import streamlit as st

from langchain\_ollama import ChatOllama

from langchain\_core.output\_parsers import StrOutputParser

from langchain\_core.prompts import (

SystemMessagePromptTemplate,

HumanMessagePromptTemplate,

AIMessagePromptTemplate,

ChatPromptTemplate

)

**2️⃣ from langchain\_ollama import ChatOllama**

🔹 **What it does:**

* **ChatOllama** is a LangChain wrapper for **Ollama**, a platform for running local LLMs (like Llama 2 or Mistral) on your own machine.
* It allows us to **chat** with an AI model using LangChain.

**3️⃣ from langchain\_core.output\_parsers import StrOutputParser**

🔹 **What it does:**

* The **StrOutputParser** converts the AI’s response into a **clean string** format.
* It ensures that the output from the LLM can be **processed easily**

🔹 **What it does:**

* LangChain allows structured **prompt engineering** using different **prompt templates**.
* These templates define **who is speaking** (System, Human, or AI).

**4.1 SystemMessagePromptTemplate**

* Defines the **AI’s persona & behavior**.
* AI will **always follow** system instructions.

**4.2 HumanMessagePromptTemplate**

* Represents **user input** (messages sent by a human).

| **Code** | **Purpose** |
| --- | --- |
| import streamlit as st | Creates a web interface for AI chatbot |
| from langchain\_ollama import ChatOllama | Connects to local LLMs via Ollama |
| from langchain\_core.output\_parsers import StrOutputParser | Cleans AI responses into plain text |
| SystemMessagePromptTemplate | Defines AI's behavior/personality |
| HumanMessagePromptTemplate | Represents user input messages |
| AIMessagePromptTemplate | Represents AI-generated responses |
| ChatPromptTemplate | Combines prompts into a structured conversation |

#Sidebar configuration

with st.sidebar:

    st.header("Configuration")

    selected\_model= st.selectbox(

        "Choose Model",

    ["deepseek-r1:1.5b", "deepseek-r1:3b"],

    index= 0

    )

    st.divider()

    st.markdown("### Model Capabilities")

    st.markdown("""

    - 🐍 Python Expert

    - 🐞 Debugging Assistant

    - 📝 Code Documentation

    - 💡 Solution Design

    """)

    st.divider()

    st.markdown("Built with [Ollama](https://ollama.ai/) | [LangChain](https://python.langchain.com/)")

**Code Explanation**

**1. Sidebar Initialization**

python

CopyEdit

with st.sidebar:

* This initializes a collapsible sidebar in the Streamlit app.

**2. Sidebar Header**

python

CopyEdit

st.header("Configuration")

* Displays a header titled **"Configuration"** at the top of the sidebar.

**3. Model Selection Dropdown**

python

CopyEdit

selected\_model = st.selectbox(

"Choose Model",

["deepseek-r1:1.5b", "deepseek-r1:3b"],

index=0

)

* **st.selectbox** creates a dropdown menu with two options:
  + "deepseek-r1:1.5b"
  + "deepseek-r1:3b"
* **index=0** sets the default selection to "deepseek-r1:1.5b".

**4. Divider for Separation**

python

CopyEdit

st.divider()

* This adds a horizontal line to visually separate sections.

**5. Display Model Capabilities**

python

CopyEdit

st.markdown("### Model Capabilities")

st.markdown("""

- 🐍 Python Expert

- 🐞 Debugging Assistant

- 📝 Code Documentation

- 💡 Solution Design

""")

* **st.markdown("### Model Capabilities")** creates a subheading.
* The markdown bullet points list the model's key features using emojis.

**6. Additional Divider**

python

CopyEdit

st.divider()

* Another horizontal divider for better UI structuring.

**7. Display Links**

python

CopyEdit

st.markdown("Built with [Ollama](https://ollama.ai/) | [LangChain](https://python.langchain.com/)")

* Uses Markdown to display clickable links to **Ollama** and **LangChain**.

**Final Output (Sidebar UI)**

| **Configuration** |
| --- |
| **Choose Model** (Dropdown) |
| DeepSeek R1: 1.5b (default) |
| DeepSeek R1: 3b |
| --- Divider --- |
| **Model Capabilities** |
| 🐍 Python Expert |
| 🐞 Debugging Assistant |
| 📝 Code Documentation |
| 💡 Solution Design |
| --- Divider --- |
| \*\*Built with Ollama |

#initiate the chat engine

llm\_engine = ChatOllama(

    model= selected\_model,

    base\_url= "http://localhost:11434",

    temperature= 0.3

)

**Attributes & Their Purpose**

| **Attribute** | **Purpose** |
| --- | --- |
| model | Specifies the AI model to use (e.g., "deepseek-r1:1.5b" or "deepseek-r1:3b" from the selectbox). |
| base\_url | The API endpoint where the **Ollama server** is running (http://localhost:11434). |
| temperature | Controls the randomness of responses. Lower values (e.g., 0.3) make the output more deterministic, while higher values (e.g., 0.9) make it more creative. |

**Why Do We Need These Attributes?**

1. **model**
   * This tells Ollama which **AI model** to use for generating responses.
   * In your case, it's dynamically selected from the Streamlit UI.
2. **base\_url**
   * Ollama runs as a local server (http://localhost:11434) where models are hosted.
   * This attribute ensures that the LangChain Ollama wrapper communicates with that server.
3. **temperature**
   * If set to 0, the model gives the same response for identical inputs (good for factual answers).
   * If set to 1.0, responses become more varied and creative.
   * Your value (0.3) balances **consistency** and **creativity**.

**How This Works**

* Once initialized, llm\_engine is used to **generate responses** by passing a user query.
* Example usage:

python

CopyEdit

response = llm\_engine.invoke("What is DeepSeek R1?")

print(response)

* + This sends the query "What is DeepSeek R1?" to the selected **DeepSeek R1** model running in Ollama.
  + The model then generates a response.

#System Prompt Configuration system\_prompt = SystemMessagePromptTemplate.from\_template( "Your are an expert AI coding assistant. Provide concise and correct solution" "with strategic print statements for debugging. Always respond in English" )

this function helps establish a framework for the AI's responses, making it a more effective and user-friendly coding assistant.

**1. SystemMessagePromptTemplate**

* **What It Is**: This is a class from the LangChain library designed to create templates for system messages that the AI can understand and respond to.
* **Purpose**: It helps in defining how the AI should behave or respond during interactions.

**2. from\_template() Method**

* **Functionality**: This method is used to create an instance of the SystemMessagePromptTemplate using a specific string template.
* **Why Use It**: It allows you to define a standard way of communicating with the AI by using a predefined message format.

**3. Prompt String**

* The string provided in the from\_template() method is crucial because it tells the AI what role to play and how to respond. Let's break down the components of this string:
  + **"Your are an expert AI coding assistant."**
    - This establishes the AI's role as a knowledgeable helper in coding.
  + **"Provide concise and correct solution"**
    - This instructs the AI to give clear and accurate answers, which is essential for effective coding assistance.
  + **"with strategic print statements for debugging."**
    - This encourages the AI to include print statements in its code suggestions. These statements help users understand what the code is doing and can aid in debugging, making it easier to spot errors.
  + **"Always respond in English."**
    - This ensures that all responses are in English, promoting clarity and consistency in communication.
* # Session state management
* if "message\_log" not in st.session\_state:
* st.session\_state.message\_log = [{ "role": "ai", "content": "Hi! I am Deepseek. How can I help you code today?"}]

**1. Session State Management**

* **Purpose**: Streamlit uses session state to store data across different interactions with the app. This allows you to maintain information between reruns of the script, which is particularly useful for interactive applications like chatbots.

**2. Checking for message\_log**

python

CopyEdit

if "message\_log" not in st.session\_state:

* **Condition**: This line checks if the key "message\_log" already exists in the session state. If it does not exist, the code inside the if block will execute.
* **Purpose**: It ensures that the message\_log is initialized only once when the user first interacts with the app. This prevents overwriting existing messages during subsequent interactions.

**3. Initializing message\_log**

python

CopyEdit

st.session\_state.message\_log = [{ "role": "ai", "content": "Hi! I am Deepseek. How can I help you code today?"}]

* **Initialization**: If the message\_log does not exist, it is initialized as a list containing a dictionary. This dictionary represents the initial message from the AI.
  + **"role": "ai"**: Indicates that the message is from the AI.
  + **"content": "Hi! I am Deepseek. How can I help you code today?"**: This is the content of the message, greeting the user and inviting them to ask for help.

**Summary of the Code's Purpose**

* **Maintaining Context**: This code snippet helps maintain a chat history or log of messages exchanged between the user and the AI. By storing the initial message in the session state, the application can keep track of the conversation as it progresses.
* **User Interaction**: It creates a welcoming environment for users, providing an introductory message that encourages them to interact with the AI coding assistant.
* The message log is a crucial component for managing conversations in your application. It helps maintain context, provides a record of interactions, and enhances the user's experience by creating a natural flow of dialogue between the user and the AI assistant.
* #Chat container
* chat\_container = st.container()

**1. Chat Container**

* **Purpose**: A container in Streamlit is a layout element that can hold and organize other widgets or components. By creating a chat container, you can group related content together, making it easier to manage and style.

**2. st.container()**

* **Functionality**: This is a built-in Streamlit function that creates a new container. It can be used to encapsulate multiple elements, such as text, input fields, buttons, and any other Streamlit components.
* **Dynamic Updates**: Containers can be updated dynamically during the application’s execution, meaning you can add or modify the content inside the container based on user interactions.

**Purpose of Using a Chat Container**

1. **Organization**: The chat container helps organize the chat-related components, keeping the chat interface separate from other parts of the application (like the sidebar or configuration settings).
2. **Styling**: By using a container, you can easily apply styles or layouts to all the chat elements at once, enhancing the visual appearance of your application.
3. **Dynamic Interaction**: You can use the chat container to display the ongoing conversation between the user and the AI, updating it in real-time as new messages are sent or received.

# Display chat messages

with chat\_container:

for message in st.session\_state.message\_log:

with st.chat\_message(message["role"]):

st.markdown(message["content"])

**1. Display Chat Messages Block**

python

CopyEdit

with chat\_container:

* **Purpose**: This line creates a context within the chat\_container, meaning that all the code inside this block will be organized within that specific container. This helps keep the chat messages visually and structurally grouped together.

**2. For Loop**

python

CopyEdit

for message in st.session\_state.message\_log:

* **Purpose**: This line starts a for loop that iterates over each message stored in st.session\_state.message\_log.
* **message**: In each iteration, message will be a dictionary that represents a single message, containing the role (e.g., "user" or "ai") and the content of the message.

**3. Displaying Each Message**

python

CopyEdit

with st.chat\_message(message["role"]):

* **Purpose**: This line creates a chat message block specifically for the role of the message (either "user" or "ai").
* **st.chat\_message(message["role"])**: This function handles the styling and layout of messages based on their role. Messages from the AI and user can be visually differentiated (e.g., using different colors or alignment).

**4. Markdown for Content**

python

CopyEdit

st.markdown(message["content"])

* **Purpose**: This line uses the st.markdown() function to render the content of the message in Markdown format, allowing for styled text and formatting.
* **message["content"]**: This retrieves the actual text of the message, which will be displayed in the chat interface.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Understanding st.chat\_message(message["role"])**

**Purpose**

* The st.chat\_message(role) function is used to create a chat message block in your Streamlit app that visually distinguishes between messages from different roles, such as "user" and "AI."
* The role parameter is passed to the function, allowing the application to apply specific styles based on whether the message is from the AI or the user.

**How It Works**

1. **Role-Based Styling**:
   * When you specify message["role"] (which could be either "user" or "ai"), the st.chat\_message function applies different styling to the chat messages based on this role.
   * For example, messages from the AI might be displayed in a different color, font, or alignment than those from the user. This makes it visually easier for users to differentiate between who is speaking in the chat.
2. **Visual Differentiation**:
   * **AI Messages**: These might be aligned to the left, have a specific background color (like light blue), or be in a particular font style to indicate that they are generated by the AI.
   * **User Messages**: Conversely, messages from the user might be aligned to the right, have a different background color (like light green), or be styled differently to show they are from the user.

**Purpose of st.markdown()**

* **Rendering Markdown**: st.markdown() allows you to write text using Markdown syntax, which is then converted into styled HTML for display in the Streamlit app. This means you can format text in various ways (e.g., bold, italics, lists) without writing raw HTML.

**Benefits of Using Markdown**

1. **Simple Formatting**: Markdown provides an easy and readable way to apply formatting to text. You can use simple symbols to create headings, lists, links, bold, and italic text.
   * **Example**:
     + \*\*bold text\*\* → **bold text**
     + \*italic text\* → *italic text*
     + # Heading → Displays as a larger heading
2. **Clarity and Readability**: The Markdown syntax is easy to read in its raw form, making it simple to understand the structure of your content even when not rendered.

# Chat input and processing

user\_query = st.chat\_input("Type your coding question here...")

**. Creating a Chat Input Field**

* **Function**: st.chat\_input()
  + This function generates an input field specifically designed for chat applications. It allows users to enter text, similar to how they would in a messaging app.

**1. Function Definition**

python

CopyEdit

def generate\_ai\_response(prompt\_chain):

* **Purpose**: This line defines a function named generate\_ai\_response that takes one argument, prompt\_chain.
* **prompt\_chain**: This argument is expected to be a series of components or steps that will process the input prompt and generate a response from the AI model.

**2. Creating the Processing Pipeline**

python

CopyEdit

processing\_pipeline = prompt\_chain | llm\_engine | StrOutputParser()

* **Purpose**: This line creates a processing pipeline using the | operator, which is likely overloaded to combine different components or stages in the processing sequence.
* **Components**:
  + **prompt\_chain**: The initial part of the pipeline, which prepares the prompt for the AI model.
  + **llm\_engine**: The language model engine (likely your AI model) that will process the prepared prompt and generate a response.
  + **StrOutputParser()**: This is likely a function or class that takes the raw output from the language model and converts it into a usable string format. This step ensures that the output is in the expected format for display or further processing.

**3. Invoking the Pipeline**

python

CopyEdit

return processing\_pipeline.invoke({})

* **Purpose**: This line executes the processing pipeline with an empty dictionary {} as input. The dictionary can be used to pass any additional parameters or context required by the pipeline components.
* **invoke({})**: The invoke method is called on the processing\_pipeline, which runs the entire sequence (from prompt\_chain to the language model to the output parser) and returns the final generated response.

**Summary**

The generate\_ai\_response function is designed to:

* Accept a prompt chain that specifies how to prepare the input for the AI model.
* Create a processing pipeline that combines the prompt chain, the AI model, and the output parser.
* Execute the pipeline to generate a response from the AI model and return that response.

**Example Usage**

Here’s an example of how you might use this function in your application:

python

CopyEdit

# Assume you have a prompt chain prepared for the AI model

user\_prompt\_chain = ... # This would be defined elsewhere in your code

# Generate a response using the AI model

response = generate\_ai\_response(user\_prompt\_chain)

# Output the response

st.markdown(response)

In this example, the user\_prompt\_chain would contain the necessary information for the model, and the generate\_ai\_response function would return the AI's response, which you could then display using Streamlit's st.markdown().