simple fixes first before advanced solutions.
        2. If response style is 'concise', keep responses under 3 sentences.
        3. If response style is 'formal', maintain professionalism in wording.
        4. If response style is 'casual', use a friendly and relaxed tone.
        5. Adapt troubleshooting steps dynamically based on user feedback.
        """

        response = client.chat.completions.create(
            model=self.model,
            messages=[
                {"role": "system", "content": "You are an IT support assistant that adapts to user preferences and troubleshooting needs."},
                {"role": "user", "content": prompt}
            ]
        )

        return response.choices[0].message.content.strip()
```

Step 4: Streamlit Web App Setup for Interaction

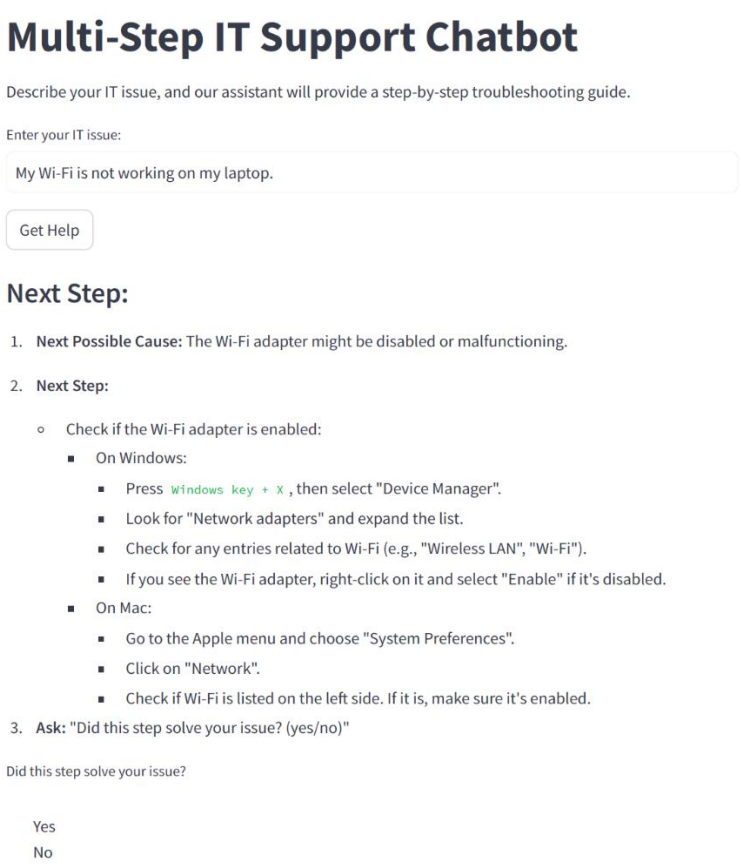
4.1 Set up the **Streamlit** UI to enable real-time interaction with the user.
4.2 Display the chatbot's title and initial instructions to the user.
4.3 Ensure that a chatbot object is created when the app is first run, and maintain the session state to track the user's issue and previous steps.

```
# Step 5: Deploy using Streamlit

st.title("AI-Powered IT Support Chatbot")

st.write("Customizable AI assistant that adapts response style and troubleshooting
approach.")



# User settings

response_style = st.selectbox("Select Response Style:", ["detailed", "concise", "formal",
"casual"])

troubleshooting_priority = st.selectbox("Select Troubleshooting Priority:", ["basic",
"advanced"])



# Initialize chatbot with selected settings

bot = CustomBehaviorITSupportBot(name="AdaptiveHelpBot", response_style=response_style,
troubleshooting_priority=troubleshooting_priority)



# User input section

user_input = st.text_area("Enter your IT issue:")



if st.button("Get Help"):

    if user_input.strip():

        response = bot.generate_reply(user_input)

        st.subheader("AI Response:")

        st.write(response)

    else:

        st.warning("Please enter a valid IT issue.")
```

Step 5: Run the code

5.1 Save the file and then run the streamlit webapp from command prompt using the command given below:
streamlit run Demo3.py

Output:



By following the above-mentioned steps, you have successfully showcased how to build an AI-powered IT support chatbot using AutoGen, OpenAI, and Streamlit. By implementing multi-step reasoning, the chatbot provides structured troubleshooting rather than generic responses, ensuring a more interactive and efficient problem-solving experience.