Building a LangGraph Node powered by a LLM:

For building a LangGraph Node we first need to install the packages langgraph, langchain, langchain-google-genai, google-generativeai. We also need to get our GOOGLE_API_KEY from Google AI Studio. In this I had used the model of Gemini-1.5-flash.

Strategy to make a safe entering of API_KEY:

To make the process of entering the API_KEY safer I had used the getpass() function which shows the API_KEY as dots while entering or it has been entered.

** NOTE: For the llm I have defined the as gemini-1.5-flash and temperature = 0.3 which makes the model more deterministic, favouring more reliable and safer responses.

Calculator Tool :

I have first defined the calculator tool as a LangGraph supported tool using @tool rom langchain_core.tools. The tool takes a string input and returns a string output.

The tool finds out if there is all the characters of the input string are either numbers, (,), operations like +-/* or " " (spaces) it puts the string (replaces the " " with "") into the eval() function which then evaluates the expression using BODMAS rule.

If apart from the above mentioned characters if it finds something else it invokes the llm and passes on the string to it.

CHATBOT NODE :

The chatbot_node is function which takes in the data type of GraphState and returns a GraphState. GraphState is a TypeDict which has two keys "input" and "output" both of which takes in strings.

The chatbot_node is integrated with the calculator tool it passes the input string to calculator if it finds any arithmetic operation like +-/* in this or else invokes the llm.

The "output" key's value is updated as the output from the calculator tool or the llm and finally this updated GraphState data type is returned.

** Then finally a graph is defined and chatbot_node is added as a node to it and it set as an entry as well as finish point since it is only the single node here in this graph. The graph is then compiled using .compile() and the return value is stored in another variable which is then invoked using .invoke() where a GraphState type Dictionary is passed for seeing the output.

**NOTE: I have also shown the image of the graph where it can be seen that there is a starting point and a ending point and a chatbot_node in between. This is the overall structure of the graph made till now.

Fashion Tool :

The fashion_tool is a custom LangChain-compatible tool that uses spaCy's NLP capabilities to extract a geographical location (like a city or country) from the user's input. Once it identifies a location, it sends a prompt to an LLM (Gemini) asking for 4–5 examples of trending clothes in that place.

Key steps performed by Fashion tool:

1. Named Entity Recognition (NER): Uses spacy to detect if the user mentioned any geographical location (GPE entity).

2. LLM Query: If a location is found, it asks the LLM: "Give 4-5 examples of trending clothes in <location>. Please don't give very descriptive answers."

3. Fallback: If no location is found, it returns a message asking the user to include a city or country.

4. Error Handling: If something goes wrong, it returns an error message.

Weather Tool :

The weather tool whose name is (get_weather) is a custom LangChain-compatible tool that uses spaCy's NLP capabilities to extract a geographical location(like a city or country) from the user's input. It takes a dictionary as input, one key is input_text and other key is API_KEY. Both of them take the user input and the Open Weather API key as values. Finally return a string describing the weather of the place.

Key steps performed by the weather tool:

1. Identifies the City: It uses spaCy to extract the name of a city or country from the user's input. If nothing is detected, it assumes the entire input is the city name.

2. Builds the API Request: It constructs a URL to fetch weather data from the OpenWeatherMap API, inserting the detected city name and your API key into the endpoint: http://api.openweathermap.org/data/2.5/weather?q={city_name}&units=metric&appid={API_KEY}

3. Calls the Weather API: It sends a request to this URL to get the latest weather data for the city.

4. Processes the Data: Once the data is received, the tool pulls out useful info like temperature, humidity, pressure, cloud cover, wind speed, and even sunrise and sunset times.

5. Returns a Summary: It formats all of this into a clean and readable weather report for the user.

6. Handles Errors Gracefully: If something goes wrong—like an invalid city or missing data—it returns a clear error message explaining the issue.

CONTINUOUS CONVERSATION :

This script creates a simple conversational chatbot using LangGraph. It can handle three types of queries: math calculations, weather updates, and fashion trends—one topic at a time.

At the start, the bot greets the user and sets up a basic memory dictionary to store the conversation history between the user and the bot. The conversation runs in a loop and ends if the user types "exit".

The core function, chatbot_node, looks at the user's input and decides how to respond:

If the input contains math symbols (+, -, *, /), it uses the calculator tool.

If it includes weather-related terms (like "temperature", "humidity", etc.), it prompts for an OpenWeatherMap API key (if not already set), and fetches weather data using the get_weather tool.

If the input is about fashion or trends, it uses the fashion_tool to fetch current clothing styles based on location.

The chatbot uses LangGraph's StateGraph to define a simple workflow with one node (chatbot). It executes the graph each time the user sends a message and returns the appropriate response, which gets printed and saved in the memory log.

This makes it a lightweight, tool-enhanced chatbot with basic memory and simple routing logic.

*** I have also made two tools a graph plotter tool and a unit_converter tool after this ***.