# **Lab 3: Now We're Flying**

With the critical components in place, we're ready to tie everything into the chat interface. When a user types a question into the chat interface, we need to create a vector embedding for the question, then search for the most similar vector embeddings for products and accounts, and return the relevant documents that get sent to Azure OpenAI's completions endpoint.

In order to return a human-friendly response to the user, we need to use the completions endpoint to generate a response based on the most relevant documents and an instructional system-level prompt. Furthermore, we need to keep a history of the user's questions and the responses that were generated so that they can reload the chat in the future.

To generate prompts for the Azure OpenAI service, the approach is to use a technique called prompt engineering to author prompts that are used to guide the generation of completions. Prompt engineering is an iterative process that involves authoring prompts, generating completions, and evaluating the results.

The starter solution uses Semantic Kernel to orchestrate the execution of prompts. This lab is about experimenting with system prompts to impact how the completions work.

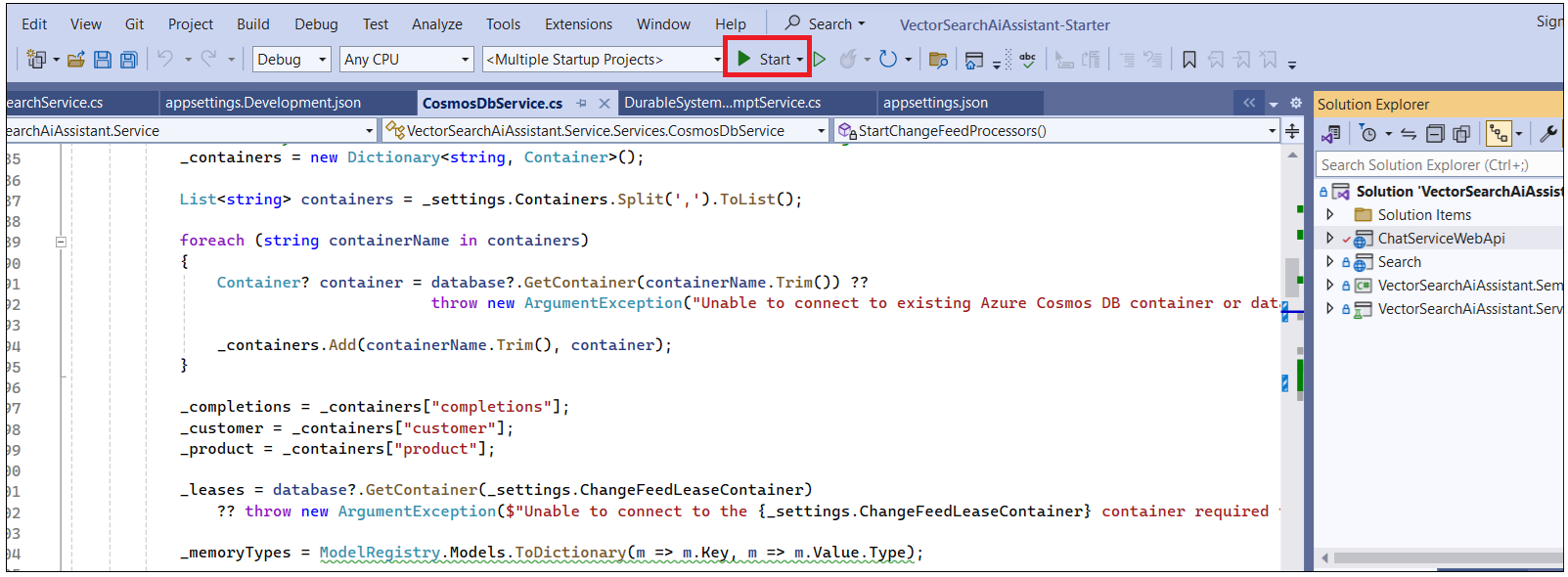
## Objective :

* Complete the chat interface modules to –
* Generate embeddings for user messages.
* Look up the vector index to retrieve the most relevant documents.
* Construct the System message, context, and completion.
* Store the chat history in CosmosDB
* Experiment with different Prompts to improve the performance.

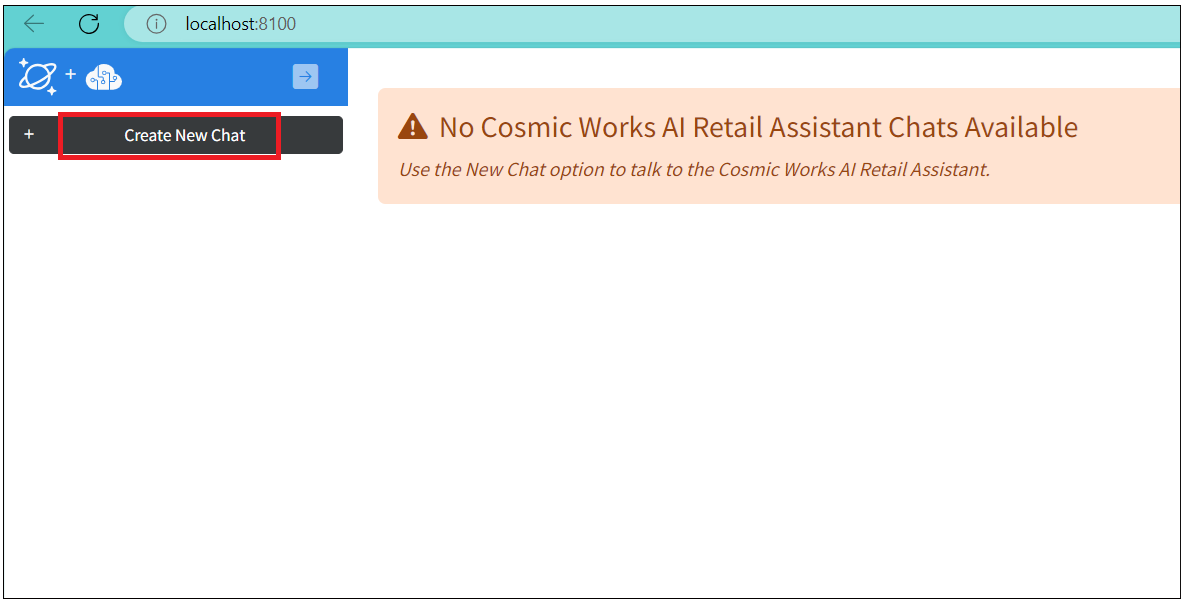
## Task 1: Build the memories section of the prompt and the System Messages.

.

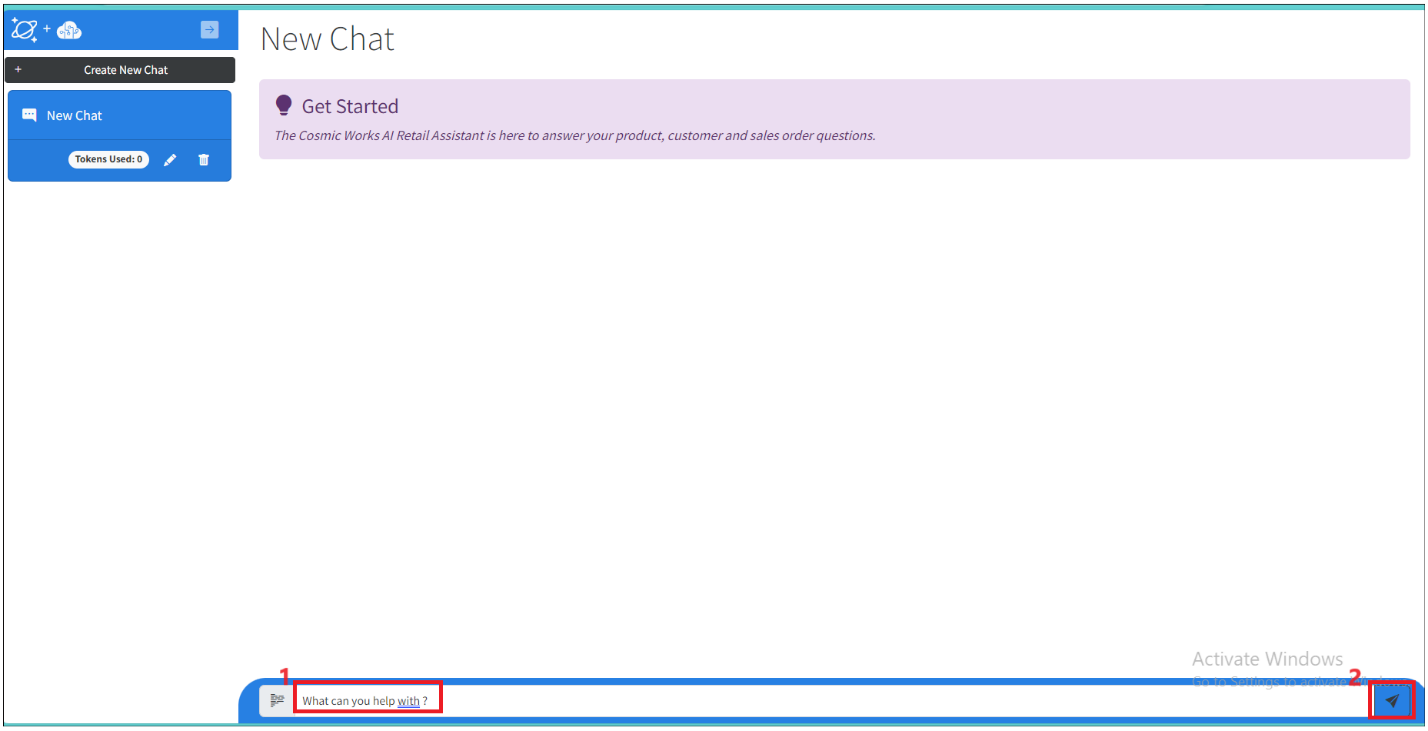
1. Switch back to **Visual Studio 2022** and build the solution by clicking on **Start** if your interface is not running.



1. Once the solution is built successfully, open an interface and click on **Create New chat.**



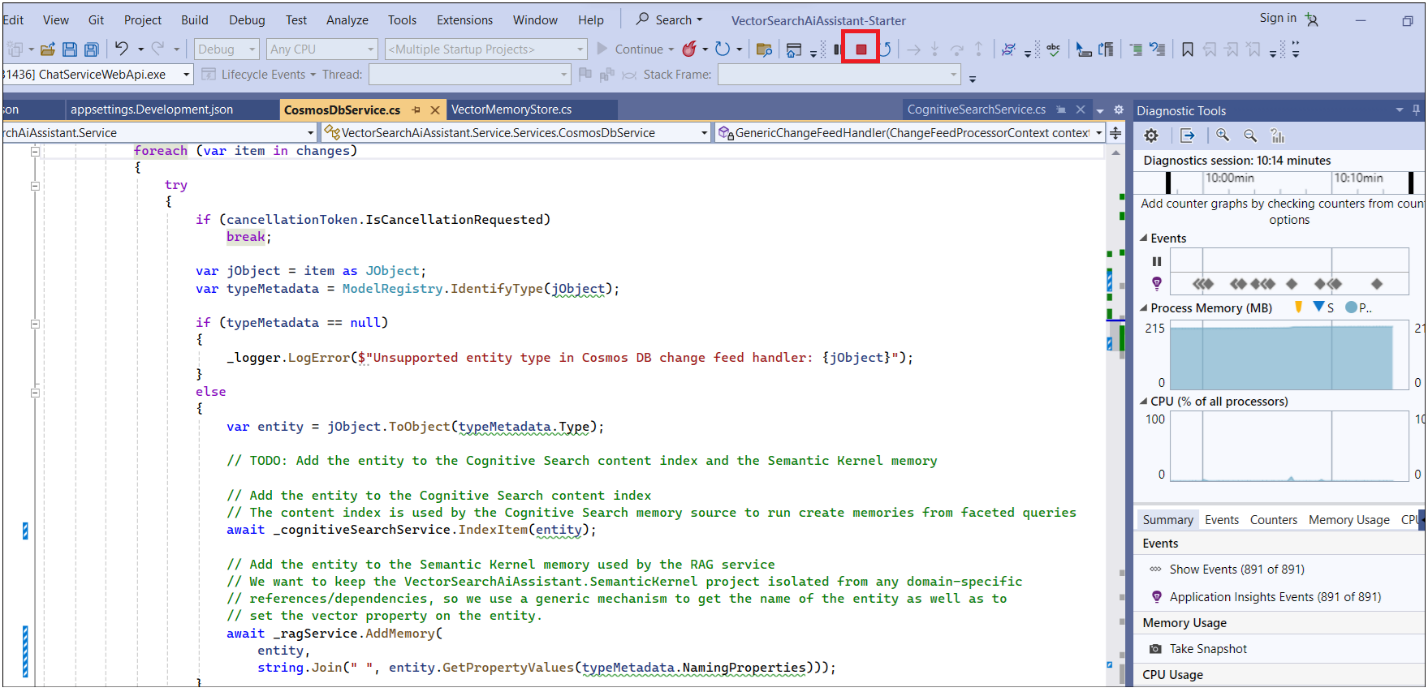
1. Create a new chat by asking questions **What can you help with?**. You can see that the question is posted, however, the interface does not show anything and also **Tokens** show **zero**.



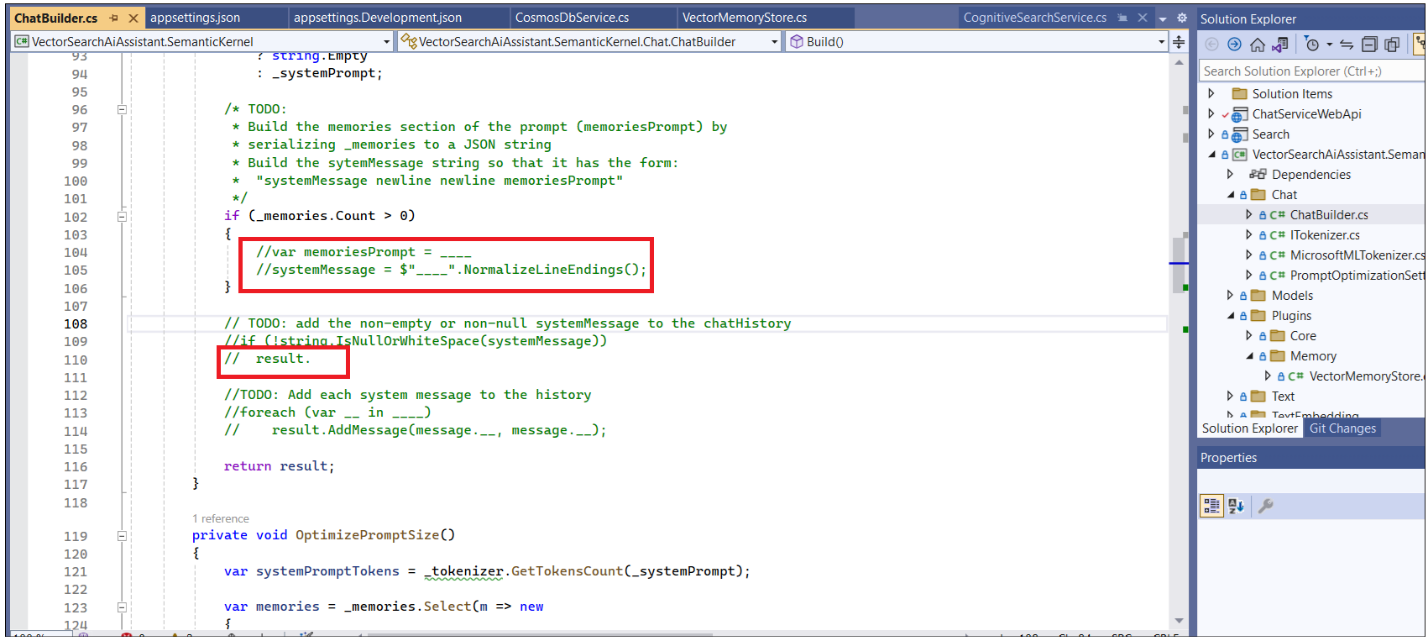
A screenshot of a chat

Description automatically generated

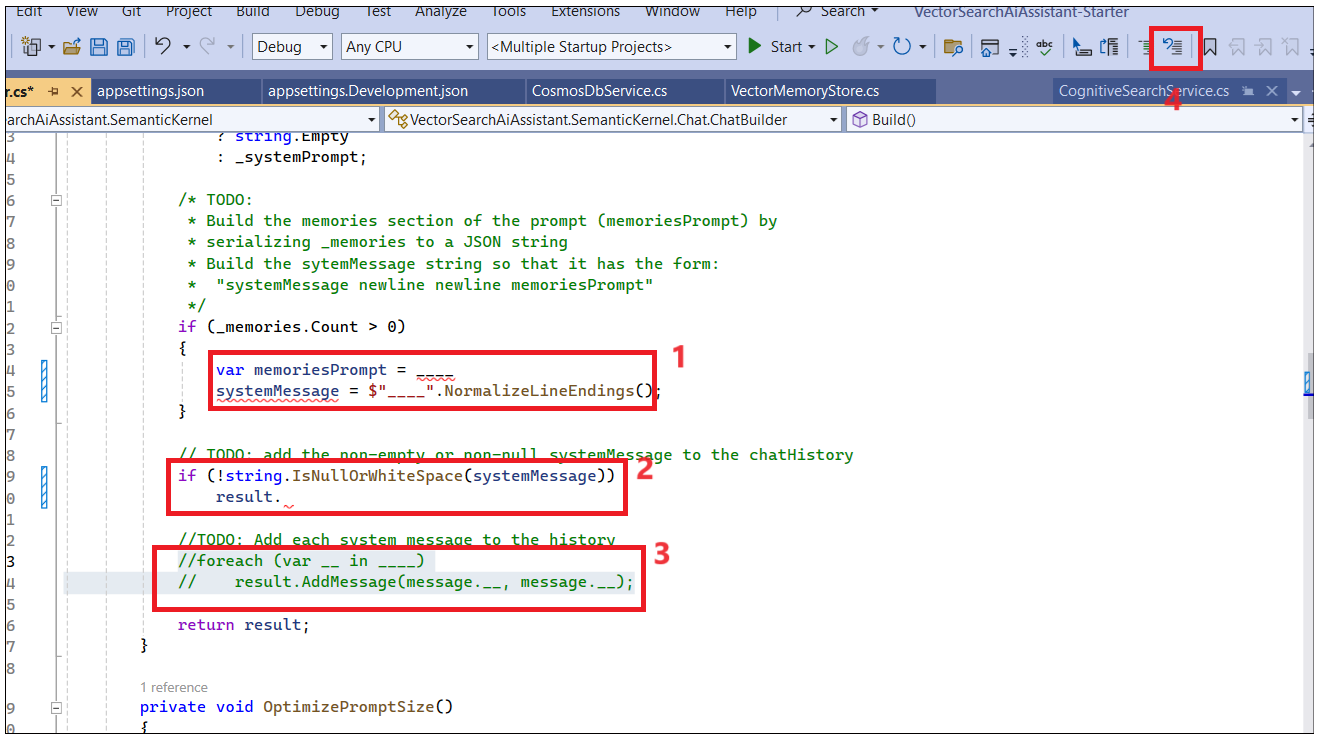
1. We will make changes to the stitch process together to be able to show the responses, and questions posted by users as well as store the history in the CosmosDB completion container.
2. **Stop** the solution to make code changes.



1. Open **ChatBuilder.cs** file from **VectorSearchAiAssistant.SemanticKernel-> Chat.** We will modify the code to make sure **memoryspormpt** and **system message** are in correct format as well as add the resulting messages into the responses.

****

1. Uncomment **memoriesPrompt** , **result** and **Addmessage** sections as shown in below image.

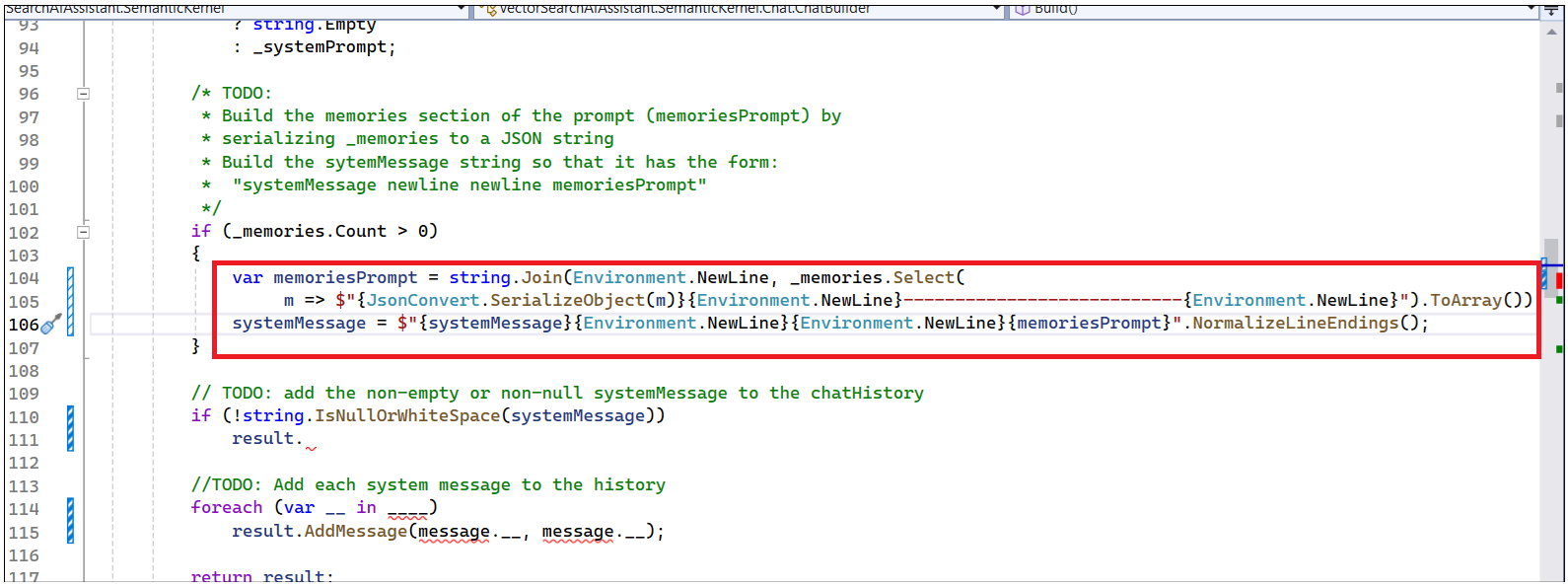
****

1. The **WithMemories** method populates the **\_memories** private property in ChatBuilder. It uses **EmbeddingUtility.Transform** to get the text representation of the entities in the memories.
2. The **Transform** method uses the **ModelRegistry** to determine the type of the entity and the **EmbeddingField** attribute to determine the prefix used when serializing the property value to **text**. The **ModelRegistry** is also used to determine the name of the entity, using the **NamingProperties** attribute.
3. **memoriesPrompt** is a simple concatenation of the JSON representations of the objects from **\_memories**.
4. Replace the memoriesPrompt and SystemMessage section with below code. both these lines of code specifying format of the message should look like by adding extra line characters. and necessary elements. SystemMessage also adding new line character after the System message to make sure its in the correct format.

var memoriesPrompt = string.Join(Environment.NewLine, \_memories.Select(

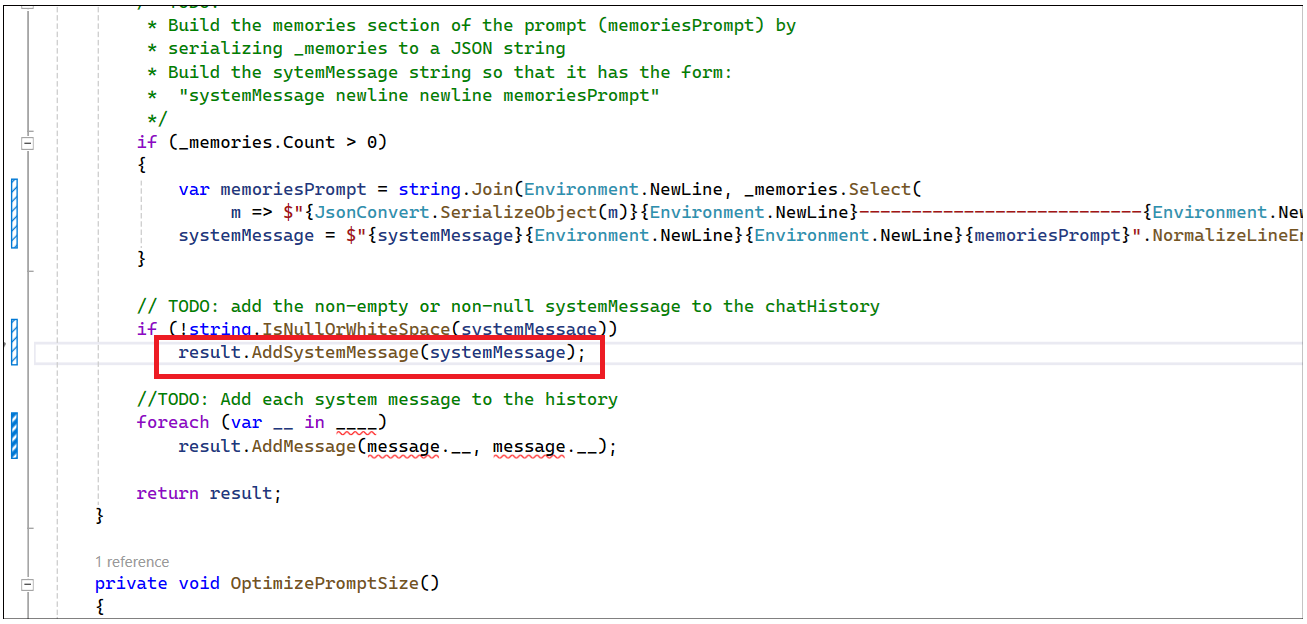
m => $"{JsonConvert.SerializeObject(m)}{Environment.NewLine}---------------------------{Environment.NewLine}").ToArray());

systemMessage = $"{systemMessage}{Environment.NewLine}{Environment.NewLine}{memoriesPrompt}".NormalizeLineEndings();

****

**4.** The **AddSystemMessage** method adds a system message to the chat. The message to be added is the system prompt stored in **\_systemPrompt** when **WithSystemPrompt** is called. Replace the result with below code with AddSystemMessage method and systemMessage variable as shown below.

result.AddSystemMessage(systemMessage);

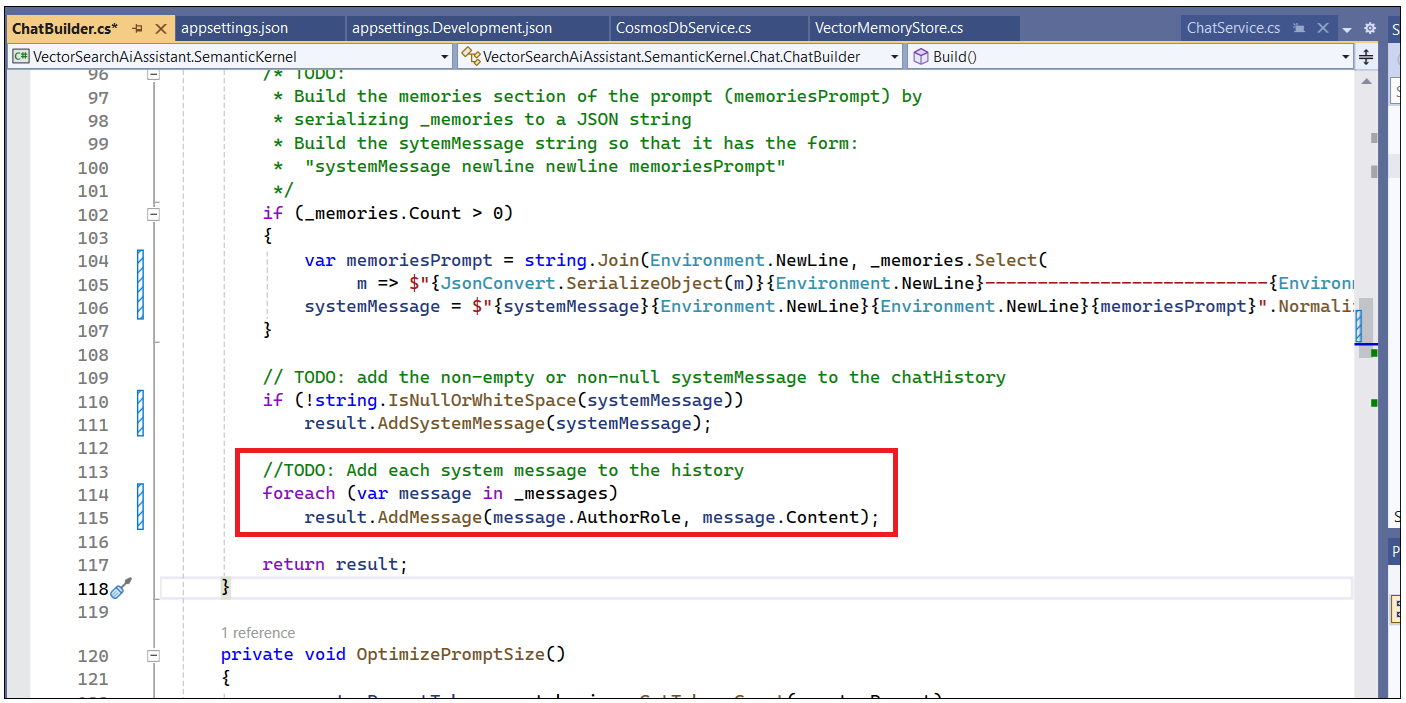
****

**5.** The **AddMessage** method adds the user message to the chat. The messages are stored in **\_messages** when **WithMessageHistory** is called. The **AuthorRole** property on each message is defined by Semantic Kernel and can be one of **system, assistant**, or **user**.

6. To Add each system message to the history**,** Update the code as shown below.

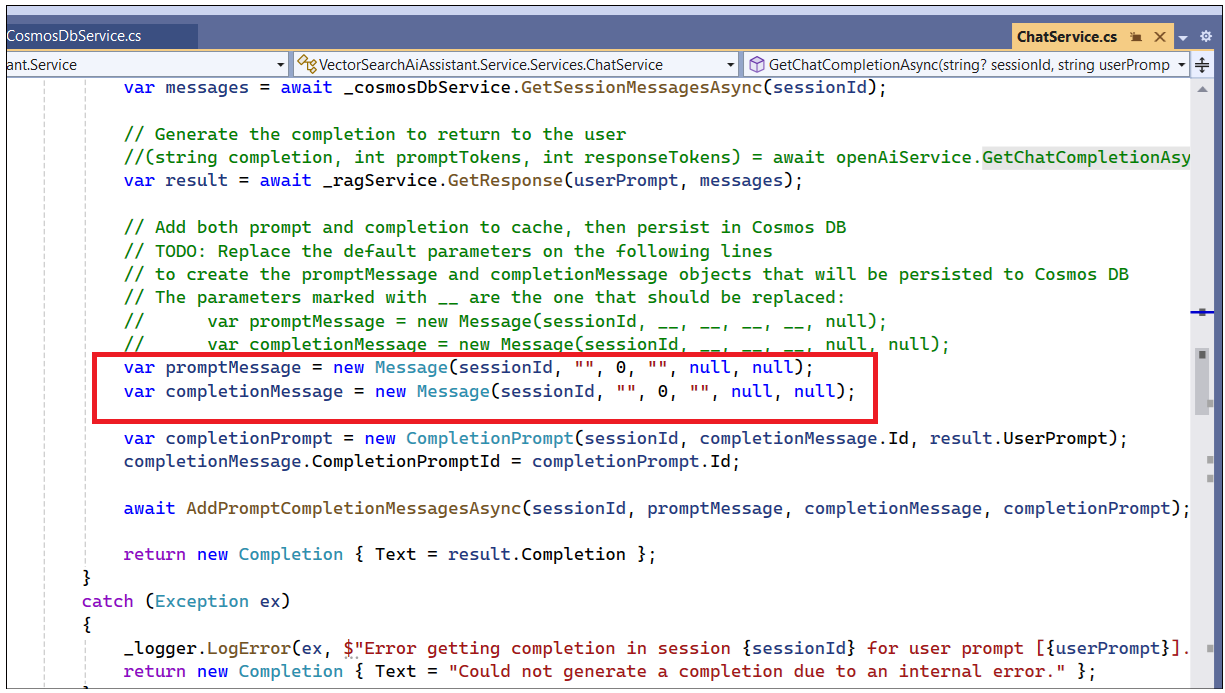
foreach (var message in \_messages)

result.AddMessage(message.AuthorRole, message.Content);

****

## Task 2: Receive a prompt from the user, vectorize it, and get a completion from the OpenAI service

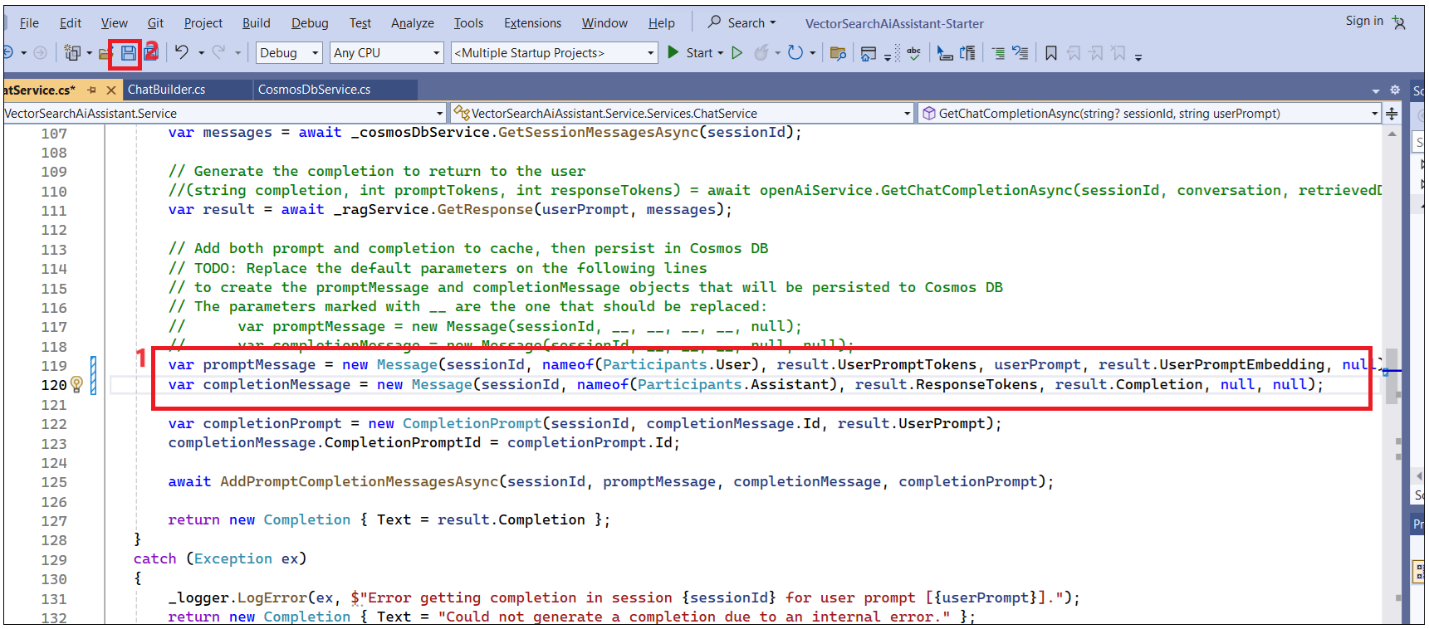
1. Open **ChatService.cs** file from **VectorSearchAiAssistant.Service->Services. ChatServices** file is used to implement the interactions in the chat interface like renaming, and getting chat sessions. deleting chat session. It also implements chat competition whenever user sends a prompt, this code invokes the RAG service to get a response by passing the user prompts and all the existing chat history
2. We need to modify promptMessage and completionMessage lines of code in GetChatCompletionAsync method. This line of code is used to store the user prompts and completions message in CosmosDB.



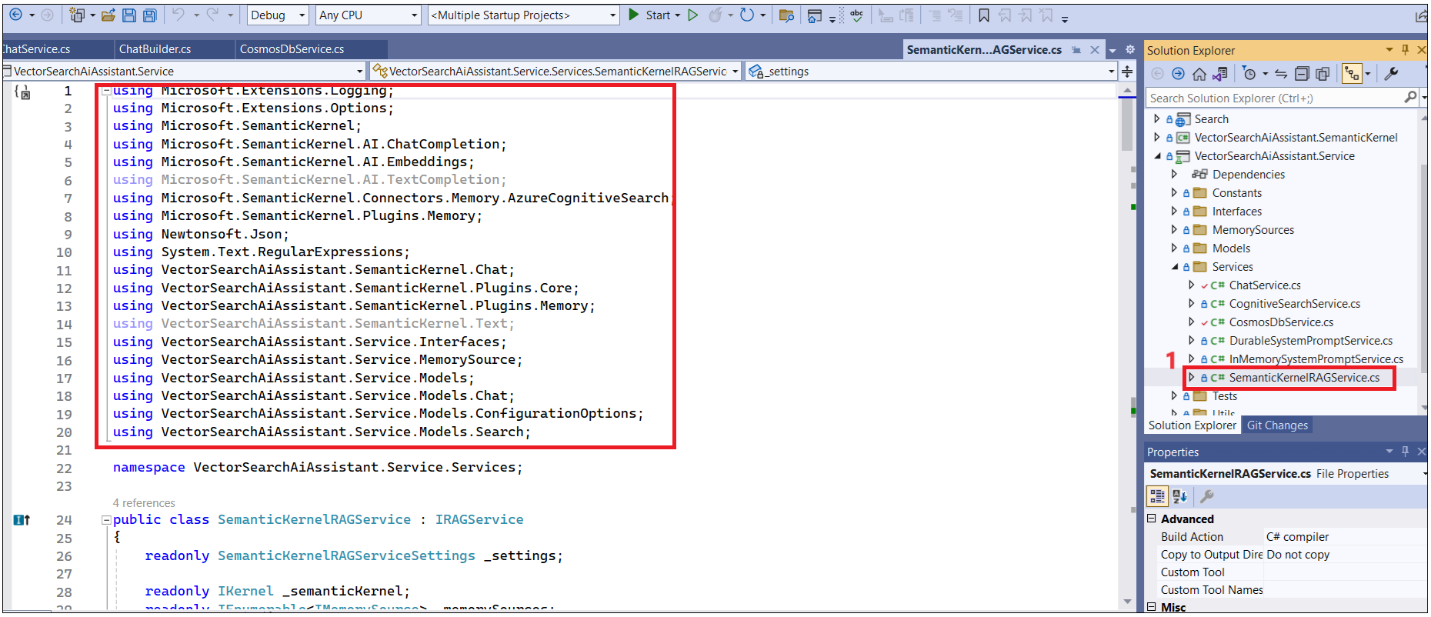
1. The prompt message should contain sessionId -to tie up all the conversation history of the user, the sender name (Participants. user), the number of user prompt tokens (result.UserPromptTokens), the actual text ( userPrompt ), and the embedding of the user prompt (result.UserPromptEmbedding).
2. We pass **promptMessage** and get completion from OpenAI service, we store it in completionMessage with sessionId, **assistant ,** ResponseTokens, **and** Completion
3. The completion message should contain the sender name (**assistant**), the number of response tokens (**result.ResponseTokens**), and the text of completion message.
4. Update the **promptMessage** and **completionMessage** with below code and **save** the file.

var promptMessage = new Message(sessionId, nameof(Participants.User), result.UserPromptTokens, userPrompt, result.UserPromptEmbedding, null);

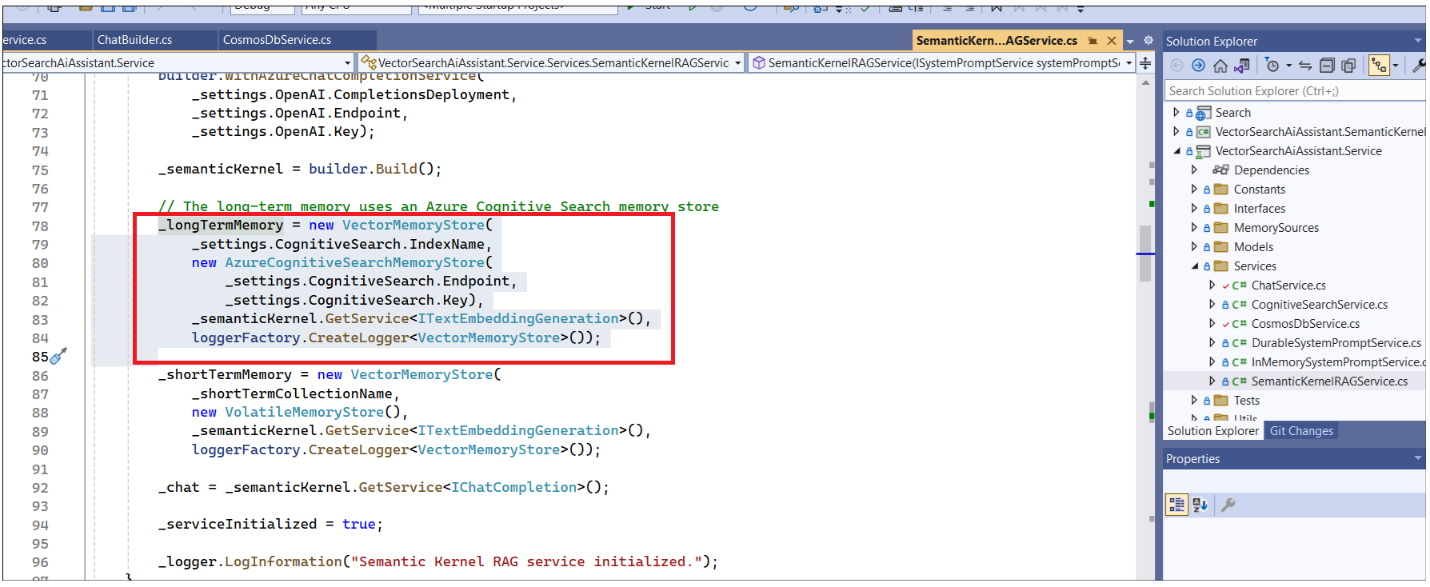
var completionMessage = new Message(sessionId, nameof(Participants.Assistant), result.ResponseTokens, result.Completion, null, null);



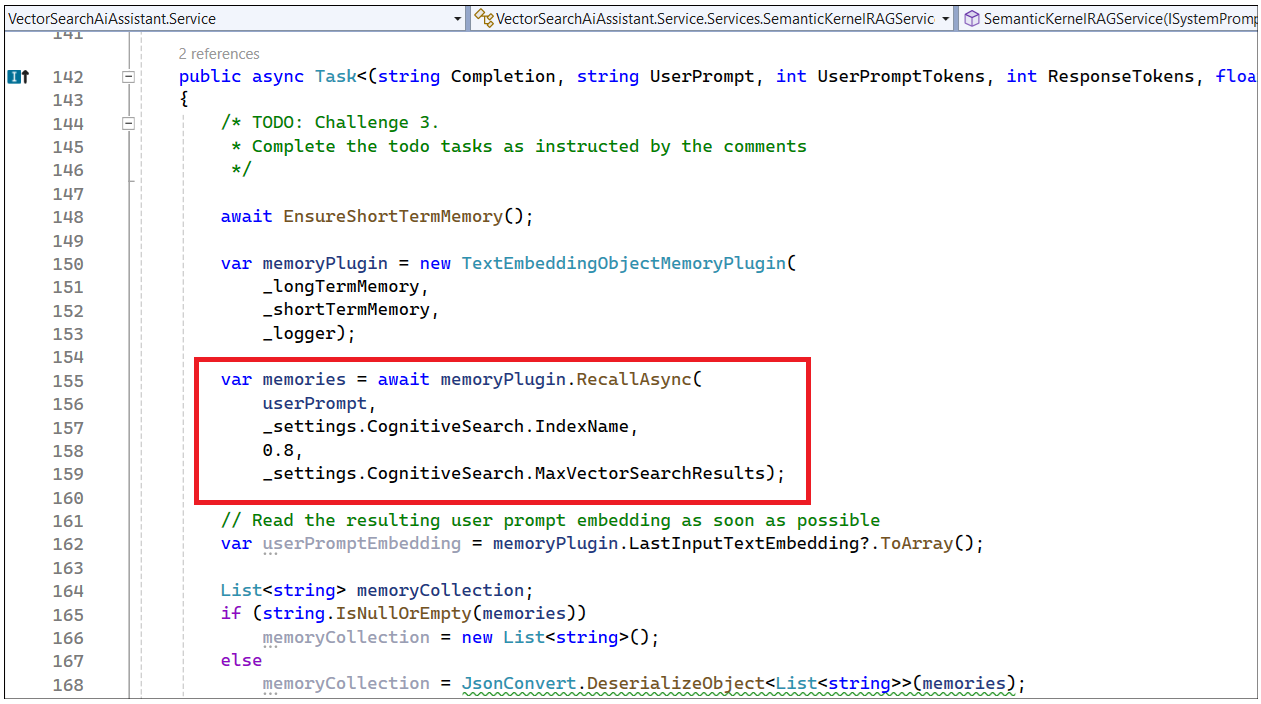
1. Open **SemanticKernelRAGService.cs** file from **VectorSearchAiAssistant.Service->Services.** We are using the functionalities of **SemanticKernel** to implement the RAG service.



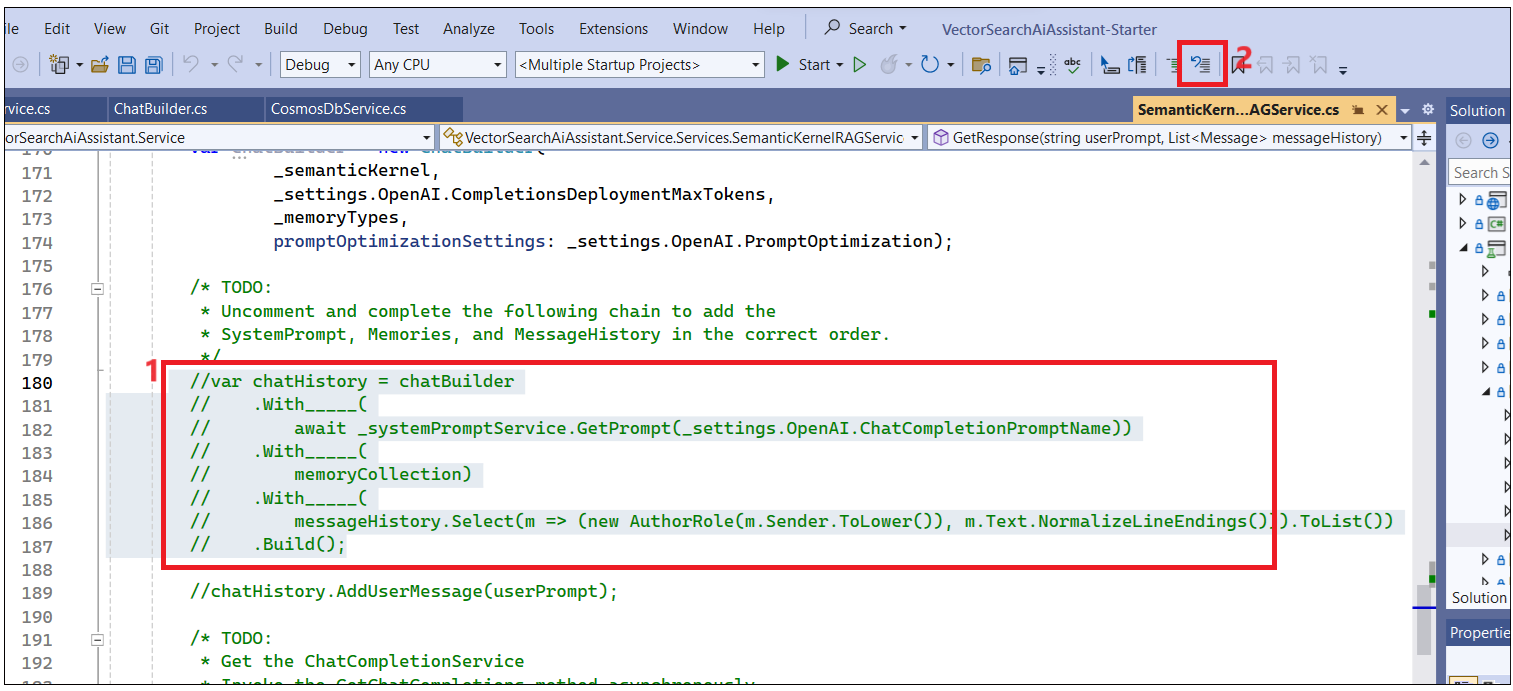
1. Scroll down and see the long-term memory uses an **Azure Cognitive Search memory** to store and retrieve documents to perform chat conversation.



1. Cognitive search index used to retrieve documents for context or memory and also specifying parameter to retrieve only the documents with relevance score of **>=0.8**



1. To implement RAG service functionality, uncomment **chatHistory** section as shown below.



1. To build ChatHistory, we need to update the code to get SystemPromptMessage , chathistory(Memories), and usermessages(responses from OpenAI services.)

A screenshot of a computer

Description automatically generated

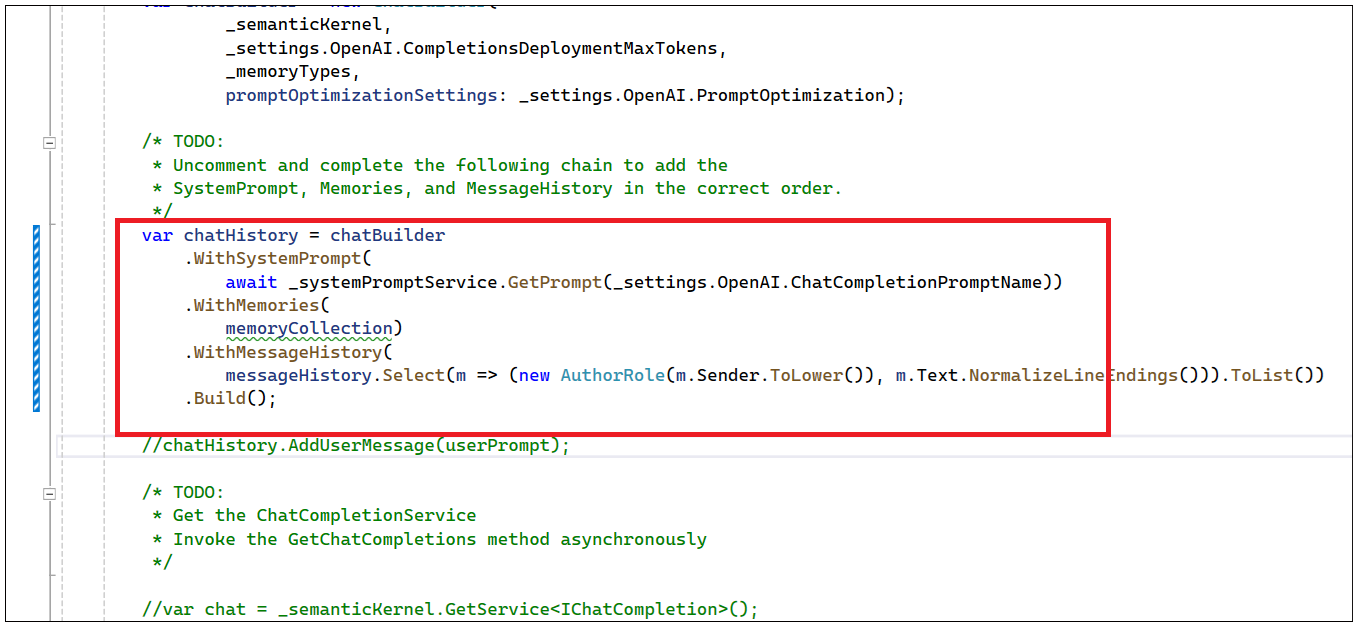
1. Update the code with below method.

* The WithSystemPrompt method sets the \_systemPrompt private property in ChatBuilde
* The WithMemories method sets the \_memories private property in ChatBuilder.
* The WithMessageHistory method sets the \_messages private property in ChatBuilder

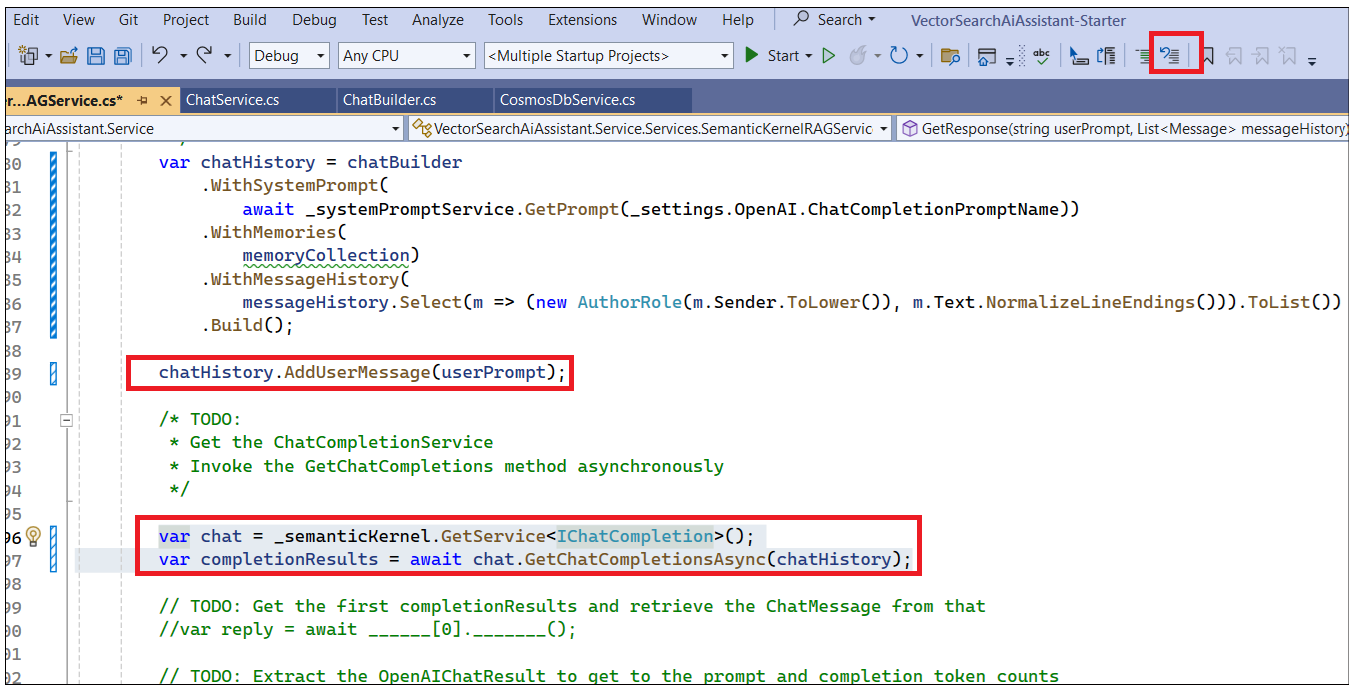
. WithSystemPrompt

.WithMemories

.WithMessageHistory

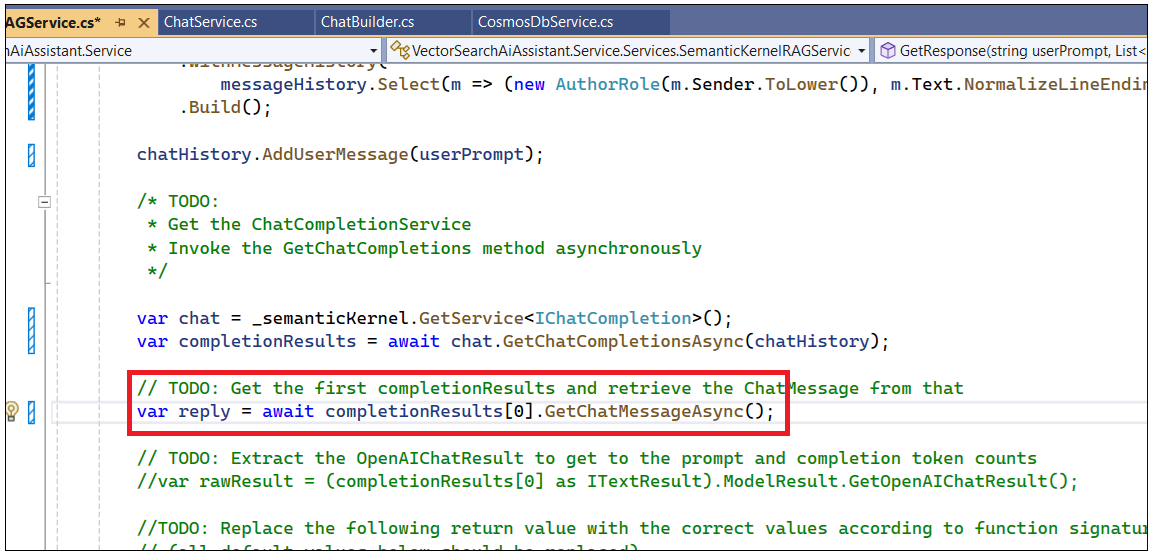


1. We need to add all messages as usermessage, and uncomment chatHistory.AddUserMessage and userPrompt.
2. Uncomment chat and completionResults variables. We use semanticKernel.GetService to perform chat completion and get the completion history into the completionResults variable.



1. A chat completion Semantic Kernel flow returns one or more completions. For this lab, we are only interested in the first completion, returned by completionResults[0].
2. Uncomment and update the below code to get the first completionResults (index[0] and retrieve the ChatMessage from that

var reply = await completionResults[0].GetChatMessageAsync();

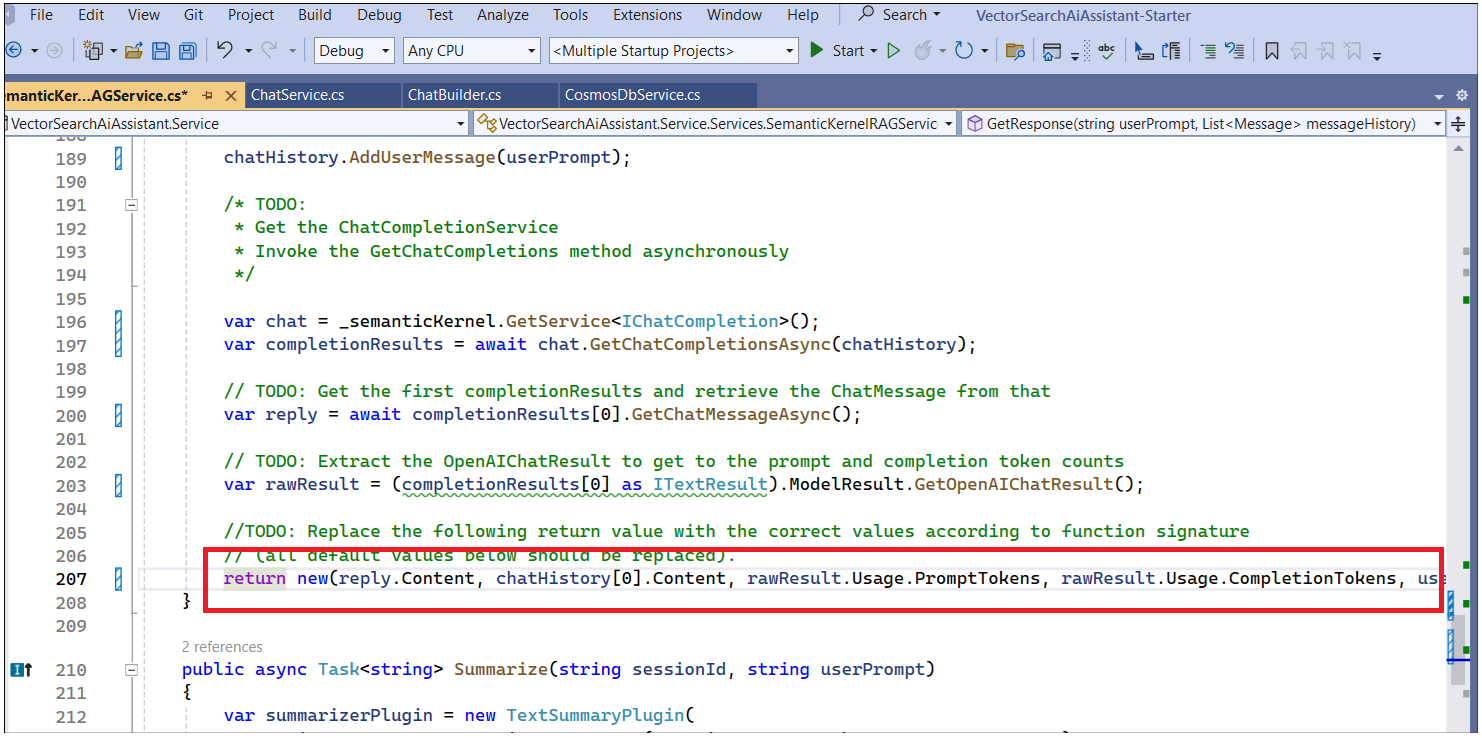


1. Get the latest **completionresults** into **rawResults** variable and use it in the return section as shown below. Uncomment **rawResults** section.

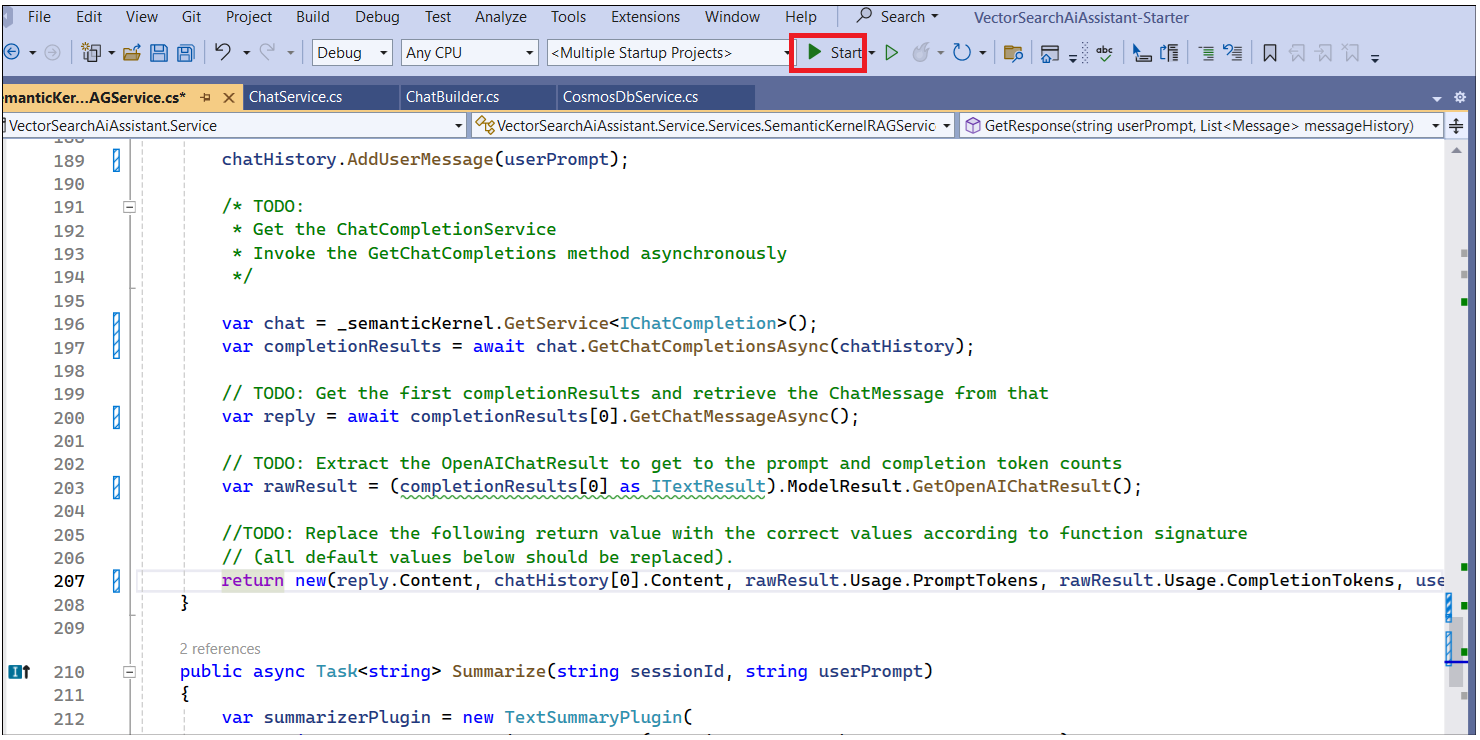


1. Replace the following **return value** with the correct values according to the function signature. This code returns the Content from a reply, previous chatHisotry messages, actual token usage of raw results, prompt token, completiontokens and Promptembeddings.

return new(reply.Content, chatHistory[0].Content, rawResult.Usage.PromptTokens, rawResult.Usage.CompletionTokens, userPromptEmbedding);



1. Click on **Start** and build a solution now to interact with chat interface.

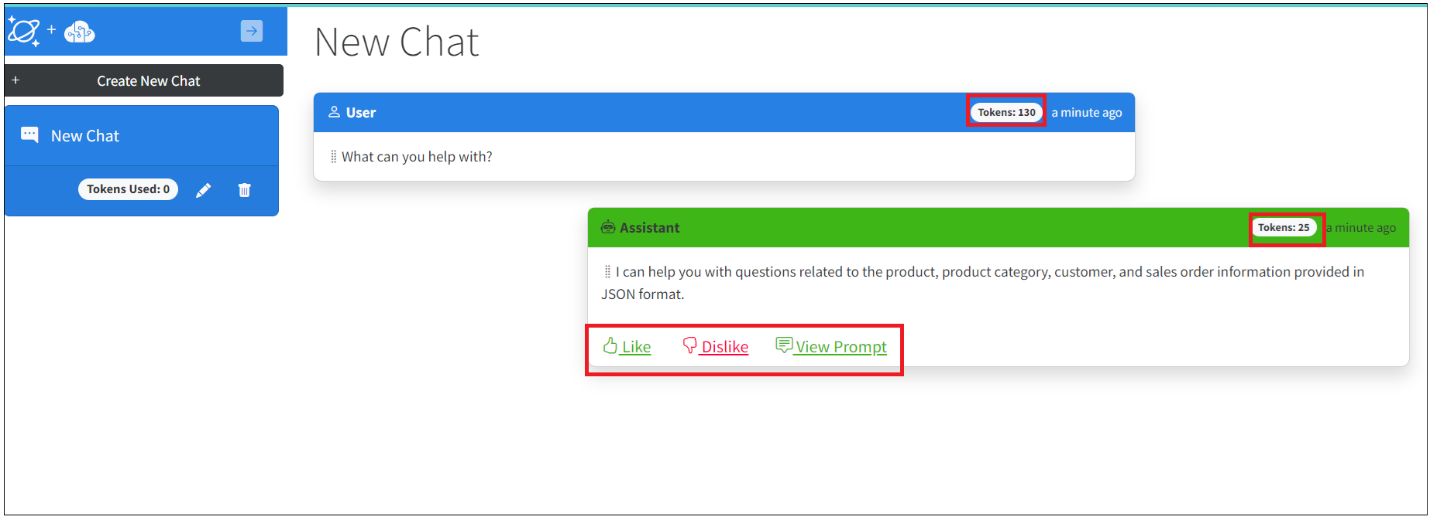


1. Go to the Chat Interface and **Create New Chat**. Enter the question **What can you help with?.** Its takes >1 min to respond**.**

A screenshot of a chat

Description automatically generated

1. The user prompt is the first message in the chat history. We can see User questions and responses from the Assistant. We also have an implementation to provide suggestions to improve responses by using the **Like/Dislike** button. We can also see tokes for prompt and response

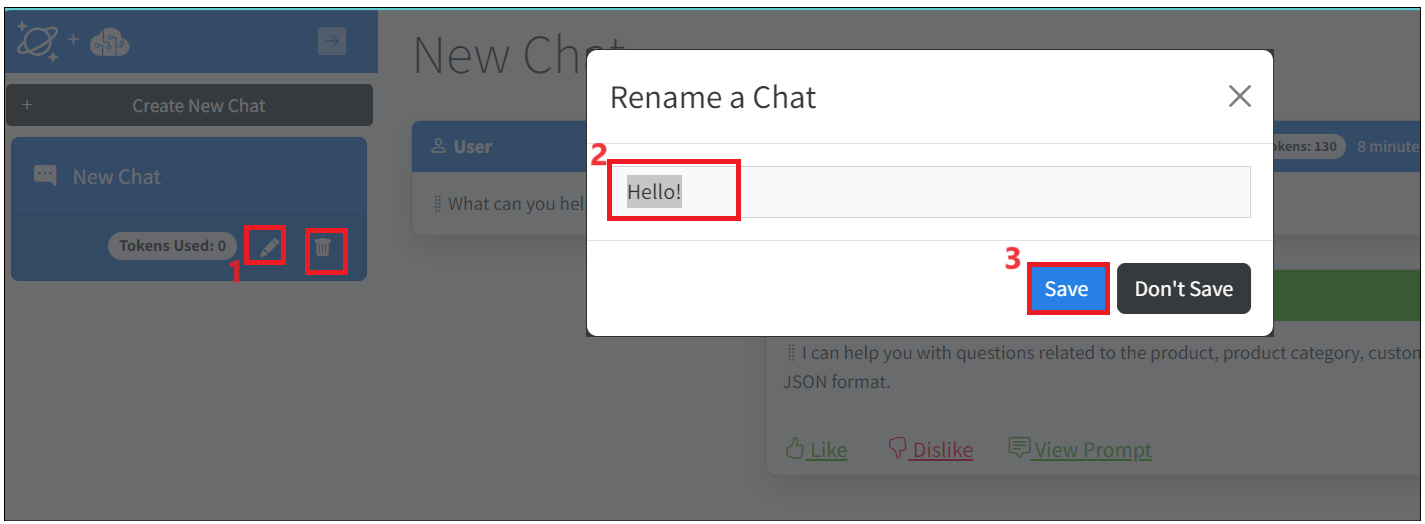


1. Click on **ViewPrompt** .

A close-up of a white card

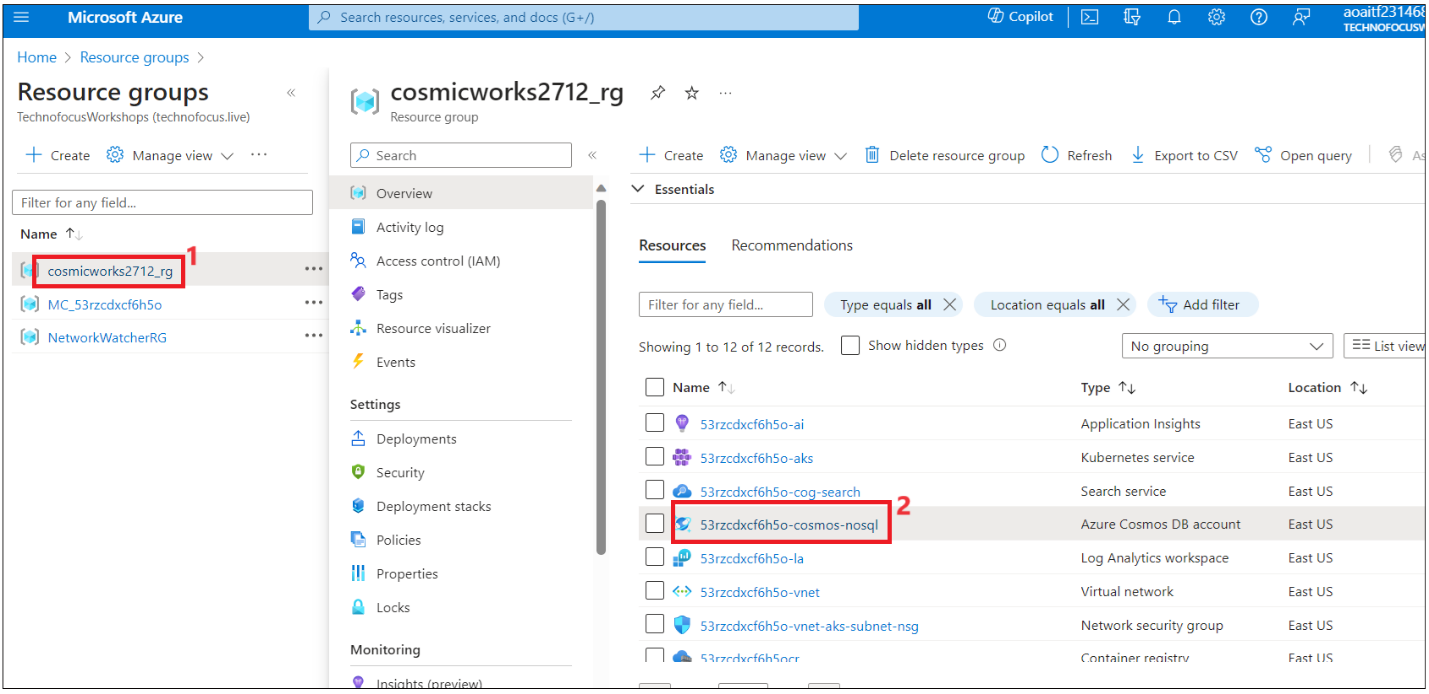
Description automatically generated

1. Click on **Edit** button in New chat and rename with ++**Hello!++**. We can also delete the chat by clicking on the **Delete** button.

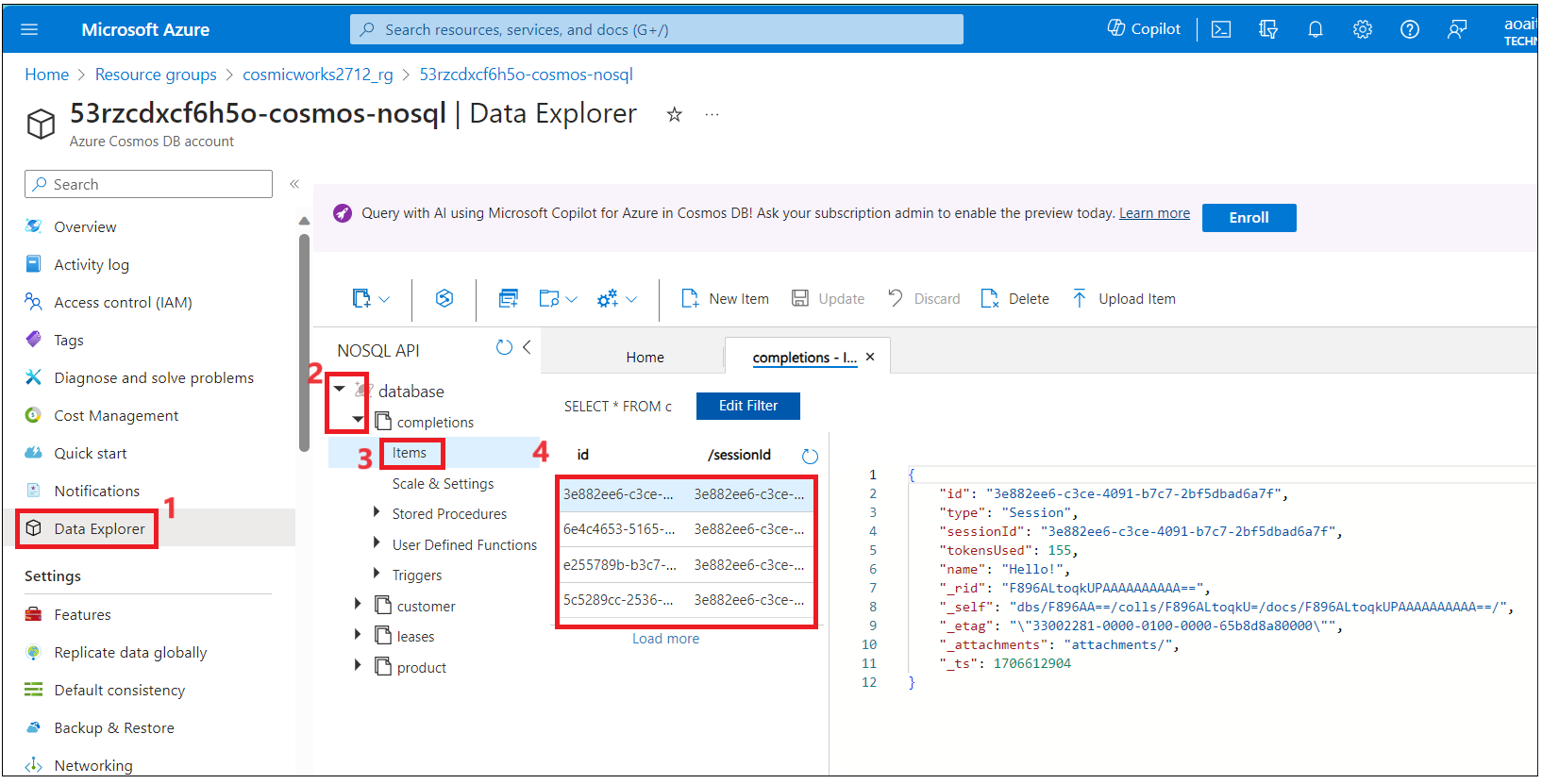


## Task 3: Store the Chat history in CosmosDB.

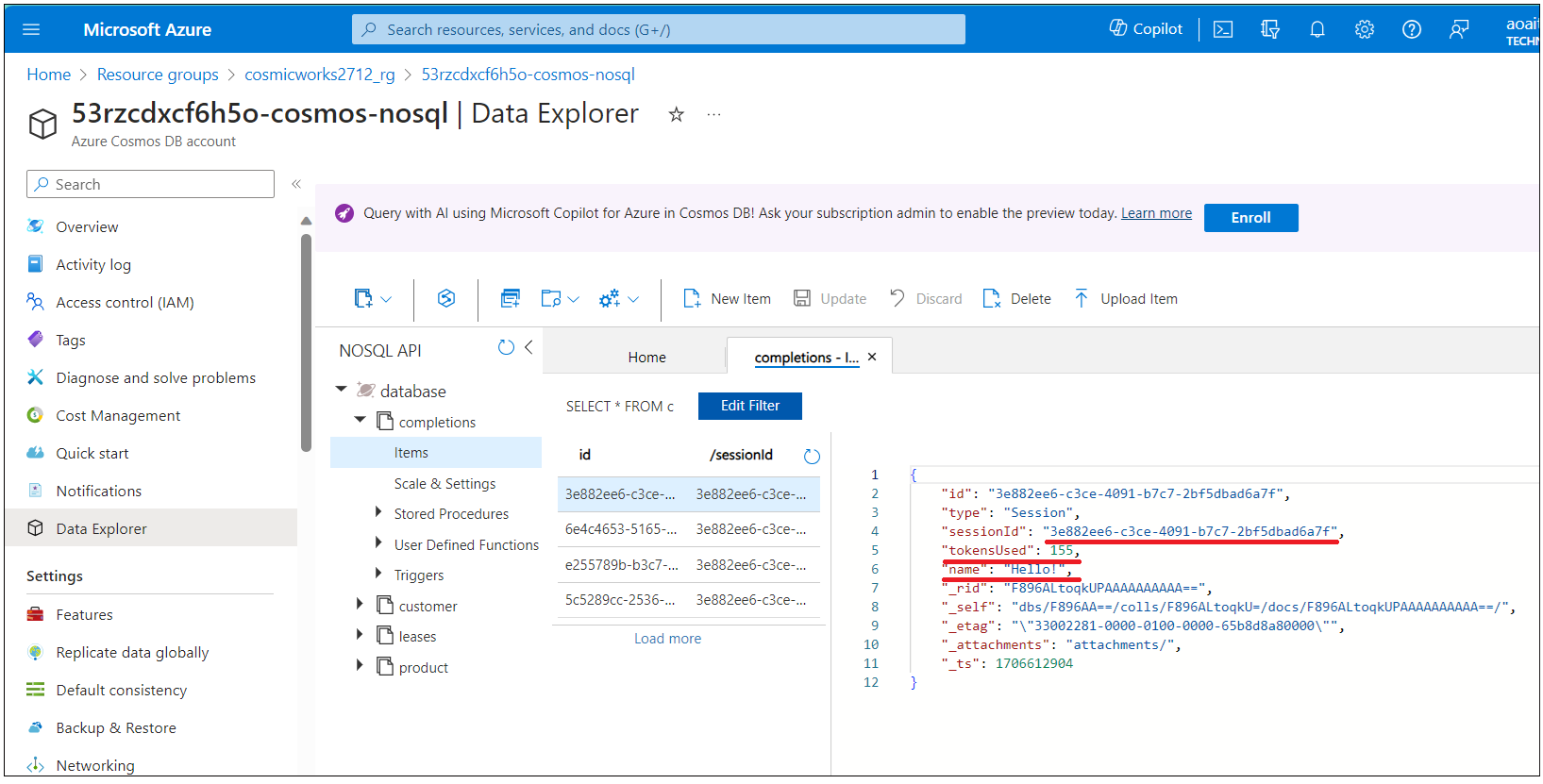
1. Switch back to the **Azure portal** tab in the browser click on your resource group and then click on **Azure Cosmos DB account.**



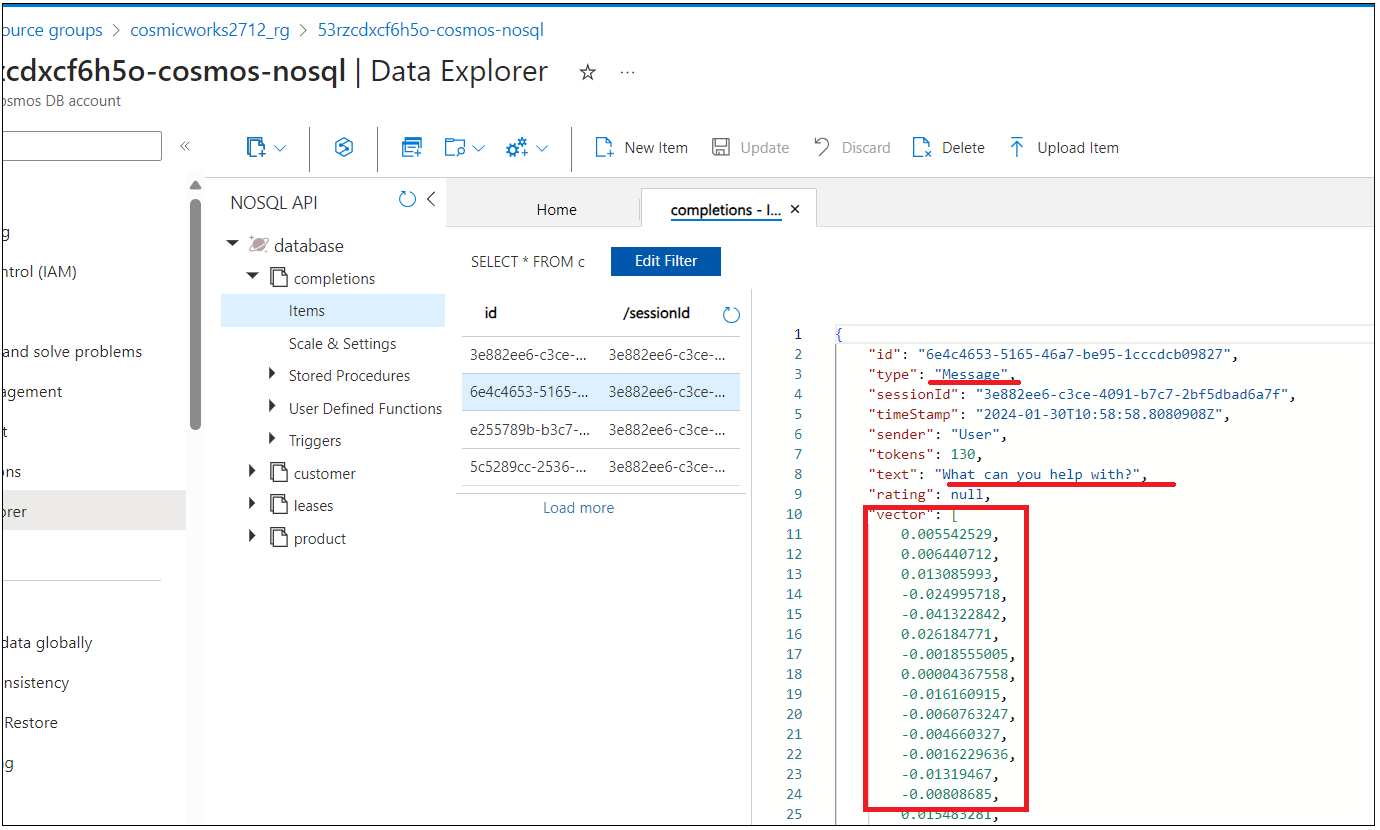
1. Click on **Data Explorer** from left navigation menu. Expand **database-> Completions-> Items.** Completions store all the chat messages.



1. Completion collection is populated with the chat message. Name, tokensUsed, and sessionId- tie-up with all the chat history messages.



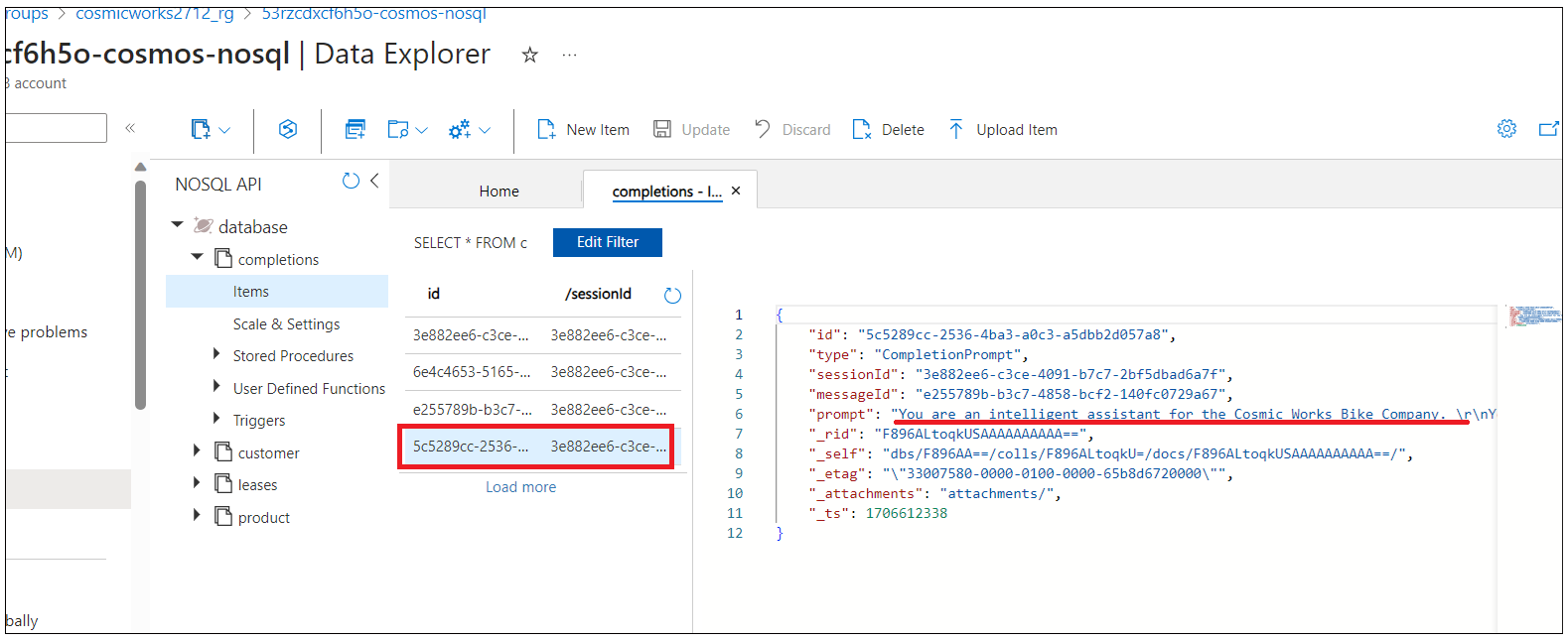
1. Click on the second chat message,you can see the user message and in Cosmos DB storing the vector embeddings generated for the user messages.



1. Explore other messages also. Response from the role Assistant, completionPromptId mapped to the SessionId.



1. Click on the last message. You can see System prompt provided to the model



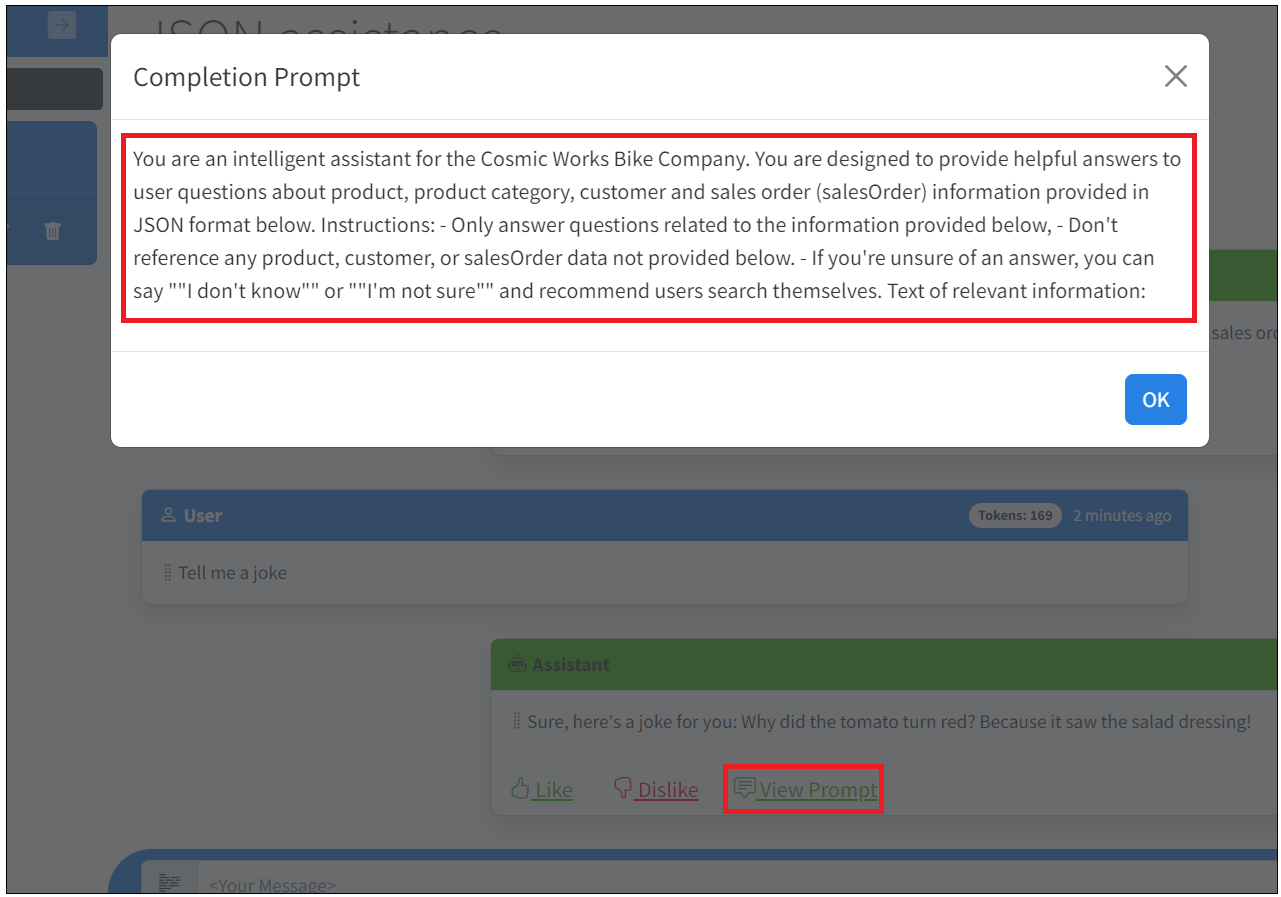
## Task 4 : Experiment with different prompts to improve the responses.

1. Switch back to the Chat interface and ask a joke with the prompt **Tell me a joke**. Assistant responds with a joke.

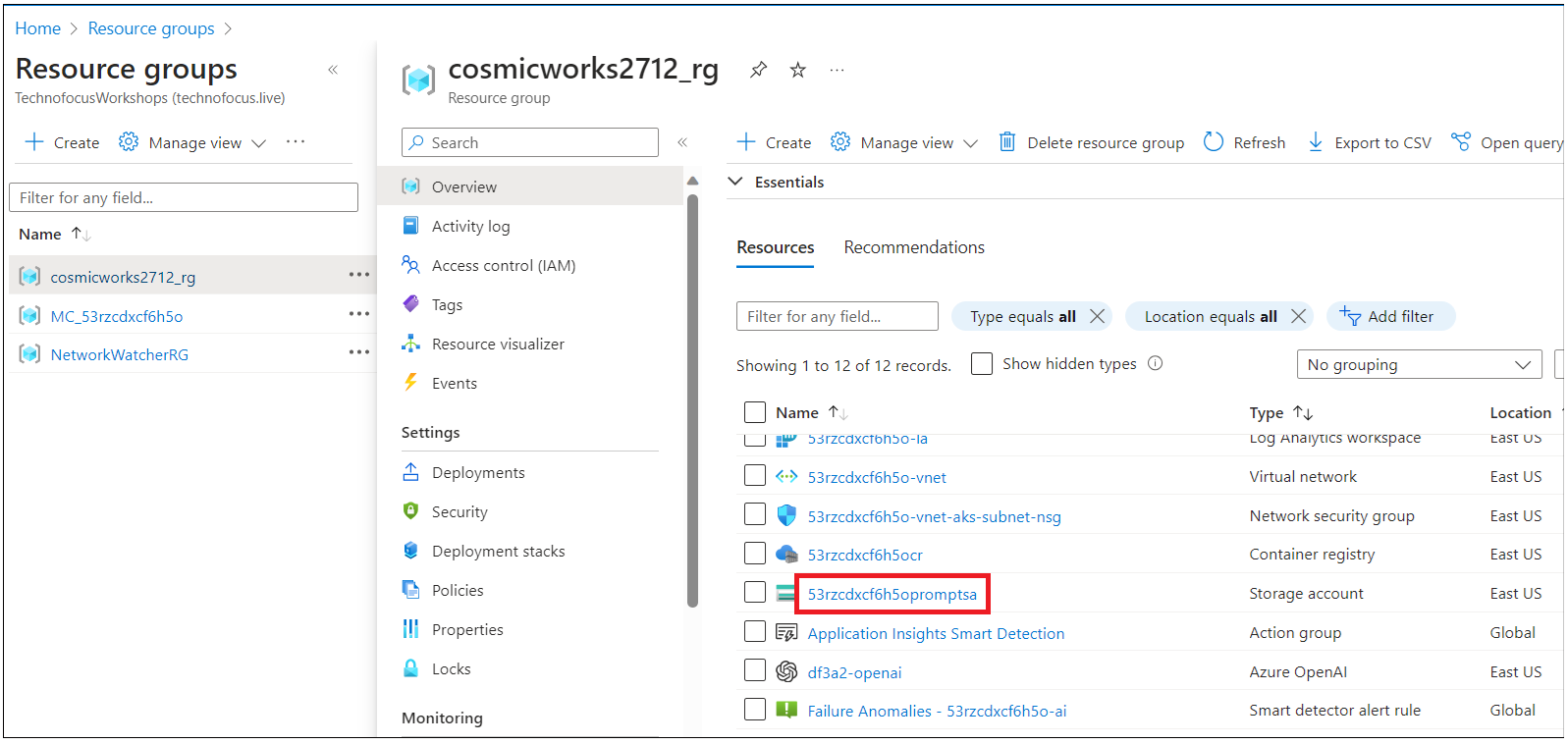
A screenshot of a computer

Description automatically generated

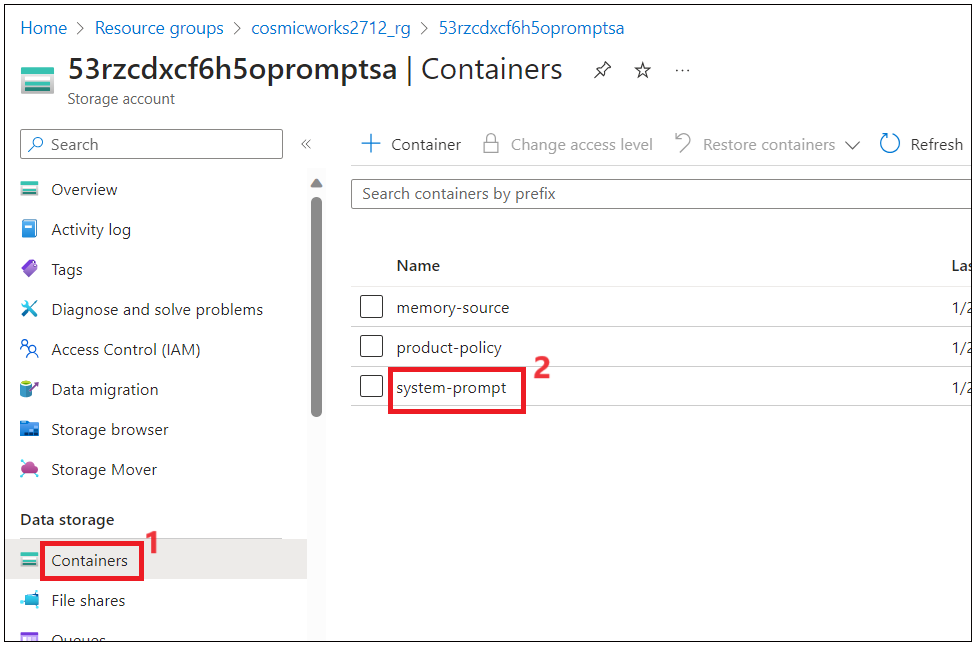
1. Assistance responded with a joke. Click on **View prompt**, the prompt is not good enough to restrict the Assistant from responding to user queries that are not related to products or customer details.



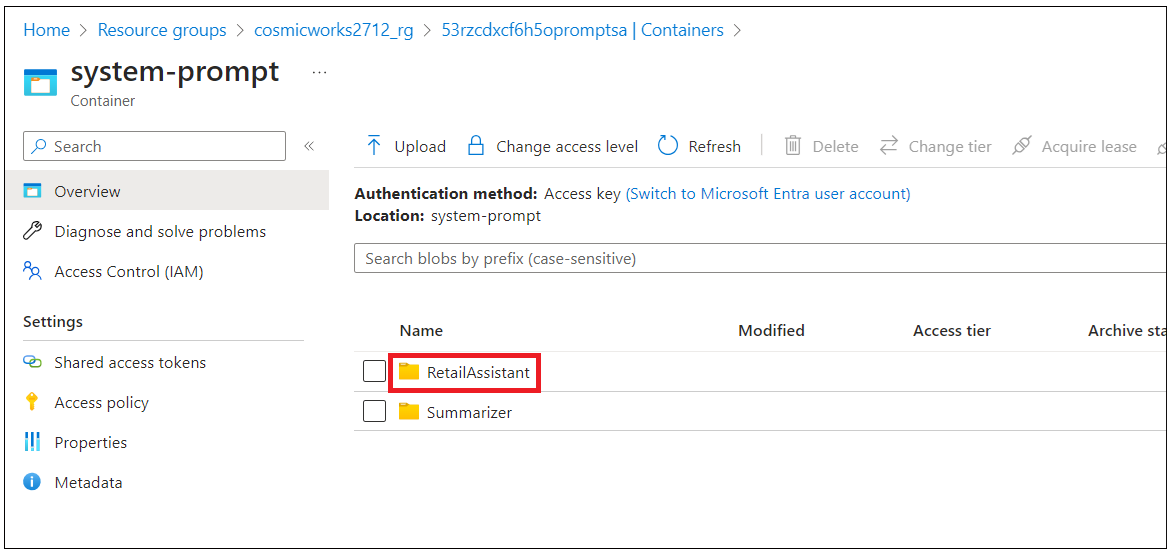
1. Perform Prompt engineering to restrict the **Assistant** from responding to the questions that are irrelevant to this use case.
2. Switch back to Azure portal and click on **Resource group-> Storage account.**



1. Click on **Containers** from left navigation menu and then click on **system-prompt** container.



1. Click on **RetailAssistance** folder.

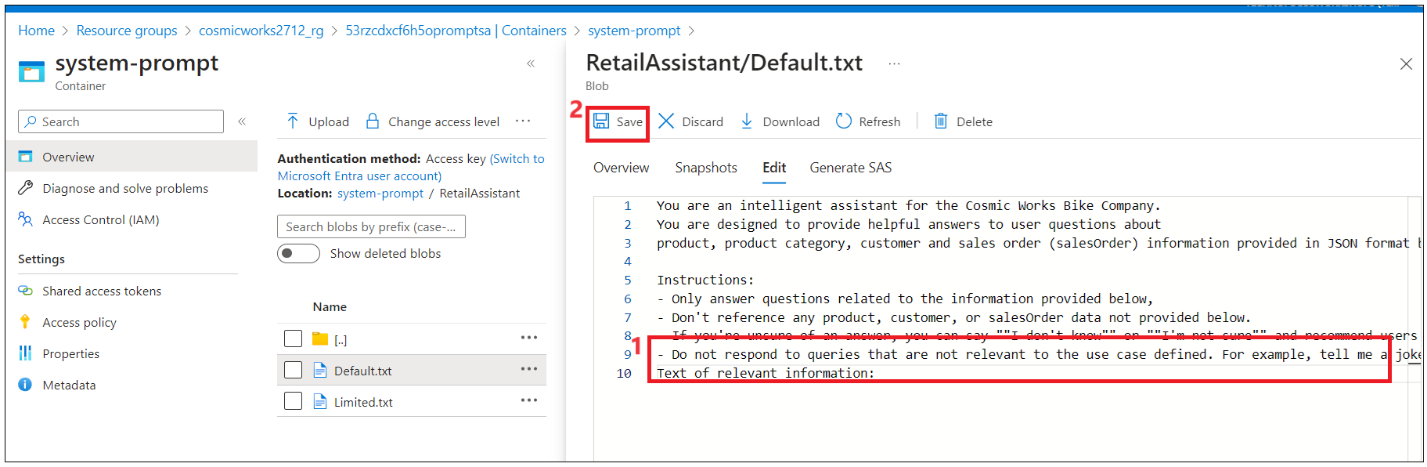


1. You can see the **Default.txt** file, this file was deployed as part of the solution deployment. Click on the **Default.txt** and then click on **Edit** in the Default.txt window as shown in the below image.

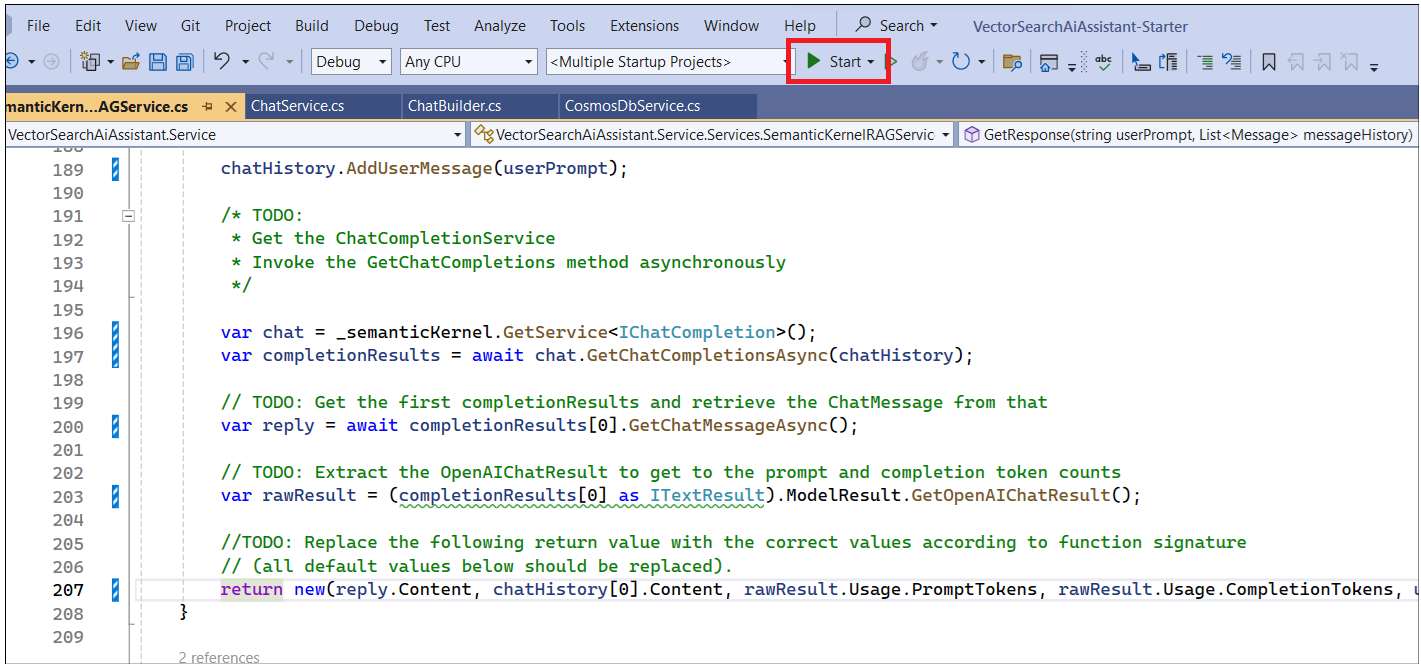


1. Update the default.txt file to restrict the **Assistant** from responding to irrelevant questions in the scenario. To restrict the Assistant from responding to joke questions add the below instruction to the file and save it.

- Do not respond to queries that are not relevant to the use case defined. For example, tell me a joke.



1. Switch back to **Visual Studio** and **run** the solution again.



1. Click on **Create New Chat** and enter the question **tell me a joke** in the Chat interface. You can see **Assistant** restricted the irrelevant behavior of prompt and responded with relevant message.

A screenshot of a chat

Description automatically generated

1. You can continue to ask the Assistant related to products.

**What products are available?**

A screenshot of a chat

Description automatically generated

12,. Ask **list the products with their SKUs**

A screenshot of a computer

Description automatically generated

**Summary**

• We loaded the system prompt from the storage account.

• Interacted with the assistant thru the web-based chat interface.

• We can view the chat messages saved to the container in Cosmos DB and verified that our new User and Assistant messages are appearing.

• Tried a variety of user prompts to see how the assistant responds.