# App.py

mport 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().

# generating ai response

    with st.spinner("Processing..."):

      prompt\_chain = build\_prompt\_chain()

      ai\_response= generate\_ai\_response(prompt\_chain)

  # Add AI response to log

    st.session\_state.message\_log.append({"role": "ai", "content": ai\_response})

    # Rerun to update chat display

    st.rerun()

✅ **What it does?**

* **st.spinner("Processing...")**: Shows a "Processing..." loading animation while AI is generating a response.
* **prompt\_chain = build\_prompt\_chain()**: Builds the conversation history (including system instructions + user messages) into a structured format for the AI.
* **ai\_response = generate\_ai\_response(prompt\_chain)**: Passes the conversation context to the AI model and generates a response.

**3️⃣ Storing AI Response**

python

CopyEdit

st.session\_state.message\_log.append({"role": "ai", "content": ai\_response})

✅ **What it does?**

* Stores the AI's response in st.session\_state.message\_log, keeping track of the conversation.

**4️⃣ Refreshing the UI to Display New Messages**

python

CopyEdit

st.rerun()

✅ **What it does?**

* **Forces Streamlit to rerun the script** and update the chat interface with the latest user and AI messages.
* Without st.rerun(), new messages wouldn’t appear instantly in the UI.

**🚀 Final Summary**

1️⃣ **User sends a message** → It's stored in st.session\_state.message\_log.  
2️⃣ **AI processes the conversation history** → Generates a response.  
3️⃣ **AI response is saved** → The chat history updates.  
4️⃣ **Streamlit reruns (st.rerun())** → The new conversation appears in the UI

# RAG\_DEEP.py

import streamlit as st

from langchain\_community.document\_loaders import PDFPlumberLoader

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

from langchain\_core.vectorstores import InMemoryVectorStore

from langchain\_ollama import OllamaEmbeddings

from langchain\_core.prompts import ChatPromptTemplate

from langchain\_ollama.llms import OllamaLLM

import streamlit as st

**✅ What it does:**

* Imports **Streamlit**, a Python framework for creating interactive web apps.
* Used for **user input, file uploads, and displaying results** in your AI-powered research assistant.

**🔹 Why is it needed?**

* Your app is built with Streamlit, so it's necessary for **handling UI elements** (file upload, chat interface, displaying responses).

python

CopyEdit

from langchain\_community.document\_loaders import PDFPlumberLoader

**✅ What it does:**

* Imports **PDFPlumberLoader**, a document loader from LangChain’s langchain\_community module.
* This loader is used to **read PDFs** and extract text from them.

**🔹 Why is it needed?**

* Since your app allows users to **upload PDFs**, this tool extracts **text** from those documents.
* Without it, you **cannot process PDFs** for AI-powered querying.

python

CopyEdit

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

**✅ What it does:**

* Imports RecursiveCharacterTextSplitter, a tool for **breaking large texts into smaller chunks**.
* This is crucial because AI models **perform better with structured, smaller text inputs**.

**🔹 Why is it needed?**

* **PDFs contain long text** that must be **split** into manageable parts before being stored in a vector database.
* Helps in **efficient retrieval and embedding storage**.

python

CopyEdit

from langchain\_core.vectorstores import InMemoryVectorStore

**✅ What it does:**

* Imports InMemoryVectorStore, which acts as a **temporary database** to store text embeddings.
* Stores vector representations of PDF content in memory for **quick retrieval**.

**🔹 Why is it needed?**

* AI models cannot directly process raw text for retrieval.
* **Embeddings (numerical representations of text)** must be stored to allow **fast similarity search** when answering queries.

python

CopyEdit

from langchain\_ollama import OllamaEmbeddings

**✅ What it does:**

* Imports OllamaEmbeddings, a tool that **converts text into embeddings** (numerical vectors).
* Uses **Ollama's DeepSeek-R1 model** for embedding generation.

**🔹 Why is it needed?**

* AI cannot compare raw text directly. Instead, **semantic similarity** is determined using embeddings.
* The app stores these embeddings in the vector database for **fast, AI-powered retrieval**.

python

CopyEdit

from langchain\_core.prompts import ChatPromptTemplate

**✅ What it does:**

* Imports ChatPromptTemplate, a tool to **define structured prompts** for the AI model.
* It helps **format user queries and retrieved document context** in a way the model understands.

**🔹 Why is it needed?**

* Instead of sending raw text, well-structured prompts improve **AI accuracy** and ensure the model gives **concise answers**.
* Helps **guide the AI** to stay factual and relevant.

python

CopyEdit

from langchain\_ollama.llms import OllamaLLM

**✅ What it does:**

* Imports OllamaLLM, which allows interaction with **Ollama’s DeepSeek-R1 model** for text generation.
* This is the **core AI model** that answers user queries.

**🔹 Why is it needed?**

* **Processes user queries** and generates responses based on PDF content.
* This is the **language model** that enables the chatbot to work.

**📌 Summary Table**

| **Line** | **Purpose** | **Why It's Needed?** |
| --- | --- | --- |
| import streamlit as st | Provides UI components | Allows user interaction (chat, file upload, etc.) |
| from langchain\_community.document\_loaders import PDFPlumberLoader | Reads PDFs | Extracts text from uploaded documents |
| from langchain\_text\_splitters import RecursiveCharacterTextSplitter | Splits long text | AI needs manageable chunks for better processing |
| from langchain\_core.vectorstores import InMemoryVectorStore | Stores embeddings | Enables fast, AI-powered document retrieval |
| from langchain\_ollama import OllamaEmbeddings | Creates text embeddings | Converts text into numerical format for searching |
| from langchain\_core.prompts import ChatPromptTemplate | Structures AI prompts | Ensures AI responses are relevant and formatted |
| from langchain\_ollama.llms import OllamaLLM | Loads AI model | Generates AI-powered responses |

st.markdown("""

    <style>

    .stApp {

        background-color: #0E1117;

        color: #FFFFFF;

    }

    /\* Chat Input Styling \*/

    .stChatInput input {

        background-color: #1E1E1E !important;

        color: #FFFFFF !important;

        border: 1px solid #3A3A3A !important;

    }

    /\* User Message Styling \*/

    .stChatMessage[data-testid="stChatMessage"]:nth-child(odd) {

        background-color: #1E1E1E !important;

        border: 1px solid #3A3A3A !important;

        color: #E0E0E0 !important;

        border-radius: 10px;

        padding: 15px;

        margin: 10px 0;

    }

    /\* Assistant Message Styling \*/

    .stChatMessage[data-testid="stChatMessage"]:nth-child(even) {

        background-color: #2A2A2A !important;

        border: 1px solid #404040 !important;

        color: #F0F0F0 !important;

        border-radius: 10px;

        padding: 15px;

        margin: 10px 0;

    }

    /\* Avatar Styling \*/

    .stChatMessage .avatar {

        background-color: #00FFAA !important;

        color: #000000 !important;

    }

    /\* Text Color Fix \*/

    .stChatMessage p, .stChatMessage div {

        color: #FFFFFF !important;

    }

    .stFileUploader {

        background-color: #1E1E1E;

        border: 1px solid #3A3A3A;

        border-radius: 5px;

        padding: 15px;

    }

    h1, h2, h3 {

        color: #00FFAA !important;

    }

    </style>

    """, unsafe\_allow\_html=True)

**1️⃣ Applying the Dark Theme to the Whole App**

css

CopyEdit

.stApp {

background-color: #0E1117;

color: #FFFFFF;

}

✅ **What it does:**

* Changes the **entire background** to a dark theme (#0E1117 - dark blue-black).
* Sets the **default text color** to white (#FFFFFF) for readability.

🔹 **Why is it needed?**

* Creates a **modern dark theme** that is easier on the eyes.
* Helps **match a professional AI assistant look**.

**2️⃣ Styling the Chat Input Box**

css

CopyEdit

.stChatInput input {

background-color: #1E1E1E !important;

color: #FFFFFF !important;

border: 1px solid #3A3A3A !important;

}

✅ **What it does:**

* Sets the **chat input box's background** to #1E1E1E (dark gray).
* Ensures the **text inside the input box is white** for visibility.
* Adds a **gray border** for a neat look.

🔹 **Why is it needed?**

* Makes the chat **blend with the dark theme**.
* Ensures **text is always readable** in the chat input field.

**3️⃣ Styling User Messages (Odd Messages)**

css

CopyEdit

.stChatMessage[data-testid="stChatMessage"]:nth-child(odd) {

background-color: #1E1E1E !important;

border: 1px solid #3A3A3A !important;

color: #E0E0E0 !important;

border-radius: 10px;

padding: 15px;

margin: 10px 0;

}

✅ **What it does:**

* Applies **dark gray (#1E1E1E) background** for user messages.
* Adds **light gray text (#E0E0E0)** for contrast.
* **Rounded corners (border-radius: 10px)** for a smooth, chat-bubble look.
* Adds **padding and margin** for better spacing.

🔹 **Why is it needed?**

* Ensures **user messages look distinct** from assistant messages.
* Makes the chat interface **visually appealing and easy to read**.

**4️⃣ Styling Assistant Messages (Even Messages)**

css

CopyEdit

.stChatMessage[data-testid="stChatMessage"]:nth-child(even) {

background-color: #2A2A2A !important;

border: 1px solid #404040 !important;

color: #F0F0F0 !important;

border-radius: 10px;

padding: 15px;

margin: 10px 0;

}

✅ **What it does:**

* Uses a **darker gray (#2A2A2A) background** for assistant messages.
* Adds **lighter text (#F0F0F0)** for easy readability.
* **Maintains padding, margin, and border-radius** for a clean UI.

🔹 **Why is it needed?**

* Helps **differentiate AI responses from user messages**.
* Ensures **text is readable even in a dark theme**.

**5️⃣ Customizing the Chat Avatar**

css

CopyEdit

.stChatMessage .avatar {

background-color: #00FFAA !important;

color: #000000 !important;

}

✅ **What it does:**

* Sets the **chat avatar color to #00FFAA (neon green)**.
* Changes **text inside the avatar to black (#000000)**.

🔹 **Why is it needed?**

* **Makes avatars stand out** against the dark background.
* **Improves UI clarity** by making chat participants easy to identify.

**6️⃣ Fixing Text Color Inside Chat Messages**

css

CopyEdit

.stChatMessage p, .stChatMessage div {

color: #FFFFFF !important;

}

✅ **What it does:**

* Ensures **all text in chat messages is white (#FFFFFF)**.

🔹 **Why is it needed?**

* Prevents **invisible or unreadable text** when messages are displayed.

**7️⃣ Styling the File Uploader Component**

css

CopyEdit

.stFileUploader {

background-color: #1E1E1E;

border: 1px solid #3A3A3A;

border-radius: 5px;

padding: 15px;

}

✅ **What it does:**

* Changes the **file upload box background** to match the dark theme.
* Adds **border, padding, and rounded corners** for a polished look.

🔹 **Why is it needed?**

* Makes the **file upload feature blend with the overall UI**.
* Ensures **it looks professional and easy to use**.

**8️⃣ Styling Headers (H1, H2, H3)**

css

CopyEdit

h1, h2, h3 {

color: #00FFAA !important;

}

✅ **What it does:**

* Changes **all headers (H1, H2, H3) to #00FFAA (neon green)**.

🔹 **Why is it needed?**

* Makes **headings stand out** against the dark background.
* Improves **readability and user focus**.

**🚀 Summary Table**

| **CSS Block** | **What It Does?** | **Why It's Needed?** |
| --- | --- | --- |
| .stApp | Sets dark mode background | Enhances UI aesthetics |
| .stChatInput input | Styles the chat input box | Ensures readability in dark mode |
| .stChatMessage:nth-child(odd) | Styles user messages | Differentiates user messages from AI responses |
| .stChatMessage:nth-child(even) | Styles assistant messages | Creates contrast between user and AI messages |
| .stChatMessage .avatar | Customizes avatar color | Improves user recognition |
| .stChatMessage p, .stChatMessage div | Fixes text color in chat | Prevents unreadable text |
| .stFileUploader | Styles the file upload box | Matches dark theme and improves UI |
| h1, h2, h3 | Changes header colors | Enhances visibility and focus |

PROMPT\_TEMPLATE = """

You are an expert research assistant. Use the provided context to answer the query.

If unsure, state that you don't know. Be concise and factual (max 3 sentences).

Query: {user\_query}

Context: {document\_context}

Answer:

"""

| **Component** | **Explanation** | **Why It's Needed?** |
| --- | --- | --- |
| "You are an expert research assistant." | Defines the model’s **role** as an expert research assistant. | Ensures the AI provides **precise, authoritative responses**. |
| "Use the provided context to answer the query." | Tells the AI to **use relevant information** from retrieved documents. | Prevents hallucinations and keeps responses **grounded in facts**. |
| "If unsure, state that you don't know." | Ensures the AI doesn’t make up answers when lacking sufficient context. | Improves **reliability and trustworthiness** of responses. |
| "Be concise and factual (max 3 sentences)." | Limits response length to **three sentences**. | Keeps answers **short, clear, and to the point**. |
| **Query: {user\_query}** | Placeholder for the **user's input question**. | Dynamically injects the **user's query** into the prompt. |
| **Context: {document\_context}** | Placeholder for **retrieved document content**. | Ensures the AI responds using **relevant knowledge** from documents. |
| **Answer:** | Prepares the model to generate the **final response**. | Gives a **structured output format**. |

PDF\_STORAGE\_PATH = 'document\_store/pdfs/'

EMBEDDING\_MODEL = OllamaEmbeddings(model="deepseek-r1:1.5b")

DOCUMENT\_VECTOR\_DB = InMemoryVectorStore(EMBEDDING\_MODEL)

LANGUAGE\_MODEL = OllamaLLM(model="deepseek-r1:1.5b")

| **Code** | **Explanation** | **Why It's Needed?** |
| --- | --- | --- |
| PDF\_STORAGE\_PATH = 'document\_store/pdfs/' | Defines the **directory path** where uploaded PDFs will be stored. | Helps organize and manage uploaded PDFs for **processing and retrieval**. |
| EMBEDDING\_MODEL = OllamaEmbeddings(model="deepseek-r1:1.5b") | Creates an **embedding model** using **DeepSeek R1 (1.5B parameters)**. | Converts text into **numerical vectors** for semantic search and retrieval. |
| DOCUMENT\_VECTOR\_DB = InMemoryVectorStore(EMBEDDING\_MODEL) | Initializes an **in-memory vector database** to store embeddings. | Allows **quick retrieval** of relevant document chunks during querying. |
| LANGUAGE\_MODEL = OllamaLLM(model="deepseek-r1:1.5b") | Loads the **DeepSeek R1 LLM** as the main **response generation model**. | Processes queries and generates **natural language responses** based on retrieved document context. |

def save\_uploaded\_file(uploaded\_file):

file\_path = PDF\_STORAGE\_PATH + uploaded\_file.name

with open(file\_path, "wb") as file:

file.write(uploaded\_file.getbuffer())

return file\_path

def load\_pdf\_documents(file\_path):

document\_loader = PDFPlumberLoader(file\_path)

return document\_loader.load()

**🔹 Purpose:**

This function takes an uploaded PDF file and **saves it** to a predefined directory (document\_store/pdfs/).

**🔹 Code Breakdown:**

python

CopyEdit

def save\_uploaded\_file(uploaded\_file):

* **Defines a function** that takes uploaded\_file as an argument.
* uploaded\_file is an object returned by st.file\_uploader() in **Streamlit**, which allows users to upload files.

python

CopyEdit

file\_path = PDF\_STORAGE\_PATH + uploaded\_file.name

* **Concatenates the directory path (PDF\_STORAGE\_PATH)** and the **uploaded file's name**.
* The file will be stored in 'document\_store/pdfs/' with its original name.

python

CopyEdit

with open(file\_path, "wb") as file:

* Opens the file **in write-binary (wb) mode**, ensuring it is written in **binary format** (necessary for PDFs).

python

CopyEdit

file.write(uploaded\_file.getbuffer())

* Retrieves the **raw binary data** of the uploaded file using uploaded\_file.getbuffer().
* **Writes the binary data** to the opened file, effectively saving it.

python

CopyEdit

return file\_path

* Returns the **file path** where the uploaded file was saved.
* This path is later used to **load and process** the document.

**2️⃣ load\_pdf\_documents(file\_path)**

**🔹 Purpose:**

This function **reads and extracts text** from a saved PDF file using PDFPlumberLoader from **LangChain**.

**🔹 Code Breakdown:**

python

CopyEdit

def load\_pdf\_documents(file\_path):

* Defines a function load\_pdf\_documents() that **accepts** file\_path (the location of the saved PDF file).

python

CopyEdit

document\_loader = PDFPlumberLoader(file\_path)

* Initializes PDFPlumberLoader, which is a **LangChain document loader** for handling PDFs.
* It allows **reading and extracting structured text** from a given PDF file.

python

CopyEdit

return document\_loader.load()

* Calls .load() on the document\_loader, which **reads** the entire PDF and **returns a structured document**.
* The returned data is typically in the form of a **list of text chunks**.

def chunk\_documents(raw\_documents):

text\_processor = RecursiveCharacterTextSplitter(

chunk\_size=1000,

chunk\_overlap=200,

add\_start\_index=True

)

return text\_processor.split\_documents(raw\_documents)

This function is used to **split large text documents into smaller chunks** to optimize text processing, embedding, and retrieval.

**1️⃣ Purpose**

✅ **Handles long documents** by breaking them into manageable chunks.  
✅ **Improves retrieval accuracy** by ensuring that only relevant chunks are retrieved.  
✅ **Enhances efficiency** for embedding and vector search.

**2️⃣ Code Breakdown**

python

CopyEdit

def chunk\_documents(raw\_documents):

* Defines the function chunk\_documents() that takes **raw documents** (extracted text from PDFs) as input.

python

CopyEdit

text\_processor = RecursiveCharacterTextSplitter(

* Initializes an instance of RecursiveCharacterTextSplitter, which is a **LangChain utility** for **splitting text** into smaller chunks.

python

CopyEdit

chunk\_size=1000,

chunk\_overlap=200,

add\_start\_index=True

* **chunk\_size=1000** → Each chunk will contain **1000 characters**.
* **chunk\_overlap=200** → **200 characters from the previous chunk** are included in the next one.
  + This overlap helps in maintaining **context** across chunks.
  + Prevents loss of **important information** at chunk boundaries.
* **add\_start\_index=True** → Keeps track of where each chunk **starts** in the original document.

python

CopyEdit

return text\_processor.split\_documents(raw\_documents)

* Calls .split\_documents(raw\_documents) on text\_processor, which **processes** and **splits** the text.
* Returns a **list of smaller chunks**.

**3️⃣ Why Is This Function Needed?**

| **Problem** | **Solution (Using this Function)** |
| --- | --- |
| Large documents are hard to process | **Breaks text into smaller chunks** |
| Embedding entire documents is inefficient | **Smaller embeddings improve retrieval accuracy** |
| Context may be lost when splitting | **Uses overlapping text to retain context** |
|  |  |

def index\_documents(document\_chunks):

DOCUMENT\_VECTOR\_DB.add\_documents(document\_chunks)

def find\_related\_documents(query):

return DOCUMENT\_VECTOR\_DB.similarity\_search(query)

**1️⃣ index\_documents(document\_chunks)**

**🔹 Purpose**

✅ **Stores processed document chunks** in a vector database.  
✅ **Enables fast similarity-based retrieval** of relevant information.

**🔹 Code Breakdown**

python

CopyEdit

def index\_documents(document\_chunks):

* Defines a function that takes document\_chunks (the text chunks created earlier).

python

CopyEdit

DOCUMENT\_VECTOR\_DB.add\_documents(document\_chunks)

* **Adds** the text chunks to the DOCUMENT\_VECTOR\_DB, which is an **in-memory vector store**.
* This stores the **embeddings** of each document chunk so they can be efficiently retrieved later.

**🔹 Why Is This Needed?**

| **Problem** | **Solution (Using This Function)** |
| --- | --- |
| Without indexing, documents cannot be searched | **Indexes the documents into a vector database** |
| Searching entire documents is slow | **Retrieves only relevant chunks quickly** |
| Traditional keyword search is inefficient | **Uses embeddings for semantic search** |

def generate\_answer(user\_query, context\_documents):

context\_text = "\n\n".join([doc.page\_content for doc in context\_documents])

conversation\_prompt = ChatPromptTemplate.from\_template(PROMPT\_TEMPLATE)

response\_chain = conversation\_prompt | LANGUAGE\_MODEL

return response\_chain.invoke({"user\_query": user\_query, "document\_context": context\_text})

This function is responsible for **generating an AI-powered response** based on the retrieved relevant documents. It **takes a user query and relevant document chunks** as input, then **uses a language model (DeepSeek R1) to generate an answer**.

**1️⃣ Function Breakdown**

python

CopyEdit

def generate\_answer(user\_query, context\_documents):

* **Defines a function** that takes two arguments:
  + user\_query: The user's question.
  + context\_documents: The most relevant document chunks retrieved from find\_related\_documents().

**2️⃣ Extract Text from Documents**

python

CopyEdit

context\_text = "\n\n".join([doc.page\_content for doc in context\_documents])

* **Extracts the actual text content** (page\_content) from each document chunk.
* **Joins** all document chunks with \n\n (double newline) to keep formatting clear.
* This combined text serves as the **context** for answering the user’s query.

**3️⃣ Create a Prompt**

python

CopyEdit

conversation\_prompt = ChatPromptTemplate.from\_template(PROMPT\_TEMPLATE)

* **Uses ChatPromptTemplate** from langchain\_core.prompts to create a structured prompt.
* PROMPT\_TEMPLATE ensures the model follows a **specific format** for answering.

**Example of PROMPT\_TEMPLATE:**

arduino

CopyEdit

You are an expert research assistant. Use the provided context to answer the query.

If unsure, state that you don't know. Be concise and factual (max 3 sentences).

Query: {user\_query}

Context: {document\_context}

Answer:

* Ensures the model **stays factual and concise**.
* Uses {user\_query} and {document\_context} as placeholders, which will be replaced dynamically.

**4️⃣ Create a Response Chain**

python

CopyEdit

response\_chain = conversation\_prompt | LANGUAGE\_MODEL

* This **chains** the conversation\_prompt with the LANGUAGE\_MODEL (DeepSeek R1).
* The | operator in LangChain is used to **pipe data between components**.
* This means:
  1. The **user query and context** are inserted into PROMPT\_TEMPLATE.
  2. The **final formatted prompt** is passed to LANGUAGE\_MODEL.
  3. The **AI model generates a response** based on the provided context.

**5️⃣ Generate the Answer**

python

CopyEdit

return response\_chain.invoke({"user\_query": user\_query, "document\_context": context\_text})

* **Replaces placeholders** ({user\_query} and {document\_context}) with actual values.
* Calls .invoke() to **execute the model and generate the final answer**.
* **Returns the AI-generated response**.

**2️⃣ Why Is This Function Needed?**

| **Problem** | **Solution (Using generate\_answer())** |
| --- | --- |
| Raw documents are too long to read | Extracts only relevant parts and generates precise answers |
| AI models need structured input | Uses a **custom prompt template** for clarity |
| Traditional search gives only keyword matches | **Uses embeddings & LLM** for better understanding |
| Users need direct, concise answers | Limits responses to **max 3 sentences** |

# UI Configuration

st.title("📘 DocuMind AI")

st.markdown("### Your Intelligent Document Assistant")

st.markdown("---")

# File Upload Section

uploaded\_pdf = st.file\_uploader(

"Upload Research Document (PDF)",

type="pdf",

help="Select a PDF document for analysis",

accept\_multiple\_files=False

)

**2️⃣ Explanation of Parameters**

| **Parameter** | **Purpose** |
| --- | --- |
| "Upload Research Document (PDF)" | 📌 **Title** displayed above the file uploader. |
| type="pdf" | 🗂 **Restricts uploads** to only **PDF** files. |
| help="Select a PDF document for analysis" | ℹ️ Shows **tooltip** when hovering over the uploader. |
| accept\_multiple\_files=False | 🚫 **Ensures only one file** can be uploaded at a time. |

**3️⃣ Why Is This Needed?**

| **Problem** | **Solution Using st.file\_uploader()** |
| --- | --- |
| Users need to provide PDFs for processing | **Creates an interactive upload button** |
| Only PDF files should be accepted | **Restricts to .pdf format** |
| Users might need guidance on what to upload | **Shows a tooltip with instructions** |
| Multiple file uploads could complicate processing | **Limits upload to a single file** |

**4️⃣ Example: How It Works in Streamlit**

When you run this in a Streamlit app:

1. **A file uploader appears** on the webpage.
2. **User selects a PDF file** from their system.
3. **The selected file is stored** in the uploaded\_pdf variable.
4. If uploaded\_pdf is None, it means **no file is uploaded yet**.

if uploaded\_pdf:

saved\_path = save\_uploaded\_file(uploaded\_pdf)

raw\_docs = load\_pdf\_documents(saved\_path)

processed\_chunks = chunk\_documents(raw\_docs)

index\_documents(processed\_chunks)

st.success("✅ Document processed successfully! Ask your questions below.")

user\_input = st.chat\_input("Enter your question about the document...")

if user\_input:

with st.chat\_message("user"):

st.write(user\_input)

with st.spinner("Analyzing document..."):

relevant\_docs = find\_related\_documents(user\_input)

ai\_response = generate\_answer(user\_input, relevant\_docs)

with st.chat\_message("assistant", avatar="🤖"):

st.write(ai\_response)

**1️⃣ Step-by-Step Code Execution**

python

CopyEdit

if uploaded\_pdf:

🔹 **Checks if the user uploaded a PDF file**.  
👉 If uploaded\_pdf is **not None**, the program proceeds with processing.

**📌 Step 1: Save the Uploaded File**

python

CopyEdit

saved\_path = save\_uploaded\_file(uploaded\_pdf)

🔹 Calls the save\_uploaded\_file() function to **store the uploaded file** on the local system.

✅ **Why?**

* Streamlit st.file\_uploader() does not save files to disk by default.
* This function ensures the PDF is **stored persistently** for further processing.

**📌 Step 2: Load the PDF and Extract Text**

python

CopyEdit

raw\_docs = load\_pdf\_documents(saved\_path)

🔹 Calls load\_pdf\_documents(saved\_path) to **extract text from the saved PDF**.

✅ **Why?**

* Extracts content using PDFPlumberLoader so that AI can process it.
* Converts **unstructured PDF content** into a structured format.

**📌 Step 3: Split the Text into Chunks**

python

CopyEdit

processed\_chunks = chunk\_documents(raw\_docs)

🔹 Calls chunk\_documents(raw\_docs) to **break long text into smaller chunks**.

✅ **Why?**

* Large documents must be **split into smaller parts** for efficient processing.
* **RecursiveCharacterTextSplitter** ensures **overlapping chunks** to maintain context.

**📌 Step 4: Store the Processed Chunks in Vector Database**

python

CopyEdit

index\_documents(processed\_chunks)

🔹 Calls index\_documents(processed\_chunks) to **store chunks in a vector database** for retrieval.

✅ **Why?**

* AI must **search and retrieve** relevant document sections efficiently.
* Uses **vector embeddings** to find the most relevant text for a user query.

**📌 Step 5: Confirm Successful Processing**

python

CopyEdit

st.success("✅ Document processed successfully! Ask your questions below.")

🔹 Displays a **confirmation message** in the Streamlit app after processing.

✅ **Why?**

* Provides feedback to the user, ensuring the document is ready for interaction.

**2️⃣ Step-by-Step User Query Handling**

python

CopyEdit

user\_input = st.chat\_input("Enter your question about the document...")

🔹 Creates a **chat input box** where users can type their questions.

✅ **Why?**

* Allows users to **ask questions** related to the uploaded document.

**📌 Step 6: Display User's Question**

python

CopyEdit

if user\_input:

with st.chat\_message("user"):

st.write(user\_input)

🔹 **Checks if the user has entered a query**  
🔹 Displays the **user’s question** inside the chat window.

✅ **Why?**

* Enhances user experience by **showing their query** before AI responds.

**📌 Step 7: Find Relevant Documents**

python

CopyEdit

with st.spinner("Analyzing document..."):

relevant\_docs = find\_related\_documents(user\_input)

🔹 Calls find\_related\_documents(user\_input) to **retrieve the most relevant document sections**.

✅ **Why?**

* Uses **vector similarity search** to fetch the **most relevant** information.

**📌 Step 8: Generate AI Response**

python

CopyEdit

ai\_response = generate\_answer(user\_input, relevant\_docs)

🔹 Calls generate\_answer(user\_input, relevant\_docs) to **generate an AI-powered response**.

✅ **Why?**

* Uses **DeepSeek R1 (Ollama LLM)** to **summarize** and **answer** based on the document.

**📌 Step 9: Display AI Response**

python

CopyEdit

with st.chat\_message("assistant", avatar="🤖"):

st.write(ai\_response)

🔹 Shows the **AI-generated response** in the Streamlit chat UI.

✅ **Why?**

* Provides a **conversational, chat-like interface** where the AI answers the query.

**3️⃣ Why is This Needed?**

| **Problem** | **How This Code Solves It** |
| --- | --- |
| PDFs are **not structured for AI** | Extracts & processes text for **better AI understanding**. |
| AI needs **relevant document parts** to answer accurately | Uses **vector search** to find the best match. |
| Users need a **simple chat interface** | Streamlit **chat UI** provides an intuitive experience. |

**4️⃣ Complete Flow of Execution**

1️⃣ **User uploads a PDF.**  
2️⃣ The file **is saved** to disk.  
3️⃣ Text **is extracted and split** into small chunks.  
4️⃣ Chunks **are stored in a vector database** for retrieval.  
5️⃣ The user **enters a question** in the chat.  
6️⃣ The system **retrieves relevant document sections**.  
7️⃣ AI **generates a response** based on the document.  
8️⃣ The AI response **is displayed** in the chat.