***LLM Integration:***

***While started this project, we have tried the below approaches to Integrate the LLM with our backend application. Below I have sorted the approaches that we tried from the older one to current implementations.***

***1. LLM Integration using Langgraph:***

***This is the first approach that we have tried when we try to integrate the LLM along with the conversation history, tools, TavilySearchResults ( which is a search engine feature). And for this we utilized the framework called Langgraph which implements a graph behind the scenes to response to the user queries.***

***This Langgraph works more similar like Agent (Agent Executor) which is the current implementation found in our code base for LLM integration. It has multiple tool calling capabilities like Agent and provides greater responses to user queries. However this approach provides good results at early stages, Due to insufficient control over the responses of the graph. We would like to move this something simple and provides more control.***

***You could find the resources to implement the langgraph by looking into the link:***[***https://langchain-ai.github.io/langgraph/tutorials/customer-support/customer-support/***](https://langchain-ai.github.io/langgraph/tutorials/customer-support/customer-support/)

***2. LLM Integration using Langchain’s Chain:***

***This is the next approach that we tried after lacking control over responses in the previous approach.***

***To begin with first of all we gathered the required modules needed for the LLM to response to user queries including the LLM(gpt-40-mini), ConversationHistory, Prompt (instructions for the llm to act), Tool Executor and so on. Let me break down each and everything along with its coding implementation.***

***2.1) Components Required for Building LLM Chatbot:***

***Here I will break down each and every components that most of the our approaches uses***

***to create a LLM chatbot. And most of these components are common across every approaches.***

***2.1.1) Conversation History:***

***First of all we need to store the conversation that is happening between the user and LLM. The reason for storing this conversation is that, We don’t want our LLM to response our questions solely based on the question alone, Instead we would like to expect our LLMs need to answer our questions based on the current question that is being asked and the previous conversations that we did the LLM. By getting these two (current question and previous conversations) a LLM can generate a great response based on the context that we are currently talking which make more sense.***

***For acheiving this conversation history, we are using two classes to implement the same.***

***# BaseChatMessageHistory – Responsible for Message Storage & Retreival. Internally used by the Below Class for saving the response.***

***# ConversationSummaryBufferMemory – Responsible for summarizing the conversation message when the conversation goes out of specified range by using pruning technique.***

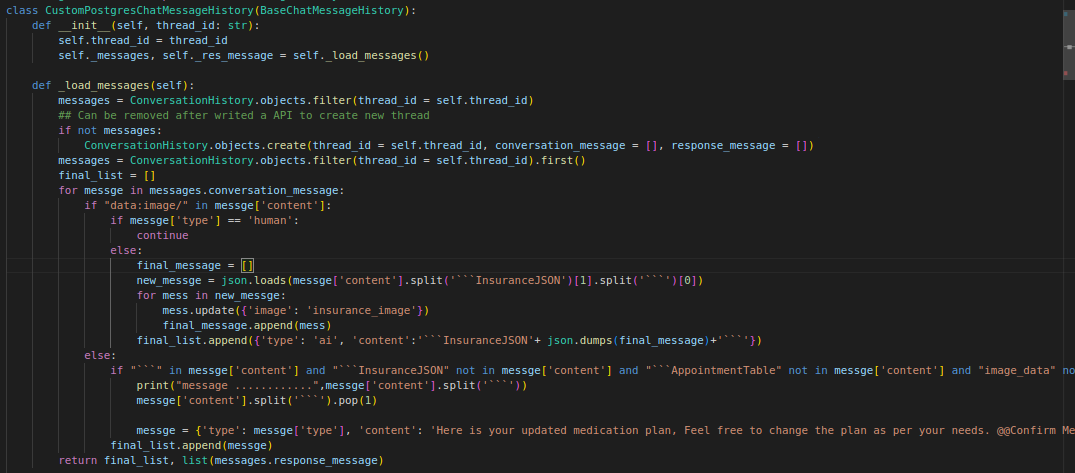
***2.1.1.1) BaseChatMessageHistory:***

***For Implementing this we utilized langchain built in conversation memory and overwritten the same for our logic.***

from langchain.schema import BaseChatMessageHistory

class CustomPostgresChatMessageHistory(BaseChatMessageHistory):

***The above given code block is where we inherited the langchain’s built in conversation memory “BaseChatMessageHistory”. And our new class consists of mainly four methods which are explained below.***

******

***In the above code snippet you can find the first two methods of our conversation memory class,***

***2.1.1.1.1) \_\_init\_\_() method:***

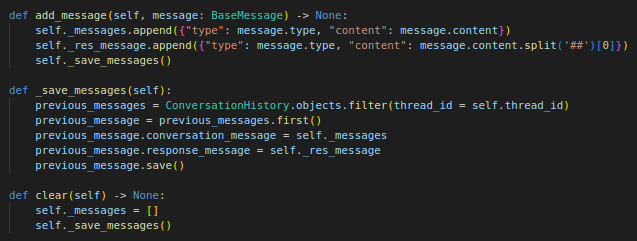
***In which the first one is our constructor method which is \_\_init\_\_ which loads the conversation message from our Database Table called “Conversation History” by calling the load\_message() function and stores it’s inside the self.\_messages variable. As It’s a constructor it’s being called automatically when creating a object for our class and receives the thread id as it’s input and stores the same.***

***2.1.1.1.2) \_load\_messages() method:***

***This method is responsible for fetching the conversation messages stored in the “Conversation History” Table by filtering with thread id provided during object creation. If there is no entry for the given threadid in the table, It creates a new entry for the given threadid in the Conversation History Table.***

***After fetching the conversation message from the table, It apply transformations to the conversation message especially if they are holds any image data to some texts like “insurance\_image” to avoid sending the original base64 content of the image, which trastically increasing the token limit of the request and may lead to skipping steps inside the usual workflows including appointment, medication reminder...***

***This function is responsible for fetching the conversation data from the table and store it for further steps by applying transformations to image data to reduce the number of tokens***

******

***2.1.1.1.3) \_save\_messages() method:***

***This method is responsible for saving the new conversation message pairs (human & AI) to the database with relevant thread id.***

***2.1.1.1.4) add\_message() method:***

***This method is responsible for appending the Human & AI message pairs to our class data attribute called \_messages, \_res\_message so that it can be saved as a JSON by calling the \_save\_messages() as mentioned in the code***

***2.1.1.1.5) clear() method:***

***This clear method is used to empty the \_messages variable of the class by assigning the empty list as the value [] After this we immediately calling the save message to reflect the same in the database for relevant thread id.***

***Next up we are moving to the next Class, Which is responsible for Saving and Loading the conversation messages by using the first class that we discussed above behind the scenes.***

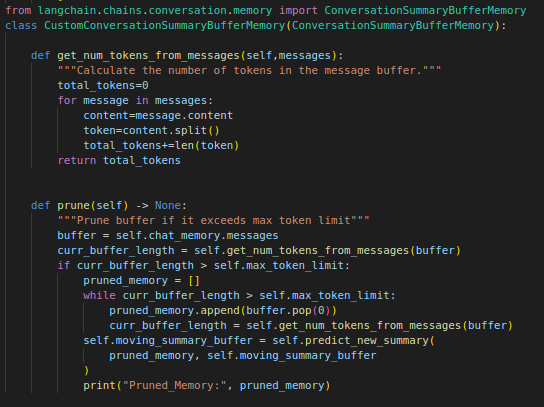
***2.1.1.2) ConversationSummaryBufferMemory:***

***For implementing this also we utilized langchain’s memory package to implement the same. As this class is mostly used to summarizing the conversation message when the conversation message is gone out of the specified token limit. The way it implements this summarization is using a method called pruning.***

***As we didn’t implemented the feature fully which is summarization, let me go through how this works at a higher level. We mostly use this as interface to get and save the conversation message in realtime.***

from langchain.chains.conversation.memory import ConversationSummaryBufferMemory

class CustomConversationSummaryBufferMemory(ConversationSummaryBufferMemory):



***If the above picture you can see the definition of our class that is overriding the “ConversationSummaryBufferMemory”. And It mainly consists of two methods and let me explain the same in below pages.***

***2.1.1.2.1) get\_num\_tokens\_from\_messages() method:***

***This function is responsible for counting the number of tokens present in the messages. And used as a helper function to our prune() method.***

***2.1.1.2.2) prune() method:***

***This method first calculates the token count of the given sentence and while the token count of given sentence is higher than the max token limit, the method pops the initial message of the conversation and stores them inside a list for future summarization.***

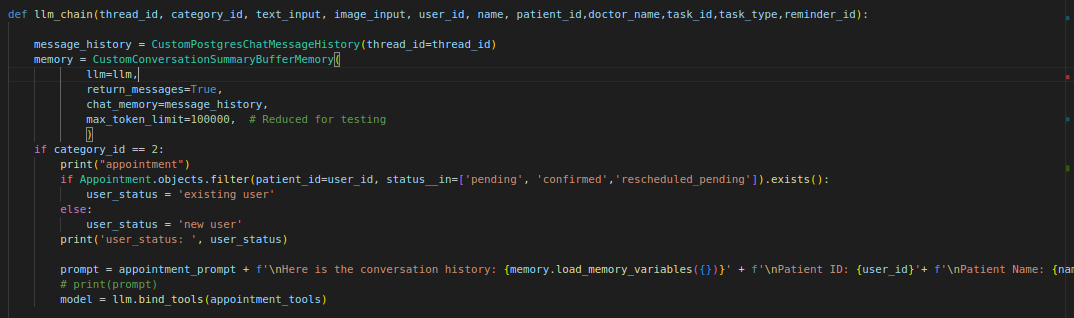
***After spiltted the additional initial content from the given sentence, the uses a llm to get the summary out of it which always will be lower then the additional content and added to the conversation message.***

***However this pruning feature is not implemented properly inside our system. This is a high level approach of how pruning works in real time.***

***Before closing this Conversation Memory section let me just explain where would we using these in our LLM pipeline. This class is mainly utilized on two areas***

***# Fetching the conversation message relevant to thread id***

***# Saving the new conversation pair to the database***

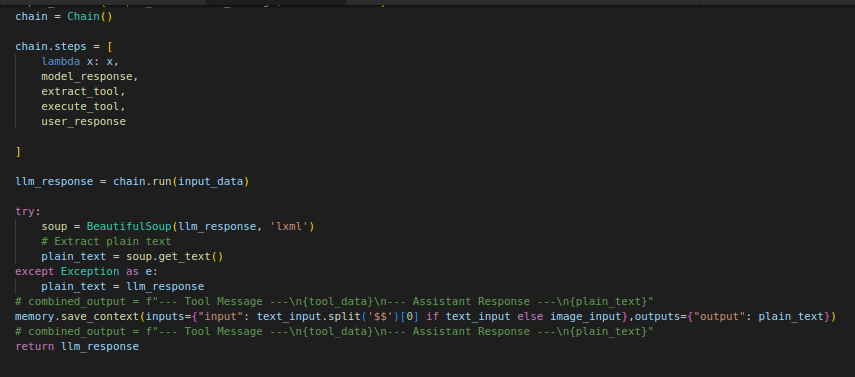


***# Fetching the conversation message relevant to thread id:***

***In the above code block you can see the implementation of the two classes that we have defined before. And the first time we utilized these classes by calling the load\_memory\_variables() method of the CustomConversationSummaryBufferMemory Class which will load the conversation message of the given thread id from the database.***

***The way it doing is that when callling the load\_memory\_variables() method of the CustomConversationSummaryBufferMemory class, It returns the value called “chat\_memory.messages” which is nothing but a object’s variable of our BaseChatMessageHistory class which holds the conversation history of the specified thread id’d conversation.***

***# Saving the new conversation pair to the database:***



***From the above code block you can find a snippet that is calling the ConversationSummaryBufferMemory class’s save\_context() method. This method is responsible for saving the current human and ai message pair to our database using our BaseChatMessageHistory class’s add\_message() function behind the scences. So these are the areas where we mostly use our conversation history activities.***

***2.1.2) Prompt Generation:***

***Upto now we have seen how to get and save the conversation details that we do inside our application. Now we are going to see how the selected conversation message is being passed to the LLM along with the user query as because this is our main objective to get meaningful responses only came by sending the user input and previous conversation message of the current chat together to get the better responses.***

***Prompt is something like the Instructions that we give to our LLMs to guide them about how it should behaves for the user’s questions. And the conversation message of the current conversation is binded within the prompt instructions. So the LLM can know what are the information that I know by using the conversation history and what do i need to do based on the instructions given.***

******

***This code block is where our prompts are playing a role while instructing our LLMs to what do to and how the previous conversation is going on by passing the conversation message into the prompt.***

***From the above code you can see that based on the categories (appointment, medication reminder, refill, referral) we frame the prompt by add the conversation history using the memory.load\_memory\_variables() method and while calling the llm by running the chain at the last line of the above code snippet, We are sending user input and prompt message with conversation history to our LLM.***

***2.1.3) LLM(Model: gpt-4o-mini) & Tools (RAG):***

***Upto this stage we got our conversation history and injected the same into our prompts along with the instructions. Now we are ready with our input to our LLM to receive a response for the same. To do that first of all we need the a LLM and tools that the LLM can execute based on the user query. After getting these two we need to buid the Chain which is combine all our components into one.***

***2.1.3.1) LLM setup:***

***For the LLM we are currently using the Open AI’s “gpt-4o-mini” model. For that langchain comes with a package called “lanchain\_openai” which provide a class to access the LLM of OpenAI.***

*from langchain\_openai import ChatOpenAI*

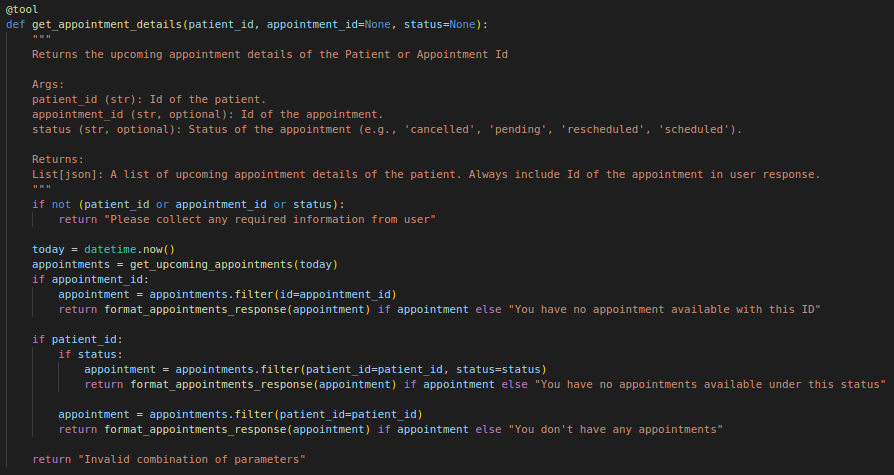
llm = ChatOpenAI(model='gpt-4o-mini', openai\_api\_key = get\_openai\_key(), temperature=0.2)

***By using the above two lines you can simply built your LLM. And to call the llm for getting the response we can call the invoke() method of the ChatOpenAI class. The implementation can be found below.***

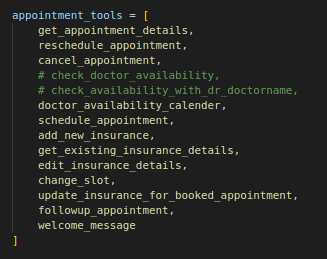
*response = llm.invoke("Hi!, How are you doing?")*

***2.1.3.2) Tools Building:***

***Tools are simply python functions that is being binded with the LLM so that the LLM can use these functions whenever it’s necessary while responding to a user query. As we are operating at 4 core healthcare workflows including appointment booking, medication reminder, refill management and referral management the possible tools like checking the doctor’s availbility from the database, booking a appointment with doctor for specific date and time. Here I list show some examples for tools and how we bind the tools with the LLM so that they can know the tools functionality and can call relavant tools based on the user query.***

******

***This is a example for a tool, and there are several ways to convert a python function to tool it’s completely based on the LLM we are using, as we are using OpenAI as our LLM. We simply import the @tool decorator for making our python function as tool and our function should consist the docstring as it’s crucial for the LLM to understand the behaviour of the tool so that the LLM can call the relevant tool based on the user query. Now I will provide the code snippet to bind the tools with the LLM.***

******

******

***By doing the above code we bind our tools with the LLMs.***

***2.1.4) Tool Executor:***

***Next up we have to create a tool executor which has the capabilities to execute the tool that is being dictated by the LLM in it’s response as LLM alone can’t have the ability to execute the tool. That’s why we are creating a additional tool executor along with the tools we have to execute the tools as per LLM response.***

***The code block for Tool Executor will be found below, which is provided to us by langgraph packages.***

*from langgraph.prebuilt import ToolExecutor, ToolInvocation*

tool\_executor = ToolExecutor(appointment\_tools + prescription\_tools + nurse\_tools + refill\_tools +referal\_tools)

***This code build a tool executor which gathers all available tools.***

***2.2) Chain Building & Calling the Chain:***

***Until now we have seen what are the components that are essential for building a LLM Chatbot. Now In this section, we are going to connect each components into one so that our Chatbot will work as we expected.***

***Langchain provides various ways to do this including LECL chains, legacy chains. Initially we build the chain with a LECL chains but due to some reasons. Be move into something that quite works like How LECL chains works.***

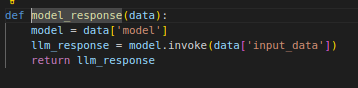
***Actually the way the LECL chains works is that, It will have a set of functions and each function is connected to one another with pipeline operator in python ( | ). And about the execution the first function receive the input and return the output, and then the first function’s output is passed as the second function’s input. By following the above pattern each function passes the it’s output to input as a next function to complete the chain and return response to the user.***

***We also did the same to build our chain, by writing four funtions to handle each functionalities and running them on a series to pass each functions output to next function’s input to replicate the behaviour of LECL chains. First of let me break down what are these four functions and what each of them do. Then we move into the Chain Building and Calling the Chain.***

***2.2.1) Four Functions:***

***2.2.1.1) model\_response() function:***

***This function is responsible for generating response to the user query. And it accepts the input data which is a combination of both user query and system prompt (instructions and conversation history) and the LLM which is being called to generate the output. The output can be a general response from the LLM or It can call a tool based on the user query. In the next step we extract the tool’s name and it’s parameter from the model’s response.***

******

***As seen in the above code, the user input and model is feeded to the function and the model generates the response by using the invoke() method. And about the function impleementation refer the below code.***

*user\_message = [SystemMessage(content =[{"type": "text", "text": prompt}]),HumanMessage(content=[{"type": "image\_url", "image\_url":{"url": image\_input }}])] if image\_input else [SystemMessage(content =[{"type": "text", "text": prompt}]),HumanMessage(content=[{"type": "text", "text": text\_input }])]*

model = llm.bind\_tools(nurse\_tools)

input\_data = {"input\_data": user\_message, "model": model}

***This is the kind of code that is passed as a input for the model\_response() function.***

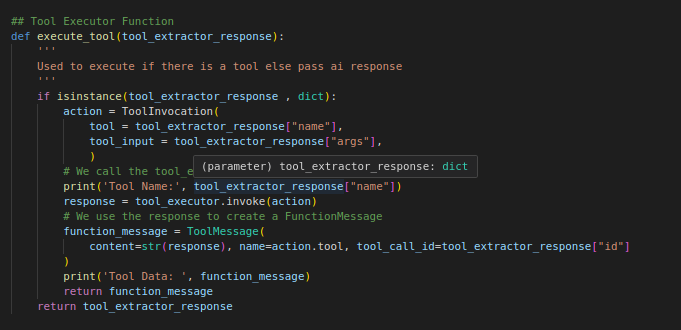
***2.2.1.2) extract\_tool() function:***

***This function is responsible for extracting the tool data including it’s name and it’s arguments from the model\_response() functions output if there is any tool call happened in the response else it returns the general response to the next function.***

******

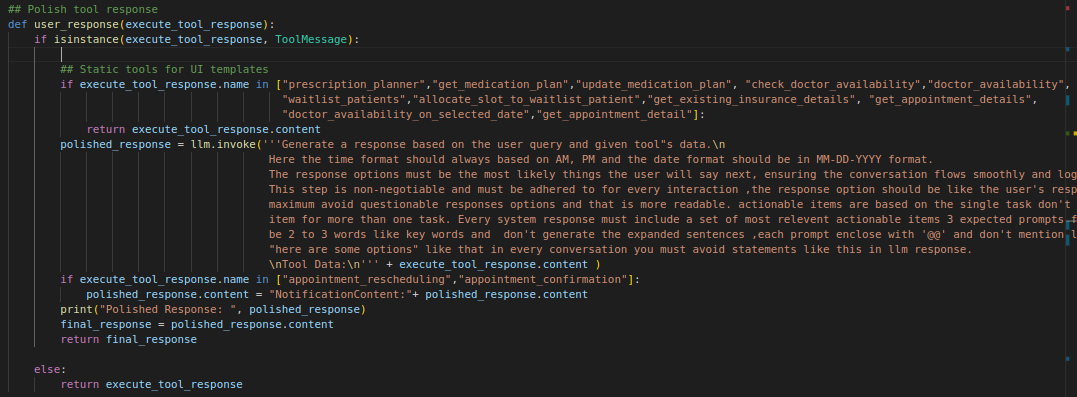
***2.2.1.3) execute\_tool() function:***

***This function is responsible for executing the extracted tool by using it’s name and arguments from the previous function and return the tool’s data as response if there is any tool details are extracted in previous function else it return the general response to the next step.***

******

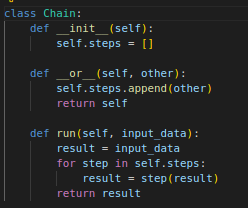
***2.2.1.4) user\_response() function:***

***This is the final function in our series of functions and is responsible for formatting the tool’s data into meaningful responses. It receives the execute\_tool() function’s output as input, If it contains the tool data then it’s creates a meaningful information from the data or some tool datas can be directly feeded to users so that the UI use this to create Templates out of it. Or if the input is general response ( not tool data ) from the LLM. It returns the same without any intervention.***



***2.2.2) Chain Implementation:***

***Now we have our four functions here, Let see how can be build a chain out of it to make our chatbot to response to the users. As told earlier we simply combine these function to execute one by one by passing each function’s output to next function’s input and process the same to get the final response to the user to build the Chain() class.***

******

******

***The first code block is the definition of our chain class’s, which have special method run() which executes the steps / functions given during the object creation and it executes the function one by one to sucessfully generate the response to the user query.***

***Usage In Real Time:***

***The second code block is the implementation of our chain in real time chat API, as you can see there***

***1) first we craft the prompt by combining the conversation history & user input***

***2) Next up we bind our llm with the tools***

***3) Next up we form our chain by giving the steps / functions required for our chatbot***

***4) Next up we call the run() method of our chain to return the response.***

***5) Next we process the output, stores the human and ai conversation pair to database and return the respone to user***

***This is how the LLM part works in our second approach. However this approach also have some limitations including one of the significant limitation is that it does have “multiple tool calling” capabilities. To resolve the issue we did moved to our new approach which is implementing Agents to handle user queries effectively and has multiple tool calling capabilities.***

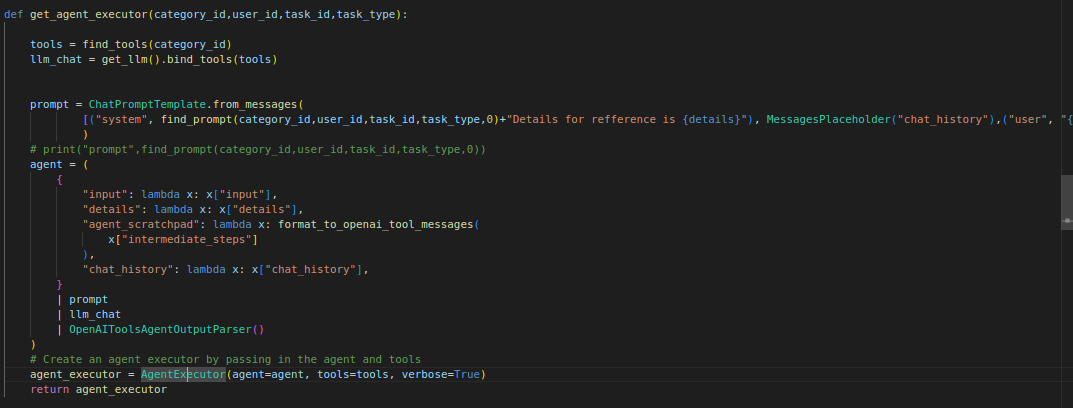
***As most of the components that we used here including conversation history, prompt generation, llm & tool building, building the chatbot are common across all implementations. So that in our next approach I just move through the Agent’s architecture and implementation and how it is used in real time.***

***3. LLM Integration using Langchain’s Agent.***

***As the components requires for building the Agent remains the same as our previous approaches the thing is going to change here is how we are putting all the components together to make our chatbot more efficient obviously with multiple tool calling capabilities.***

***3.1) Agent Definition:***

***Let me quickly define the function that combines the all components required for our Agent as return the Agent itself as it’s response. The Agent is nothing but a architectural implementation of code which uses Chain class that we previously used in behind it’s implementations.***

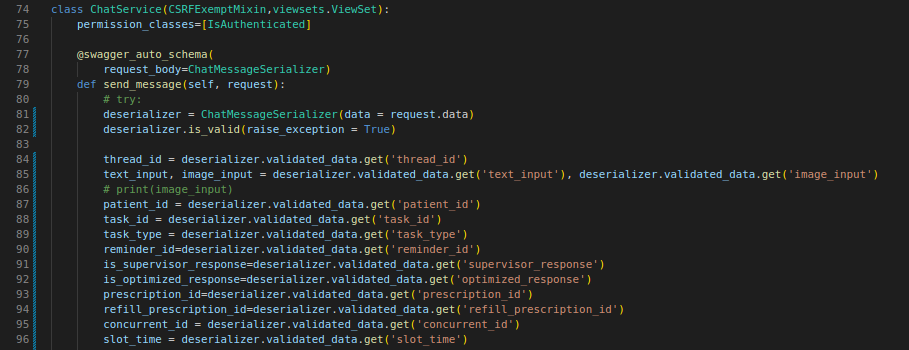
******

***As you can see above, you will find the components which are previously familier to you including tools, prompts, llms, and llm-tool integration. And after getting the components we building a LECL chain implementation and store this into a variable called “agent” and we passing the agent (basically a chain) and tools into our “AgentExecutor” class go form the agent executor. And for the agent executor we doesn’t need any external tool executor as the AgentExecutor takes care for the same internally and have multiple tool calling capabilities.***

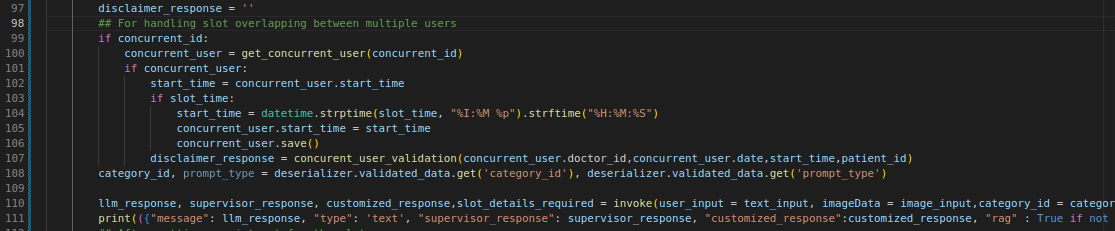
***And until now we implemented our agent from the above code. Now lets see how it’s implemented in realtime inside our chat API’s invoke() function by explaining the code snippet. For this sake, Let me just go through the sendmessage API which is the core API that handles the conversation inside our application.***

***3.2) AgentExecutor Usage In Real Time (send message API):***

***Let me break down each of code block’s inside the sendmessage API and the main invoke() function in which our AgentExecutor is implemented.***

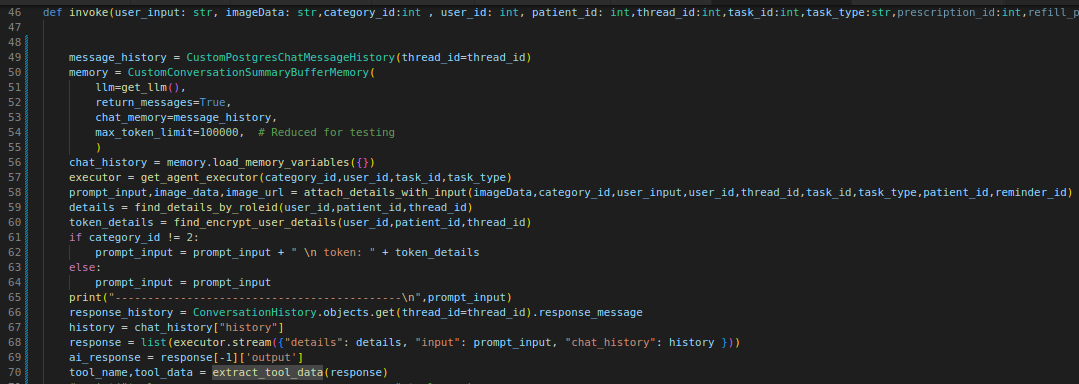


***Until now the send messge API is just storing the payload details that the client gave to us via serializers.***



***In this part first we are handling a scenario, where a specific slot of a doctor on specific date & time is choosen by multiple users and this code block is specifically responsible for generating the disclaimer message about the current slot’s status to remind the users. And secondly you can see that we called the invoke() function from which we receive a response from our bot. So let see what’s happening inside the code block.***

***The invoke() function:***



***This is the function that is the core of LLM response generation. Let me explain what is being done in the above code snippet one by one.***

***1) First off all, we are using our two classes that are meant for get conversation history and we are getting the conversation message for the given thread id.***

***2) After getting the conversation message, we are sending them into a function called get\_agent\_executor() function which builds the agent by combining the components as we seen in the Agent definition part.***

***3) After builded the agent, we are getting the user details and the user’s workflow categories (appointment, refill..) based details by calling the relevant functions.***

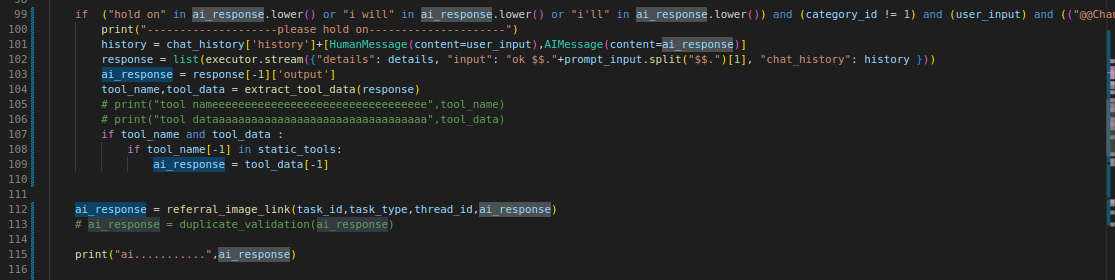
***4) After getting all the required things, we are calling the stream() method of our AgentExecutor by passing the user input, user’s details and their relevant workflow details to get the relevant response from the LLM agent.***

***5) After getting the responses from the LLM, we are checking that if there is any tool calls are happened inside the LLM response by using extract\_tool\_data() function.***

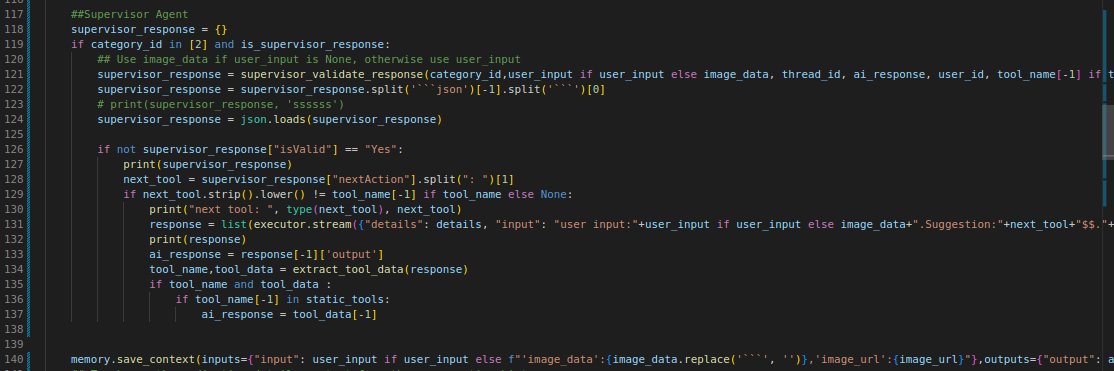
***Next up we will see what are the processes that we do before sending the response to the user, and also how we tackle the inaccurate responses from LLM.***

******

***In this code block, first we are applying transformations to the tool data based on the category they are from. In the next code block we prevent that the current response is not as same as the previous response, to acheive this we check the current llm response with previous llm response to check that the response is not looping over time. The solution to this problem is that we calling the llm again to generate the response to the user query again.***

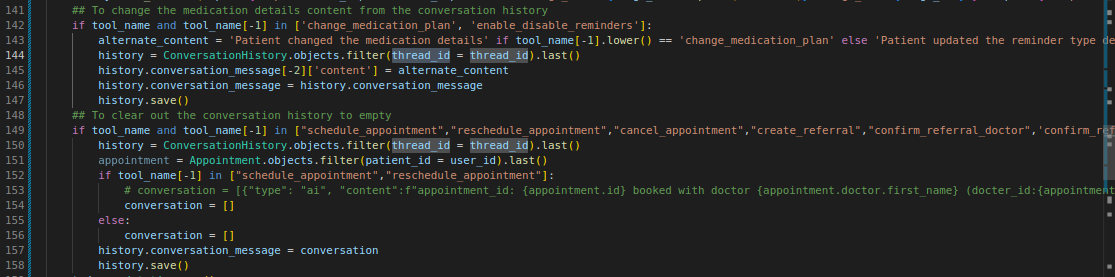


***This code block is to prevent the stopper responses that the LLM make like hold on for a moment, i will do the same, which makes gap inside our healthcare workflows. The solution to this problem is again that is implemented above calling the llm one more time.***

******

***This code block is used to implement a supervisor agent, to verify the llm responses are going on the track based on the current categories’s workflow by using another agent which is act as a supervisot for our previous agent.***

***After that we are saving the current input & output pairs to the databases by using the save\_context() method of our ConversationSummaryBufferMemory Class which is using BaseChatMessageHistory Class’s add\_message() method.***

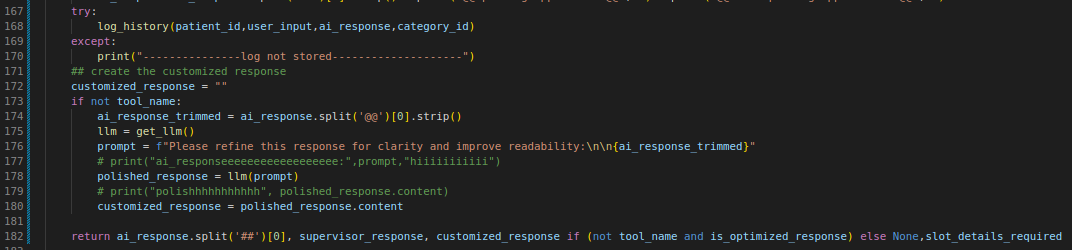


***In this code block after saving the response message to the databases, We change some of their content for example***

***1. While calling specific tools in the medication reminder we didn’t write the tool data directly to the history, instead we use some alternate content for them.***

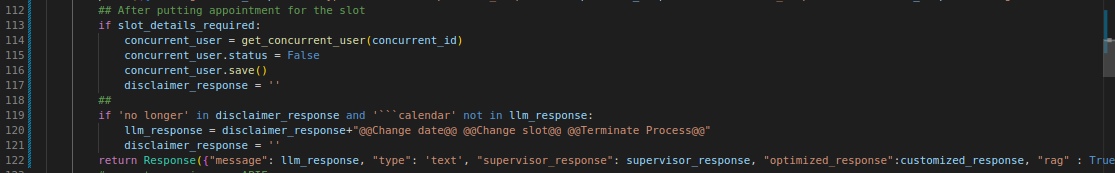
***2. While scheduled an appointment and rescheduled an appointment we just clearing out the conversation history to empty list to avoid irrelevant responses and response looping.***

***After this block we log the conversation and handle where there is no tools called inside our llm responses and return it’s the llm response as output. You can find the code for the same below.***



***This code block is responsible for logging each and every conversation and handle the llm response when there is no tool is called and return the same as it’s output.***

***Now we have gone through the invoke() function, then we will be resuming where we left off inside our API.***



***After getting the response from the function, now we are handling the concurrent slot picking scenario by checking whether the specified slot is closed by booking an appointment so that we make that specific slot free for other users.***

***So after that we sending the response for the sendmessage API to the users.***