**About LangChain and its Top-level building blocks**

**Why is LangChain important?**

LLMs excel at responding to prompts in a general context, but struggle in a specific domain they were never trained on. *Prompts* are queries people use to seek responses from an LLM. For example, an LLM can provide an answer to how much a computer costs by providing an estimate. However, it can't list the price of a specific computer model that your company sells.

To do that, machine learning engineers must integrate the LLM with the organization’s internal data sources and apply [*prompt engineering*](https://aws.amazon.com/what-is/prompt-engineering/)—a practice where a data scientist refines inputs to a generative model with a specific structure and context.

LangChain streamlines intermediate steps to develop such data-responsive applications, making prompt engineering more efficient. It is designed to develop diverse applications powered by language models more effortlessly, including [chatbots](https://aws.amazon.com/what-is/chatbot/), question-answering, content generation, summarizers, and more.

The following sections describe benefits of LangChain.

**Repurpose language models**

With LangChain, organizations can repurpose LLMs for domain-specific applications without retraining or fine-tuning. Development teams can build complex applications referencing proprietary information to augment model responses. For example, you can use LangChain to build applications that read data from stored internal documents and summarize them into conversational responses. You can create a Retrieval Augmented Generation (RAG) workflow that introduces new information to the language model during prompting. Implementing context-aware workflows like RAG reduces model hallucination and improves response accuracy.

**Simplify AI development**

LangChain simplifies [artificial intelligence (AI)](https://aws.amazon.com/what-is/artificial-intelligence/) development by abstracting the complexity of data source integrations and prompt refining. Developers can customize sequences to build complex applications quickly. Instead of programming business logic, software teams can modify templates and libraries that LangChain provides to reduce development time.

**Developer support**

LangChain provides AI developers with tools to connect language models with external data sources. It is open-source and supported by an active community. Organizations can use LangChain for free and receive support from other developers proficient in the framework.

How does LangChain work?

With LangChain, developers can adapt a language model flexibly to specific business contexts by designating steps required to produce the desired outcome.

**Chains**

*Chains* are the fundamental principle that holds various AI components in LangChain to provide context-aware responses. A chain is a series of automated actions from the user's query to the model's output. For example, developers can use a chain for:

* Connecting to different data sources.
* Generating unique content.
* Translating multiple languages.
* Answering user queries.

**Links**

Chains are made of *links*. Each action that developers string together to form a chained sequence is called a link. With links, developers can divide complex tasks into multiple, smaller tasks. Examples of links include:

* Formatting user input.
* Sending a query to an LLM.
* Retrieving data from cloud storage.
* Translating from one language to another.

In the LangChain framework, a link accepts input from the user and passes it to the LangChain libraries for processing. LangChain also allows link reordering to create different AI workflows.

**Overview**

To use LangChain, developers install the framework in Python with the following command:

*pip install langchain*

Developers then use the chain building blocks or LangChain Expression Language (LCEL) to compose chains with simple programming commands. The *chain()* function passes a link's arguments to the libraries. The*execute()* command retrieves the results. Developers can pass the current link result to the following link or return it as the final output.

Below is an example of a chatbot chain function that returns product details in multiple languages.

*chain([*

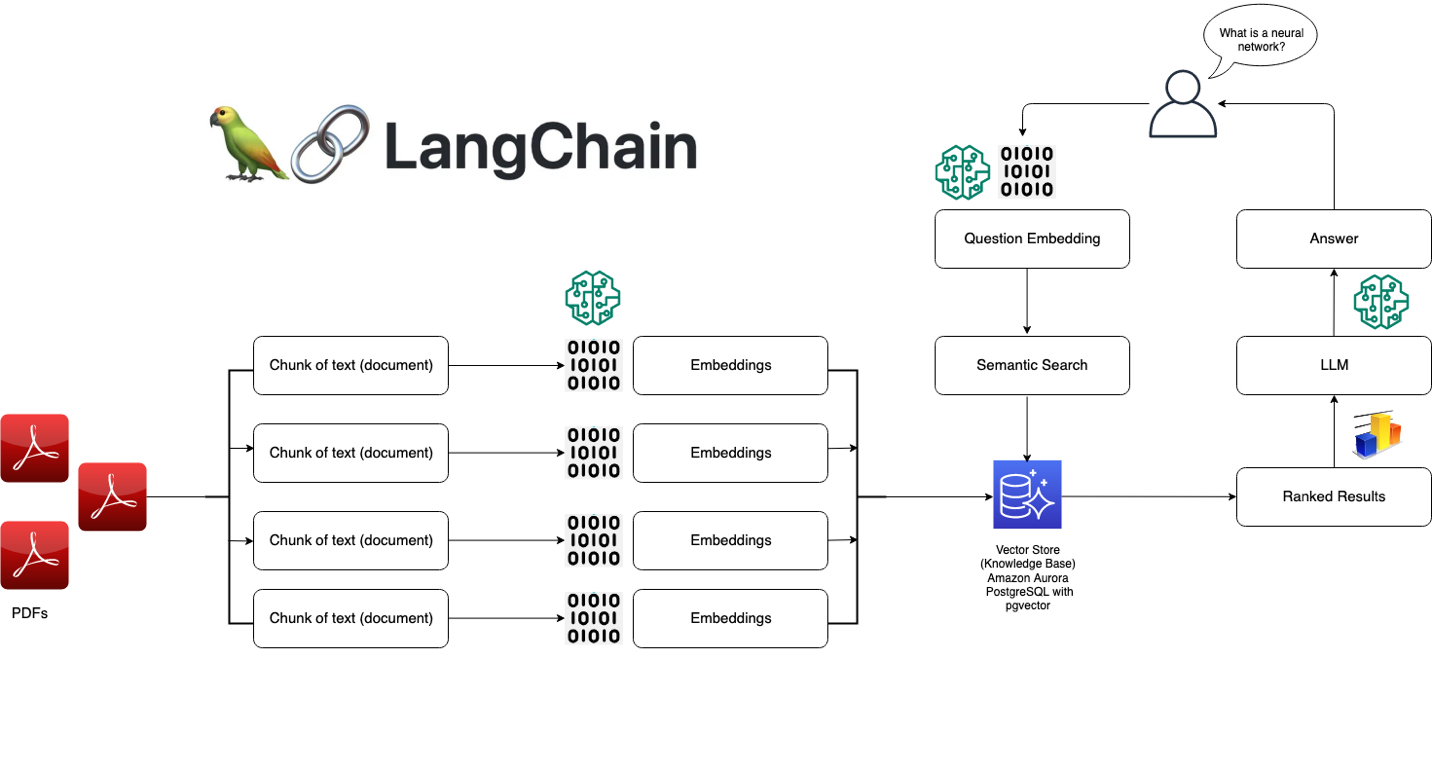
*retrieve\_data\_from\_product\_database().*

*send\_data\_to\_language\_model().*

*format\_output\_in\_a\_list().*

*translate\_output\_in\_target\_language()*

*])*



**What are the core components of LangChain?**

Using LangChain, software teams can build context-aware language model systems with the following modules.

**LLM interface**

LangChain provides APIs with which developers can connect and query LLMs from their code. Developers can interface with public and proprietary models like GPT, Bard, and PaLM with LangChain by making simple API calls instead of writing complex code.

**Prompt templates**

*Prompt templates* are pre-built structures developers use to consistently and precisely format queries for AI models. Developers can create a prompt template for chatbot applications, few-shot learning, or deliver specific instructions to the language models. Moreover, they can reuse the templates across different applications and language models.

**Agents**

Developers use tools and libraries that LangChain provides to compose and customize existing chains for complex applications. An *agent* is a special chain that prompts the language model to decide the best sequence in response to a query. When using an agent, developers provide the user's input, available tools, and possible intermediate steps to achieve the desired results. Then, the language model returns a viable sequence of actions the application can take.

**Retrieval modules**

LangChain enables the architecting of RAG systems with numerous tools to transform, store, search, and retrieve information that refine language model responses. Developers can create semantic representations of information with word embeddings and store them in local or cloud vector databases.

**Memory**

Some conversational language model applications refine their responses with information recalled from past interactions. LangChain allows developers to include memory capabilities in their systems. It supports:

* Simple memory systems that recall the most recent conversations.
* Complex memory structures that analyze historical messages to return the most relevant results.

**Callbacks**

Callbacks are codes that developers place in their applications to log, monitor, and stream specific events in LangChain operations. For example, developers can track when a chain was first called and errors encountered with callbacks

**Applications of LangChain**

LangChain is a powerful tool that can be used to build a wide range of LLM-powered applications. It is simple to use and has a large user and contributor community.

* **Conversational Agents**: Build chatbots and virtual assistants that can engage in meaningful, context-aware conversations with users.
* **Document Summarization**: Automatically generate summaries of long documents, making it easier for users to digest large amounts of information.
* **Question Answering (Chatbot)**: Create systems that can answer questions based on a given context or a set of documents.
* **Workflow Automation**: Design workflows that involve multiple steps, such as data extraction, processing, and reporting, all powered by language models.
* **Content Generation**: Generate creative content, such as articles, stories, or marketing copy, with the help of language models.

here’s the **clean LangChain map** exactly in the buckets you asked:

**1) Models**

* **LLMs / Chat models**: ChatOpenAI, ChatAnthropic, local LLMs (HF).
* **Embeddings**: OpenAIEmbeddings, HuggingFaceEmbeddings.
* (Optional) **Other modalities**: vision, audio models (varies by provider).

**2) Prompts**

* **Templates**: PromptTemplate, ChatPromptTemplate (system/human/ai messages).
* **Examples**: few-shot + example selectors.
* **Output control**: structured output / output parsers (JSON schema, regex).

**3)** **Indexes (RAG plumbing)**

* **Loaders**: PDFs/CSV/HTML/web/DB via langchain\_community.document\_loaders.
* **Splitters**: RecursiveCharacterTextSplitter, token/semantic splitters.
* **Vector stores**: Chroma, FAISS, Pinecone, Weaviate, Milvus.
* **Retrievers**: .as\_retriever() with k, MMR; multi-query, ensemble, compression/rerank.

**4) Chains**

* **Basic**: LLMChain (prompt → model).
* **Retrieval**: RetrievalQA, ConversationalRetrievalChain.
* **Doc chains**: map-reduce / refine / stuff.
* **LCEL (Runnable graph)**: RunnableSequence, branching, parallel, retries, fallbacks, streaming.

**5) Memory**

* **ConversationBufferMemory** (all turns), **BufferWindowMemory** (last k).
* **ConversationSummaryMemory** (token-efficient).
* **EntityMemory**, **VectorStoreRetrieverMemory** (long-term recall).
* **Backends**: in-memory, Redis, SQL, etc.

**6) Agents**

* **Tools**: search, calculator, code, HTTP, SQL, custom tools.
* **Policies/Types**: tool-calling (functions), ReAct, plan-and-execute.
* **Runtime**: AgentExecutor (+ create\_tool\_calling\_agent / ReAct builders).

*“LangChain is a framework for developing applications powered by language models. It enables applications that are:*

***Data-aware****: connect a language model to other sources of data*

***Agentic****: allow a language model to interact with its environment”*

**What are the main Props of LangChain?**

1. **Components**: a set of implementations for each abstraction, together with abstractions for working with language models. Regardless of whether you use the rest of the LangChain framework, components are flexible and simple to use.
2. **Off-the-shelf chains:** a planned arrangement of parts for carrying out particular higher-level tasks, mostly in a sequential order.

**Modules Provided By LangChat**

[***Model I/O***](https://python.langchain.com/docs/modules/model_io/)[*​*](https://python.langchain.com/docs/get_started/introduction#model-io)

*Interface with language models*

[***Data connection***](https://python.langchain.com/docs/modules/data_connection/)[*​*](https://python.langchain.com/docs/get_started/introduction#data-connection)

*Interface with application-specific data*

[***Chains***](https://python.langchain.com/docs/modules/chains/)[*​*](https://python.langchain.com/docs/get_started/introduction#chains)

*Construct sequences of calls*

[***Agents***](https://python.langchain.com/docs/modules/agents/)[*​*](https://python.langchain.com/docs/get_started/introduction#agents)

*Let chains choose which tools to use given high-level directives*

[***Memory***](https://python.langchain.com/docs/modules/memory/)[*​*](https://python.langchain.com/docs/get_started/introduction#memory)

*Persist application state between runs of a chain*

[***Callbacks***](https://python.langchain.com/docs/modules/callbacks/)[*​*](https://python.langchain.com/docs/get_started/introduction#callbacks)

*Log and stream intermediate steps of any chain*

**Use Cases Of LangChain**

LangChain can be used for a variety of use cases including building:

1. ChatBots
2. Analyzing strucured data
3. Answering questions using resources
4. So much more

**Variety Of Large Language Models**

LangChain allows you to build Language model based applications and give you the option to use a variety of **LLM**(Large Language Models). You can choose to use ChatGPT, Hugging face amongst other LLMs.

In this article we’ll only use ChatGPT3.5 so go ahead and create an account and get an API ke.

Installation Of Requirements

We’ll need a couple of libraries to use as we go along. Run the following to install them.

Install the bare minimum to get started

# pip install langchain

Installing OpenAI module in Python, this will allow us to communicate with the APIs that is used by ChatGPT

# pip install openai

Installing decouple, a Python library that will enable us to read environment variables from the **.env** file will be create in a few.

# pip install python-decouple

**Building A Predictive Model With OpenAI**

The code below is to build a simple prediction model using LangChain and OpenAI. We’ll simple create a basic LLM that takes in text and generates a prediction or a simple output.

I’ll be using a **.env**file to store my OpenAI API keys, create a simple .env file in your directory and add the following

OPANAI\_API\_KEY="sk-\*\*\*\*"

The text in qoutes is your OpenAI API key.

Below is the code to generate the simple prediction model.

from langchain.llms import OpenAI  
# decouple to read .env variables(OpenAI Key)  
from decouple import config  
# import openAI from langChain  
from langchain.llms import OpenAI  
  
# instantiate the OpenAI intance  
llm = OpenAI(openai\_api\_key=config("OPANAI\_API\_KEY"))  
  
# make a prediction  
prediction = llm.predict("What is the largest city in Africa?")  
  
# print the prediction  
print(prediction)

When you execute this file, you should get an output!!

Get Prince Krampah’s stories in your inbox

Join Medium for free to get updates from this writer.

Subscribe

Congrats for creating your first LLM program.

**Chat Models**

Chat models are a variation of language models and use language models under the hood to perform text completion and other tasks. There are mainly four chat models that is provided by LangChain they are:

1. AIMessage
2. HumanMessage
3. SystemMessage
4. hatMessage -- ChatMessage

The most commonly used chat models are AIMessage, HumanMessage and SystemMessage.

Let’s write a simple chat model:

#### Chat Models ####  
from decouple import config  
  
from langchain.chat\_models import ChatOpenAI  
from langchain.schema import (  
 AIMessage,  
 HumanMessage,  
 SystemMessage  
)  
  
chat\_model: ChatOpenAI = ChatOpenAI(openai\_api\_key=config("OPANAI\_API\_KEY"))  
  
prediction\_msg: dict = chat\_model.predict\_messages(  
 [HumanMessage(content="When was the end of the first world war?"),   
SystemMessage(content="Talk like you are a German, keep your responses below 50 words but more than 40 words.")])  
  
print(prediction\_msg)  
print(prediction\_msg.content)

System messages are used to pass messages to the model on how to behave. Example, let’s say we are creating a chatBot to act as a voice assistant. The system message will be something like:

*“Your name is Helen, you are a voice assistant to John. You’ll take care of reminding him of his daily activities. Use polite responses and you tone of language should include some humour”*

**Prompts**

The majority of LLM applications do not immediately feed user input into an LLM. The user input is typically included to a longer text document called a **prompt template** that gives further background information on the current task or activity needed to be performed.

from decouple import config  
from langchain.llms import OpenAI  
  
# import PromptTemplate  
from langchain.prompts import PromptTemplate  
  
llm = OpenAI(openai\_api\_key=config("OPANAI\_API\_KEY"))  
  
  
prompt = PromptTemplate.from\_template(  
 "What is the name of the capital city of {country}?")  
prompt\_formatted\_str: str = prompt.format(country="United States")  
  
  
print(prompt\_formatted\_str)  
  
prediction: str = llm.predict(prompt\_formatted\_str)  
  
print(prediction)

**Prompts with chat models**

from decouple import config  
from langchain.prompts.chat import (  
 ChatPromptTemplate,  
 SystemMessagePromptTemplate,  
 HumanMessagePromptTemplate,  
)  
from langchain.chat\_models import ChatOpenAI  
  
sys\_template: str = """You are a language translater, an English speaker wants to translate {original\_sentence} to {desired\_language}. Tell him the corrent answer."""  
system\_message\_prompt = SystemMessagePromptTemplate.from\_template(sys\_template)  
  
student\_template = "Translate {original\_sentence} to {desired\_language}"  
student\_message\_prompt = HumanMessagePromptTemplate.from\_template(  
 student\_template)  
  
chat\_prompt = ChatPromptTemplate.from\_messages(  
 [system\_message\_prompt, student\_message\_prompt])  
  
chat\_prompt\_format = chat\_prompt.format\_messages(  
 original\_sentence="I love Pizza!", desired\_language="French")  
  
print(chat\_prompt\_format)  
  
chat\_model: ChatOpenAI = ChatOpenAI(openai\_api\_key=config("OPANAI\_API\_KEY"))  
  
  
prediction\_msg: dict = chat\_model.predict\_messages(messages=chat\_prompt\_format)  
  
print("#######################################")  
print(prediction\_msg.content)  
print("#######################################")

**Chain**

Now that we have a model and a prompt template, we should put them both together. Models, prompts, and other primitives like chains can all be linked together using chains.

**Chains with simple LLM**

#### Prompts ####  
from decouple import config  
from langchain.llms import OpenAI  
from langchain.chains import LLMChain  
  
# import PromptTemplate  
from langchain.prompts import PromptTemplate  
  
llm = OpenAI(openai\_api\_key=config("OPANAI\_API\_KEY"))  
  
  
prompt = PromptTemplate.from\_template(  
 "What is the name of the capital city of {country}?")  
  
# we do not need this any more  
# prompt\_formatted\_str: str = prompt.format(country="United States")  
# print(prompt\_formatted\_str)  
  
chain = LLMChain(llm=llm, prompt=prompt)  
prediction: str = chain.run(country="United States")  
  
print(prediction)

**Chains with Chat models**

#### Chains wiht Chat models ####  
#### Chat models with Prompts ####  
from decouple import config  
from langchain.prompts.chat import (  
 ChatPromptTemplate,  
 SystemMessagePromptTemplate,  
 HumanMessagePromptTemplate,  
)  
from langchain.chat\_models import ChatOpenAI  
from langchain.chains import LLMChain  
  
sys\_template: str = """You are a language translater, an English speaker wants to translate {original\_sentence} to {desired\_language}. Tell him the corrent answer."""  
system\_message\_prompt = SystemMessagePromptTemplate.from\_template(sys\_template)  
  
student\_template = "Translate {original\_sentence} to {desired\_language}"  
student\_message\_prompt = HumanMessagePromptTemplate.from\_template(  
 student\_template)  
  
chat\_prompt = ChatPromptTemplate.from\_messages(  
 [system\_message\_prompt, student\_message\_prompt])  
  
# we do not longer need this anymore  
# chat\_prompt\_format = chat\_prompt.format\_messages(  
# original\_sentence="I love Pizza!", desired\_language="French")  
# print(chat\_prompt\_format)  
  
chat\_model: ChatOpenAI = ChatOpenAI(openai\_api\_key=config("OPANAI\_API\_KEY"))  
  
# we do not need this any more  
# prediction\_msg: dict = chat\_model.predict\_messages(messages=chat\_prompt\_format)  
  
# Create the LLM chain  
chain: LLMChain = LLMChain(llm=chat\_model, prompt=chat\_prompt)  
  
prediction\_msg: dict = chain.run(  
 original\_sentence="I love Pizza!", desired\_language="French")  
  
print("#######################################")  
print(prediction\_msg)  
print("#######################################")

**Agents**

Our initial chain followed a pre-planned path. We must be able to choose actions on the fly based on inputs in order to manage complex workflows.

In order to choose which actions to take and in what order, agents do just this by using a language model. Agents have access to a variety of tools, and they alternate between selecting one, running it, and evaluating the results until they reach a conclusion.

To Use an agent we first need to create the agent, to load an agent, you need to choose a couple of things:

1. **LLM/Chain:** Depends on what you want to do, we’ll go over both of them
2. **Tools:** Something that performs a specific task at hand. There are a variety of tools at hand: Google search tools, Database look up tools, Python REPL too and so much more.
3. **Agent name:**This is what the official documentation has to say about agent names:

*“A string that references a supported agent class. An agent class is largely parameterized by the prompt the language model uses to determine which action to take. Because this notebook focuses on the simplest, highest level API, this only covers using the standard supported agents. If you want to implement a custom agent, see*[*here*](https://python.langchain.com/docs/modules/agents/how_to/custom_agent.html)*. For a list of supported agents and their specifications, see*[*here*](https://python.langchain.com/docs/modules/agents/agent_types/)*.”*

We’ll be using the Serper API to make like queries or searches on the internet incase our agents needs to query any live information we are interested in. Install this library using:

# pip install google-search-results

Signup for an API key on serpAPI [here](https://serpapi.com/)

Once you you get the api key, you need to add it to your environment variables as follows:

# export SERPAPI\_API\_KEY="[Your API key goes here]"

Let’s write a simple program using an agent

from langchain.agents import AgentType, initialize\_agent, load\_tools  
from langchain.llms import OpenAI  
from decouple import config  
# create LLM model  
llm = OpenAI(temperature=0, openai\_api\_key=config("OPANAI\_API\_KEY"))  
  
  
# specify tools you want to use  
tools = load\_tools(["serpapi", "llm-math"], llm=llm)  
  
# initialize the agent  
agent = initialize\_agent(  
 tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True)  
  
  
# run agent  
agent\_result = agent.run(  
 "What was the lowest temperature in Nairobi yesterday in Fahrenheit?")  
  
print(agent\_result)

**Memory**

Although the chains and agents we’ve seen so far are stateless, many applications need that we take a look back at previous interactions. With a chatbot, for instance, this is unquestionably the situation because you need it to comprehend fresh messages in light of previous ones.

You can preserve application state with the help of the Memory module. The fundamental Memory interface is straightforward: utilizing the most recent run inputs and outputs, it allows you to update state, and it also allows you to change (or contextualize) the incoming data.

Many different memory systems are integrated. The simplest of them is a buffer memory, which does nothing more complicated than prefix the most recent inputs and outputs to the input being used at the moment.

from langchain import OpenAI, ConversationChain  
from decouple import config  
import time  
  
llm = OpenAI(temperature=0, openai\_api\_key=config("OPANAI\_API\_KEY"))  
  
# Conversation  
conversation = ConversationChain(  
 llm=llm,  
 verbose=True  
)  
  
conversation.run("How are you doing today?")  
time.sleep(10)  
conversation.run("I would like to know what is the largest country on Earth.")  
time.sleep(10)  
conversation.run("Thank you and have a good day.")

**Adding Memory**

from langchain.prompts import (  
 ChatPromptTemplate,  
 MessagesPlaceholder,  
 SystemMessagePromptTemplate,  
 HumanMessagePromptTemplate  
)  
from langchain.chains import ConversationChain  
from langchain.chat\_models import ChatOpenAI  
from langchain.memory import ConversationBufferMemory  
import time  
  
prompt = ChatPromptTemplate.from\_messages([  
 SystemMessagePromptTemplate.from\_template(  
 "Your name is hilda, you are a historian, very talkative and smart "  
 "Answer the questions you are being asked and have a sense of humour "  
 "You should be kind and polite in your responses"  
 ),  
 MessagesPlaceholder(variable\_name="history"),  
 HumanMessagePromptTemplate.from\_template("{input}")  
])  
  
llm = ChatOpenAI(temperature=0)  
memory = ConversationBufferMemory(return\_messages=True)  
conversation = ConversationChain(memory=memory, prompt=prompt, llm=llm)  
  
conversation.predict(input="Hi there!")  
time.sleep(5)  
conversation.predict(  
 input="Can you tell me about the history of the first world war, who are the winners you think?")  
time.sleep(5)  
conversation.predict(input="Hi there!")

**Conclusion**

This is all I have for you today, congratulations for making it this far. If you enjoy this content and would love more of it, follow me on [YouTube](https://www.youtube.com/c/CodeWithPrince) at [Code with Prince](https://www.youtube.com/c/CodeWithPrince).

Happy coding!

**Tiny end-to-end sketch (uses all six)**

# MODELS

from langchain\_openai import ChatOpenAI, OpenAIEmbeddings

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

# PROMPTS

from langchain.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from\_messages([

("system", "You answer using the provided context only."),

("human", "{question}\n\nContext:\n{context}")

])

# INDEXES (load → split → embed → vector store → retriever)

from langchain\_community.document\_loaders import TextLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain\_community.vectorstores import Chroma

docs = TextLoader("my\_notes.txt").load()

chunks = RecursiveCharacterTextSplitter(chunk\_size=800, chunk\_overlap=100).split\_documents(docs)

vec = Chroma.from\_documents(chunks, OpenAIEmbeddings())

retriever = vec.as\_retriever(search\_kwargs={"k": 4})

# CHAINS (retrieval + prompt + model) with LCEL

from langchain.schema.runnable import RunnableLambda, RunnablePassthrough

def format\_context(q):

ctx = retriever.get\_relevant\_documents(q)

return "\n\n".join(d.page\_content for d in ctx)

rag\_chain = (

{"question": RunnablePassthrough(), "context": RunnableLambda(format\_context)}

| prompt | llm

)

# MEMORY (plug into a chat loop if you want ongoing context)

from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory(return\_messages=True)

# AGENTS (optional): expose the retriever as a Tool

from langchain.agents import Tool, AgentExecutor, create\_tool\_calling\_agent

tools = [Tool(name="doc\_search", func=lambda q: format\_context(q), description="Search project notes")]

agent\_prompt = ChatPromptTemplate.from\_messages([

("system", "You are helpful. Use tools when needed."),

("human", "{input}")

])

agent = create\_tool\_calling\_agent(llm, tools, agent\_prompt)

agent\_exec = AgentExecutor(agent=agent, tools=tools, verbose=False)

# Use either:

answer = rag\_chain.invoke("What did we decide about Q3 targets?")

# or with tools/agent:

agent\_answer = agent\_exec.invoke({"input": "Find Q3 targets from notes and summarize."})

print(answer.content)

If you want, I can turn this into a **starter repo layout** (env, file structure, requirements) or swap in **Pinecone/FAISS** and a **summary memory** for longer chats.

**Create requirements.txt**

txt

CopyEdit

langchain

langchain-openai

langchain-community

chromadb

matplotlib

openai

**2️⃣ Create my\_notes.txt (sample doc)**

txt

CopyEdit

In Q1 2024, our company launched two new products that exceeded sales expectations.

In Q2 2024, we expanded into three new markets: India, Brazil, and South Korea.

Customer satisfaction increased by 15% due to better support response times.

In Q3 2024, we plan to focus on AI-powered product recommendations.

In Q4 2024, our target is to increase revenue by 20% compared to Q4 2023.

**3️⃣ Create rag\_demo.py**

python

CopyEdit

# -------------------------

# 📌 LangChain RAG Example with Chart Visualization

# -------------------------

# --- 1. Imports ---

# LangChain model & embeddings

from langchain\_openai import ChatOpenAI, OpenAIEmbeddings

# For prompts

from langchain.prompts import ChatPromptTemplate

# For document loading & splitting

from langchain\_community.document\_loaders import TextLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

# Vector database

from langchain\_community.vectorstores import Chroma

# LCEL (LangChain Expression Language) utilities

from langchain.schema.runnable import RunnableLambda, RunnablePassthrough

# Chart plotting

import matplotlib.pyplot as plt

# To store API key securely

import os

# --- 2. Set API key ---

# Replace 'your-openai-api-key' with your real key or set via environment variable

os.environ["OPENAI\_API\_KEY"] = "your-openai-api-key"

# --- 3. Load the model ---

# ChatOpenAI: a chat-optimized LLM (using GPT-4o-mini here for cost/speed)

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

# --- 4. Create a prompt template ---

# This template has two placeholders: {question} and {context}

prompt = ChatPromptTemplate.from\_messages([

("system", "You are a helpful assistant. Use ONLY the provided context to answer."),

("human", "{question}\n\nContext:\n{context}")

])

# --- 5. Load the document ---

loader = TextLoader("my\_notes.txt")

docs = loader.load() # List[Document]

# --- 6. Split into chunks ---

# RecursiveCharacterTextSplitter tries to break text at paragraph/sentence boundaries

splitter = RecursiveCharacterTextSplitter(chunk\_size=50, chunk\_overlap=10)

chunks = splitter.split\_documents(docs)

# --- 📊 Chart 1: Chunk length distribution ---

chunk\_lengths = [len(doc.page\_content) for doc in chunks]

plt.bar(range(len(chunk\_lengths)), chunk\_lengths)

plt.title("Chunk Lengths")

plt.xlabel("Chunk Index")

plt.ylabel("Number of Characters")

plt.show()

# --- 7. Create embeddings & store in vector DB ---

embeddings = OpenAIEmbeddings()

vector\_store = Chroma.from\_documents(chunks, embeddings)

# --- 8. Create retriever ---

# Retriever finds top-k similar chunks for a given query

retriever = vector\_store.as\_retriever(search\_kwargs={"k": 3})

# --- 9. Define a helper to format context ---

def format\_context(query):

# Get relevant docs

relevant\_docs = retriever.get\_relevant\_documents(query)

# Join their content into one string

return "\n\n".join(d.page\_content for d in relevant\_docs)

# --- 📊 Chart 2: Query relevance heatmap ---

# For visualization, let's embed each chunk and query, then plot similarities

import numpy as np

query = "What are our plans for Q3 2024?"

query\_vec = embeddings.embed\_query(query)

chunk\_vecs = [embeddings.embed\_query(doc.page\_content) for doc in chunks]

similarities = [np.dot(query\_vec, vec) / (np.linalg.norm(query\_vec) \* np.linalg.norm(vec)) for vec in chunk\_vecs]

plt.bar(range(len(similarities)), similarities)

plt.title("Query-Chunks Similarity")

plt.xlabel("Chunk Index")

plt.ylabel("Cosine Similarity")

plt.show()

# --- 10. Build RAG chain ---

rag\_chain = (

{"question": RunnablePassthrough(), "context": RunnableLambda(format\_context)}

| prompt

| llm

)

# --- 11. Run a query ---

question = "What are our plans for Q3 2024?"

answer = rag\_chain.invoke(question)

# --- 12. Print the answer ---

print("\nQUESTION:", question)

print("ANSWER:", answer.content)

**4️⃣ How This Works Step-by-Step**

1. **Load your document** (my\_notes.txt).
2. **Split** into small chunks (good for embeddings).
3. **Visualize** chunk sizes with a bar chart.
4. **Embed & store** chunks in Chroma vector DB.
5. **Retrieve** top matches for a query using cosine similarity.
6. **Plot** similarity scores between the query and each chunk.
7. **Build a RAG chain** = Query → Retrieve context → Prompt → Model.
8. **Run & print** the final LLM answer.

**5️⃣ Running the Script**

pip install -r requirements.txt

python rag\_demo.py

* First chart: shows how text is chunked.
* Second chart: shows which chunks are most similar to your question.
* Console: shows the RAG answer from the LLM.

If you want, I can **extend this** so you can **chat continuously with memory** and see a **conversation timeline chart** where similarity changes over time. That would make it even more intuitive.

Yes please

let’s **extend** the above RAG example so it becomes a **chatbot with memory** and a **timeline chart** showing how relevant chunks change over the conversation.

This will make it more **intuitive** for beginners because you’ll:

* See **how the retrieval context changes** as you ask more questions
* Get a visual **conversation history timeline**

**📂 Updated Project Structure**

my\_langchain\_rag/

│

├── rag\_chat\_demo.py # Extended chatbot script with memory + chart

├── my\_notes.txt # Sample document

└── requirements.txt # Dependencies

**rag\_chat\_demo.py**

# -------------------------

# 📌 LangChain RAG Chatbot with Memory & Similarity Timeline

# -------------------------

# --- 1. Imports ---

from langchain\_openai import ChatOpenAI, OpenAIEmbeddings

from langchain.prompts import ChatPromptTemplate

from langchain\_community.document\_loaders import TextLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain\_community.vectorstores import Chroma

from langchain.schema.runnable import RunnableLambda, RunnablePassthrough

from langchain.memory import ConversationBufferMemory

from langchain.agents import Tool, AgentExecutor, create\_tool\_calling\_agent

import matplotlib.pyplot as plt

import numpy as np

import os

# --- 2. Set API key ---

os.environ["OPENAI\_API\_KEY"] = "your-openai-api-key" # replace with real key

# --- 3. Load the model ---

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

# --- 4. Create a prompt template ---

prompt = ChatPromptTemplate.from\_messages([

("system", "You are a helpful assistant. Use ONLY the provided context to answer."),

("human", "{question}\n\nContext:\n{context}")

])

# --- 5. Load and split documents ---

loader = TextLoader("my\_notes.txt")

docs = loader.load()

splitter = RecursiveCharacterTextSplitter(chunk\_size=50, chunk\_overlap=10)

chunks = splitter.split\_documents(docs)

# --- 📊 Chart 1: Chunk length distribution ---

chunk\_lengths = [len(doc.page\_content) for doc in chunks]

plt.bar(range(len(chunk\_lengths)), chunk\_lengths)

plt.title("Chunk Lengths")

plt.xlabel("Chunk Index")

plt.ylabel("Number of Characters")

plt.show()

# --- 6. Create embeddings & store in vector DB ---

embeddings = OpenAIEmbeddings()

vector\_store = Chroma.from\_documents(chunks, embeddings)

retriever = vector\_store.as\_retriever(search\_kwargs={"k": 3})

# --- 7. Helper for retrieving context ---

def format\_context(query):

relevant\_docs = retriever.get\_relevant\_documents(query)

return "\n\n".join(d.page\_content for d in relevant\_docs)

# --- 8. Memory setup ---

memory = ConversationBufferMemory(return\_messages=True)

# --- 9. Track similarity changes over chat ---

similarity\_timeline = [] # list of lists (per question)

def log\_similarity(query):

query\_vec = embeddings.embed\_query(query)

chunk\_vecs = [embeddings.embed\_query(doc.page\_content) for doc in chunks]

similarities = [np.dot(query\_vec, vec) / (np.linalg.norm(query\_vec) \* np.linalg.norm(vec)) for vec in chunk\_vecs]

similarity\_timeline.append(similarities)

# --- 10. Build RAG chain ---

rag\_chain = (

{"question": RunnablePassthrough(), "context": RunnableLambda(format\_context)}

| prompt

| llm

)

# --- 11. Chat loop ---

questions = [

"What are our plans for Q3 2024?",

"Which markets did we expand into in Q2 2024?",

"What is the revenue target for Q4 2024?"

]

for q in questions:

log\_similarity(q) # log similarity for chart

answer = rag\_chain.invoke(q)

memory.chat\_memory.add\_user\_message(q)

memory.chat\_memory.add\_ai\_message(answer.content)

print(f"\nUSER: {q}")

print(f"BOT: {answer.content}")

# --- 📊 Chart 2: Similarity timeline ---

similarity\_array = np.array(similarity\_timeline) # shape: (num\_questions, num\_chunks)

plt.figure(figsize=(8, 5))

for chunk\_idx in range(similarity\_array.shape[1]):

plt.plot(similarity\_array[:, chunk\_idx], marker='o', label=f"Chunk {chunk\_idx}")

plt.title("Chunk Similarity Over Conversation")

plt.xlabel("Question Number")

plt.ylabel("Cosine Similarity")

plt.legend()

plt.show()

**💡 What’s New vs. the First Script**

1. **Conversation Memory** (ConversationBufferMemory)
   * Stores user & AI messages so you can build a continuous conversation.
   * We don’t yet feed it back into the RAG chain (kept simple for now), but you can.
2. **Similarity Timeline Chart**
   * Each time you ask a question, we measure cosine similarity between the query and each document chunk.
   * This shows **which chunks were most relevant at each step** in your conversation.
3. **Multi-turn Questions**
   * Three example questions so you can see the chart change over time.

**🖼 How the Charts Help**

* **Chart 1 (Chunk Lengths)** → Visualizes how your document was split.
* **Chart 2 (Similarity Timeline)** → Shows how the retriever’s focus changes across conversation turns.

If you want, I can **upgrade this again** so:

* The chatbot **remembers previous answers** and uses them in later responses.
* You can **type interactively** in the console instead of a fixed list of questions.
* The similarity timeline updates **live** as you chat.

Do you want me to make that interactive + memory-powered version next?

Yes please go ahead

**Conversation Chatbot APP -1**

Nice — below is a **single, fully runnable script** that builds a **RAG chatbot with live conversation memory** and **interactive charts** that update as you chat. I’ve written **detailed explanations for each block and key line** so a beginner can follow along.

⚠️ Prerequisites you must do before running:

1. Install dependencies (shown below).
2. Set your OpenAI API key in the environment: export OPENAI\_API\_KEY="sk-..." (Linux/macOS) or setx OPENAI\_API\_KEY "sk-..." (Windows).
3. Put my\_notes.txt (sample content shown below) in the same folder as the script.

**requirements.txt**

nginx

langchain

openai

chromadb

matplotlib

numpy

**my\_notes.txt (sample - keep this in the project folder)**

vbnet

In Q1 2024, our company launched two new products that exceeded sales expectations.

In Q2 2024, we expanded into three new markets: India, Brazil, and South Korea.

Customer satisfaction increased by 15% due to better support response times.

In Q3 2024, we plan to focus on AI-powered product recommendations.

In Q4 2024, our target is to increase revenue by 20% compared to Q4 2023.

**rag\_chat\_interactive.py (full script with explanations)**

python

"""

RAG Chatbot (interactive) with Conversation Memory + Live Similarity Timeline

Run: python rag\_chat\_interactive.py

"""

# -------------------------

# 1) Imports

# -------------------------

# langchain imports for LLMs, embeddings, chains, memory, loaders, splitters, vectorstore

from langchain.chat\_models import ChatOpenAI

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.document\_loaders import TextLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain.vectorstores import Chroma

from langchain.chains import ConversationalRetrievalChain

from langchain.memory import ConversationBufferMemory

# plotting and numeric tools

import matplotlib.pyplot as plt

import numpy as np

import os

import time

# -------------------------

# Explanation (Imports)

# -------------------------

# ChatOpenAI -> Chat-optimized LLM wrapper (calls OpenAI chat models)

# OpenAIEmbeddings -> turn text into numeric vectors (embeddings)

# TextLoader -> loads plain text files into LangChain Document objects

# RecursiveCharacterTextSplitter -> splits long text into chunks

# Chroma -> local vector DB (stores embeddings + metadata)

# ConversationalRetrievalChain -> ready-made chain: retrieval + LLM + memory

# ConversationBufferMemory -> keeps chat history (in RAM)

# matplotlib/numpy -> used for charts and similarity calculations

# -------------------------

# 2) API Key (expects env var)

# -------------------------

# The script uses the OPENAI\_API\_KEY environment variable.

# You can set it in your shell BEFORE running the script:

# export OPENAI\_API\_KEY="sk-..." (Linux/macOS)

# setx OPENAI\_API\_KEY "sk-..." (Windows)

if "OPENAI\_API\_KEY" not in os.environ:

raise RuntimeError("Please set OPENAI\_API\_KEY in environment before running.")

# -------------------------

# 3) Load & split documents

# -------------------------

# Load the sample text file into a Document object

loader = TextLoader("my\_notes.txt") # loads file -> list of Document(s)

docs = loader.load() # docs is a list; each item has .page\_content

# Split long documents into smaller chunks suitable for embeddings + retrieval

splitter = RecursiveCharacterTextSplitter(

chunk\_size=120, # aim: ~120 characters per chunk (adjust to your needs)

chunk\_overlap=20 # keep some overlap so context isn't lost at boundaries

)

chunks = splitter.split\_documents(docs) # returns a list of smaller Document objects

# -------------------------

# 4) Plot chunk length distribution (one-time)

# -------------------------

# Show how documents were chunked (useful to inspect chunk sizes)

chunk\_lengths = [len(d.page\_content) for d in chunks]

plt.figure(figsize=(8, 3))

plt.bar(range(len(chunk\_lengths)), chunk\_lengths)

plt.title("Chunk Lengths (characters)")

plt.xlabel("Chunk index")

plt.ylabel("Length (chars)")

plt.tight\_layout()

plt.show(block=False) # non-blocking show so script can continue

time.sleep(1) # short pause so the window renders on some platforms

# -------------------------

# 5) Create embeddings and vectorstore (Chroma)

# -------------------------

# Create an embeddings object (uses your OpenAI key)

embeddings = OpenAIEmbeddings()

# Create a Chroma vectorstore from the chunks: this computes embeddings and saves them locally

# If you re-run often during development, you might want to set persist\_directory to keep data.

vectorstore = Chroma.from\_documents(documents=chunks, embedding=embeddings)

# Create a retriever from the vectorstore (this is what the chain will call)

retriever = vectorstore.as\_retriever(search\_kwargs={"k": 3}) # return top 3 chunks by default

# -------------------------

# 6) Memory & Conversational RAG chain

# -------------------------

# ConversationBufferMemory saves all user and AI messages for the session.

memory = ConversationBufferMemory(memory\_key="chat\_history", return\_messages=True)

# ConversationalRetrievalChain wires up:

# - retriever (vectorstore)

# - chat LLM

# - and optional memory automatically (it will include previous user/assistant messages)

llm = ChatOpenAI(temperature=0.0, model="gpt-4o-mini") # pick model you have access to

chain = ConversationalRetrievalChain.from\_llm(

llm=llm,

retriever=retriever,

memory=memory,

return\_source\_documents=True # we will also show which docs were used

)

# -------------------------

# Explanation (Chain + Memory)

# -------------------------

# ConversationalRetrievalChain expects a dict like {"question": "..."} when invoked.

# It will:

# 1) use the retriever to fetch context chunks for the question

# 2) combine question + retrieved context + chat history (from memory)

# 3) call the LLM to produce an answer

# 4) store the turn in memory automatically

# -------------------------

# 7) Helpers: similarity logging + plotting setup

# -------------------------

# We'll compute cosine similarity between a query and each chunk's embedding so we can chart "which chunk matters"

def cosine\_similarity(a, b):

# a and b are 1D numpy vectors

return float(np.dot(a, b) / (np.linalg.norm(a) \* np.linalg.norm(b) + 1e-12))

# Precompute embeddings for chunks once to speed up similarity checks

chunk\_vecs = [np.array(embeddings.embed\_query(d.page\_content)) for d in chunks]

# Data structure to store similarity timeline: list of arrays; one entry per question

similarity\_timeline = []

# Setup the live plot for similarity timeline

plt.ion() # interactive mode on so we can update plots live

fig, ax = plt.subplots(figsize=(9, 5))

def update\_similarity\_plot(similarity\_timeline):

ax.clear()

if len(similarity\_timeline) == 0:

ax.set\_title("No queries yet - ask a question to see similarities")

ax.set\_xlabel("Query index")

ax.set\_ylabel("Cosine similarity")

plt.pause(0.01)

return

sim\_array = np.array(similarity\_timeline) # shape: (num\_queries, num\_chunks)

num\_queries, num\_chunks = sim\_array.shape

# plot each chunk's similarity across queries as a line

for chunk\_idx in range(num\_chunks):

ax.plot(range(num\_queries), sim\_array[:, chunk\_idx], marker='o', label=f"chunk {chunk\_idx}")

ax.set\_title("Chunk Similarity Over Conversation")

ax.set\_xlabel("Query number (0 = first question)")

ax.set\_ylabel("Cosine similarity")

ax.set\_ylim(0, 1) # cosine similarity range for normalized vectors is -1..1 but embeddings typically non-negative in practice

ax.legend(loc='upper right', fontsize='small', ncol=2)

fig.tight\_layout()

plt.draw()

plt.pause(0.01)

# show initial empty plot

update\_similarity\_plot(similarity\_timeline)

# -------------------------

# 8) Interactive chat loop

# -------------------------

print("\n==== RAG Chat (type 'exit' or 'quit' to stop) ====\n")

while True:

user\_q = input("You: ").strip()

if user\_q.lower() in ("exit", "quit"):

print("Goodbye!")

break

if user\_q == "":

continue

# 1) Compute similarity between query and each chunk (for plotting)

q\_vec = np.array(embeddings.embed\_query(user\_q))

similarities = [cosine\_similarity(q\_vec, cv) for cv in chunk\_vecs]

similarity\_timeline.append(similarities)

# 2) Update the live similarity plot

update\_similarity\_plot(similarity\_timeline)

# 3) Call the conversational retrieval chain

# It accepts {"question": ..., "chat\_history": ...} but memory is attached, so we just pass question

result = chain({"question": user\_q})

# result is a dict like:

# {"answer": "...", "source\_documents": [...], "chat\_history": [...] } (fields may vary)

answer\_text = result.get("answer") or result.get("output\_text") or result

source\_docs = result.get("source\_documents", [])

# 4) Print the answer and which source docs were used

print("\nBot:", answer\_text, "\n")

if source\_docs:

print("Top source chunks used:")

for i, sd in enumerate(source\_docs):

snippet = sd.page\_content.replace("\n", " ").strip()

print(f" [{i}] {snippet[:200]}{'...' if len(snippet) > 200 else ''}")

else:

print("No source documents were returned.")

# 5) Show current conversation memory short summary (last few turns)

# memory.chat\_memory.return\_messages() is available as memory.load\_memory\_variables({})

mem\_var = memory.load\_memory\_variables({}) # returns dict; key is "chat\_history"

# print last 4 messages (user or assistant)

print("\nConversation (last messages):")

chat\_messages = mem\_var.get("chat\_history", [])

# messages are objects; we will print type + text for clarity

for i, msg in enumerate(chat\_messages[-6:]): # last 6 items

# msg could be a ChatMessage object; msg.content has the text typically

content = getattr(msg, "content", str(msg))

role = getattr(msg, "type", None) or getattr(msg, "role", None) or "message"

print(f" [{i}] {role}: {content[:200]}{'...' if len(content) > 200 else ''}")

print("\n--- Ready for next question ---\n")

**Step-by-step notes / explanations (beginner-friendly)**

1. **Imports** — we import LangChain components (LLM wrapper, embeddings, loader, splitter, vectorstore, chain, memory) and matplotlib/numpy for plotting and numeric work.
2. **OPENAI\_API\_KEY** — the script expects the key in the environment. This is the safest simple approach to run the code.
3. **Document loading & splitting** — TextLoader reads the file; RecursiveCharacterTextSplitter splits text into smaller chunks so embeddings are more focused and retrieval works better.
4. **Chunk plot** — we display a one-time bar chart showing how big each chunk is. This helps you tune chunk\_size and chunk\_overlap.
5. **Embeddings & Chroma** — OpenAIEmbeddings() uses OpenAI to compute embeddings; Chroma.from\_documents(...) stores those embeddings locally and provides a retriever.
6. **ConversationalRetrievalChain + Memory** — this chain wires retrieval + LLM + chat memory automatically. When you call chain({"question": user\_q}) it will:
   * fetch top-k similar chunks via the retriever,
   * construct a prompt that includes the retrieved context and chat history,
   * call the chat LLM (ChatOpenAI),
   * return an answer and optionally source documents,
   * and store the turn in memory.
7. **Similarity timeline** — we precompute chunk embeddings once and compute cosine similarities with each user query. We plot these over time so you can see which chunk(s) the retriever is focusing on as conversation progresses.
8. **Interactive loop** — the while-loop reads user input, logs similarities, updates the plot, calls the chain, prints the answer + source chunks + recent memory, and repeats until you type exit or quit.

**How to run**

1. Create a virtual environment and install deps:

bash

python -m venv venv

source venv/bin/activate # or venv\Scripts\activate on Windows

pip install -r requirements.txt

1. Set your OpenAI key:

bash

export OPENAI\_API\_KEY="sk-..." # mac/linux

# or on Windows PowerShell:

# setx OPENAI\_API\_KEY "sk-..."

1. Run the script:

bash

python rag\_chat\_interactive.py

Type natural questions such as:

* What are our plans for Q3 2024?
* Which markets did we expand into?
* What is the revenue target for Q4?

Watch the similarity timeline chart update after each question. The conversation memory will grow — the chain uses memory to keep replies contextual.

**Possible extensions (I can provide any of these if you want)**

* Persist memory to Redis or a file between runs.
* Use a persistent Chroma persist\_directory so embeddings survive restarts.
* Add streaming responses (token-by-token) while the model generates.
* Use a browser UI (FastAPI + simple HTML) instead of console I/O.
* Add Reranking or MMR retriever to improve retrieval quality.

If you want, I can now:

* Provide the **exact code** to persist Chroma to disk and persist memory to Redis (so chat persists across runs), or
* Convert this into a **small FastAPI web UI** so you can chat from your browser and see charts there.

Which of those would you like next?

**Full Stack Project 2**

**Go with the second one**

Nice — I’ll give you a **complete, runnable FastAPI web app** that turns the earlier interactive RAG chatbot into a **browser UI** with:

* Chatbox (send questions)
* Answers returned from a **ConversationalRetrievalChain** (LLM + Chroma retriever)
* Two charts rendered and updated in the browser:
  + Chunk length distribution (one-time)
  + Live similarity timeline across the chat (updates after each query)
* **Chroma persistence to disk** so embeddings survive restarts
* Clear, beginner-friendly explanations and **step-by-step run instructions**

**Project layout (what to create)**

markdown

my\_langchain\_rag\_web/

├── app.py

├── requirements.txt

├── my\_notes.txt

└── templates/

└── index.html

**requirements.txt**

Save this file and install with pip install -r requirements.txt.

css

CopyEdit

fastapi

uvicorn[standard]

jinja2

langchain

openai

chromadb

matplotlib

numpy

python-multipart

**my\_notes.txt (sample)**

Use the same sample or replace with your own notes.

vbnet

CopyEdit

In Q1 2024, our company launched two new products that exceeded sales expectations.

In Q2 2024, we expanded into three new markets: India, Brazil, and South Korea.

Customer satisfaction increased by 15% due to better support response times.

In Q3 2024, we plan to focus on AI-powered product recommendations.

In Q4 2024, our target is to increase revenue by 20% compared to Q4 2023.

**templates/index.html**

Create a folder named templates and put this file inside. This is the simple UI for chat + charts.

html

CopyEdit

<!doctype html>

<html>

<head>

<meta charset="utf-8" />

<title>LangChain RAG Chat (Web)</title>

<style>

body { font-family: Arial, sans-serif; margin: 16px; }

.container { display: flex; gap: 20px; }

.left { flex: 1; min-width: 380px; }

.right { width: 480px; }

.chat { border: 1px solid #ddd; padding: 10px; height: 420px; overflow-y: auto; background:#fafafa; }

.message.user { text-align: right; color: #222; margin: 6px 0; }

.message.bot { text-align: left; color: #004; margin: 6px 0; }

.controls { margin-top: 10px; display:flex; gap:8px; }

textarea { width:100%; height: 70px; }

button { padding: 8px 12px; }

.sources { font-size: 0.9em; color:#333; margin-top:6px; background:#fff; padding:8px; border-radius:6px; border:1px solid #eee }

.chart { border:1px solid #eee; padding:8px; background:#fff; margin-bottom:10px; }

</style>

</head>

<body>

<h2>LangChain RAG Chat (Web)</h2>

<div class="container">

<div class="left">

<div id="chat" class="chat"></div>

<div class="controls">

<textarea id="input" placeholder="Ask something from your documents..."></textarea>

<div style="display:flex;flex-direction:column;">

<button id="send">Send</button>

<button id="clear">Clear</button>

</div>

</div>

<div id="sources" class="sources"></div>

</div>

<div class="right">

<div class="chart">

<h4>Chunk Lengths</h4>

<img id="chunks\_img" src="/chart/chunks.png" alt="chunks" style="width:100%;height:auto"/>

</div>

<div class="chart">

<h4>Similarity Timeline</h4>

<img id="sim\_img" src="/chart/similarity.png" alt="similarity" style="width:100%;height:auto"/>

</div>

</div>

</div>

<script>

async function postQuestion(q) {

const resp = await fetch('/ask', {

method: 'POST',

headers: {'Content-Type':'application/json'},

body: JSON.stringify({question: q})

});

return resp.json();

}

function addMessage(role, text){

const chat = document.getElementById('chat');

const div = document.createElement('div');

div.className = 'message ' + (role==='user' ? 'user' : 'bot');

div.innerText = (role==='user' ? 'You: ' : 'Bot: ') + text;

chat.appendChild(div);

chat.scrollTop = chat.scrollHeight;

}

document.getElementById('send').addEventListener('click', async ()=>{

const ta = document.getElementById('input');

const q = ta.value.trim();

if(!q) return;

addMessage('user', q);

ta.value = '';

document.getElementById('sources').innerText = 'Thinking...';

const data = await postQuestion(q);

addMessage('bot', data.answer || '(no answer)');

// show sources

const srcDiv = document.getElementById('sources');

if(data.source\_snippets && data.source\_snippets.length){

srcDiv.innerHTML = '<b>Top source snippets:</b><ol>' + data.source\_snippets.map(s => `<li>${s}</li>`).join('') + '</ol>';

} else {

srcDiv.innerText = '(no sources returned)';

}

// refresh similarity chart image (add timestamp to bust cache)

document.getElementById('sim\_img').src = '/chart/similarity.png?ts=' + Date.now();

});

document.getElementById('clear').addEventListener('click', async ()=>{

// clear chat in UI and on server

await fetch('/clear', {method:'POST'});

document.getElementById('chat').innerHTML = '';

document.getElementById('sources').innerHTML = '';

document.getElementById('sim\_img').src = '/chart/similarity.png?ts=' + Date.now();

});

</script>

</body>

</html>

**app.py — full FastAPI app (save in project root)**

This is the main server. Save it as app.py. It:

* loads documents, splits them to chunks,
* persists Chroma to disk (so embeddings are reused),
* creates a ConversationalRetrievalChain with memory,
* exposes endpoints:
  + / (UI),
  + POST /ask (send question, returns JSON with answer + source snippets),
  + POST /clear (reset memory),
  + /chart/chunks.png and /chart/similarity.png (dynamic PNG charts).

python

CopyEdit

# app.py

"""

FastAPI + LangChain RAG Web UI

Run:

set OPENAI\_API\_KEY=sk-... # Windows cmd (or use env export on mac/linux)

uvicorn app:app --reload

Then open http://127.0.0.1:8000/

"""

from fastapi import FastAPI, Request, Form

from fastapi.responses import HTMLResponse, JSONResponse, StreamingResponse

from fastapi.templating import Jinja2Templates

import io, os, time

import matplotlib.pyplot as plt

import numpy as np

# LangChain imports

from langchain.chat\_models import ChatOpenAI

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.document\_loaders import TextLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain.vectorstores import Chroma

from langchain.chains import ConversationalRetrievalChain

from langchain.memory import ConversationBufferMemory

# -------------------------

# Basic config & checks

# -------------------------

if "OPENAI\_API\_KEY" not in os.environ:

raise RuntimeError("Set OPENAI\_API\_KEY environment variable before running.")

# fastapi + templates

app = FastAPI()

templates = Jinja2Templates(directory="templates")

# -------------------------

# Load documents and create vector store (persisted)

# -------------------------

DOC\_PATH = "my\_notes.txt"

CHROMA\_DIR = "chroma\_persist" # directory to persist Chroma embeddings

# 1) Load the text documents

loader = TextLoader(DOC\_PATH)

docs = loader.load()

# 2) Split into chunks

splitter = RecursiveCharacterTextSplitter(chunk\_size=120, chunk\_overlap=20)

chunks = splitter.split\_documents(docs)

# 3) Create embeddings and chroma vectorstore (persistent)

embeddings = OpenAIEmbeddings()

vectorstore = Chroma.from\_documents(documents=chunks, embedding=embeddings, persist\_directory=CHROMA\_DIR)

# Persist once (this writes to CHROMA\_DIR)

vectorstore.persist()

# 4) Create retriever

retriever = vectorstore.as\_retriever(search\_kwargs={"k": 3})

# -------------------------

# Create LLM, memory, and chain

# -------------------------

llm = ChatOpenAI(temperature=0.0, model="gpt-4o-mini") # choose the model you have access to

memory = ConversationBufferMemory(memory\_key="chat\_history", return\_messages=True)

chain = ConversationalRetrievalChain.from\_llm(

llm=llm,

retriever=retriever,

memory=memory,

return\_source\_documents=True

)

# -------------------------

# Precompute chunk embeddings for similarity plotting

# -------------------------

# Use the same embedding function to compute chunk vectors once

chunk\_vecs = [np.array(embeddings.embed\_query(d.page\_content)) for d in chunks]

similarity\_timeline = [] # list of lists; append similarity array on each question

# -------------------------

# Helpers: plotting endpoints

# -------------------------

def fig\_to\_png\_bytes(fig):

buf = io.BytesIO()

fig.savefig(buf, format="png", bbox\_inches="tight")

buf.seek(0)

return buf

@app.get("/", response\_class=HTMLResponse)

async def home(request: Request):

return templates.TemplateResponse("index.html", {"request": request})

@app.get("/chart/chunks.png")

def chart\_chunks():

# one-time chart showing chunk length distribution

lengths = [len(d.page\_content) for d in chunks]

fig, ax = plt.subplots(figsize=(6, 2.8))

ax.bar(range(len(lengths)), lengths)

ax.set\_title("Chunk Lengths (characters)")

ax.set\_xlabel("Chunk index")

ax.set\_ylabel("Length (chars)")

fig.tight\_layout()

buf = fig\_to\_png\_bytes(fig)

plt.close(fig)

return StreamingResponse(buf, media\_type="image/png")

@app.get("/chart/similarity.png")

def chart\_similarity():

# dynamic similarity timeline chart

fig, ax = plt.subplots(figsize=(6, 4))

if not similarity\_timeline:

ax.text(0.5, 0.5, "No queries yet.\nAsk something to see similarity timeline.",

ha='center', va='center', fontsize=12)

ax.set\_axis\_off()

else:

sim\_array = np.array(similarity\_timeline) # shape (num\_queries, num\_chunks)

num\_queries, num\_chunks = sim\_array.shape

for chunk\_idx in range(num\_chunks):

ax.plot(range(num\_queries), sim\_array[:, chunk\_idx], marker='o', label=f"chunk {chunk\_idx}")

ax.set\_title("Chunk Similarity Over Conversation")

ax.set\_xlabel("Query number (0 = first question)")

ax.set\_ylabel("Cosine similarity")

ax.set\_ylim(0, 1)

ax.legend(fontsize='small', ncol=2, loc='upper right')

fig.tight\_layout()

buf = fig\_to\_png\_bytes(fig)

plt.close(fig)

return StreamingResponse(buf, media\_type="image/png")

# -------------------------

# Endpoint: ask a question

# -------------------------

@app.post("/ask")

async def ask(req: Request):

body = await req.json()

question = body.get("question", "").strip()

if not question:

return JSONResponse({"error": "empty question"}, status\_code=400)

# 1) compute similarity for plotting

q\_vec = np.array(embeddings.embed\_query(question))

def cosine(a,b): return float(np.dot(a,b)/(np.linalg.norm(a)\*np.linalg.norm(b)+1e-12))

sims = [cosine(q\_vec, cv) for cv in chunk\_vecs]

similarity\_timeline.append(sims)

# 2) call the retrieval chain (it uses memory)

res = chain({"question": question})

# result typically contains 'answer' and 'source\_documents'

answer = res.get("answer") or res.get("output\_text") or str(res)

source\_docs = res.get("source\_documents", [])

# pick up short snippets to show in UI

snippets = []

for sd in source\_docs:

text = sd.page\_content.replace("\n", " ").strip()

snippets.append(text[:300] + ("..." if len(text) > 300 else ""))

return JSONResponse({"answer": answer, "source\_snippets": snippets})

# -------------------------

# Endpoint: clear memory

# -------------------------

@app.post("/clear")

async def clear\_memory():

# Reset conversation memory and similarity timeline while preserving vectorstore

global memory, chain, similarity\_timeline

memory = ConversationBufferMemory(memory\_key="chat\_history", return\_messages=True)

chain = ConversationalRetrievalChain.from\_llm(llm=llm, retriever=retriever, memory=memory, return\_source\_documents=True)

similarity\_timeline = []

return JSONResponse({"ok": True})

**How this app works — quick beginner-friendly breakdown**

1. **Startup**:
   * app.py loads my\_notes.txt using TextLoader.
   * Splits into chunks via RecursiveCharacterTextSplitter.
   * Creates embeddings via OpenAIEmbeddings() and stores them in a **Chroma** directory (chroma\_persist).
   * vectorstore.persist() writes embeddings to disk — so the next run can reuse them (faster & cheaper).
   * Builds a ConversationalRetrievalChain with ChatOpenAI and a ConversationBufferMemory. The chain will:
     + retrieve top-k chunk(s),
     + combine them with chat history,
     + call the chat LLM to produce an answer,
     + store the turn in memory automatically.
2. **UI**:
   * index.html displays chat and two images: chunk lengths and similarity timeline.
   * The similarity chart is updated by the server and the client reloads the image after each query (cache-busted with timestamp query param).
3. **/ask**:
   * Receives a JSON question.
   * Computes cosine similarities to each chunk (used by the similarity chart).
   * Calls the RAG chain and returns the answer plus short source snippets.
4. **/clear**:
   * Resets in-memory chat memory and similarity timeline (keeps persisted Chroma intact).

**Run instructions (step-by-step)**

1. Create a Python venv and activate it:

bash

CopyEdit

python -m venv venv

# Linux / macOS

source venv/bin/activate

# Windows (PowerShell)

venv\Scripts\Activate.ps1

1. Install dependencies:

bash

CopyEdit

pip install -r requirements.txt

1. Set your OpenAI API key in the environment:

* macOS / Linux:

bash

CopyEdit

export OPENAI\_API\_KEY="sk-..."

* Windows cmd:

cmd

CopyEdit

set OPENAI\_API\_KEY=sk-...

* Windows PowerShell:

powershell

CopyEdit

$env:OPENAI\_API\_KEY = "sk-..."

1. Start the server:

bash

CopyEdit

uvicorn app:app --reload

1. Open your browser to:

cpp

CopyEdit

http://127.0.0.1:8000/

1. Ask questions in the text area. After each question:

* The answer appears in chat.
* Top source snippets shown below the chat.
* Similarity timeline image refreshes to reflect which chunks the retriever found most relevant at each turn.

**Notes, tips & possible extensions**

* **Persisted Chroma**: embeddings are stored in chroma\_persist. If you change your documents, either:
  + delete the chroma\_persist folder and restart, or
  + add code to upsert changed docs (more advanced).
* **Memory persistence**: this example keeps memory in RAM only. If you want memory across server restarts, I can add Redis-backed memory or file-based persistence for chat history.
* **Safety & cost**: test with small prompts and carefully manage which LLM you use. gpt-4o-mini is used as an example — replace with a model you have access to.
* **Scaling**: for production, you’d add authentication, rate-limiting, async handling, and persistent memory.

If you want, I can now:

* add **Redis-based persistent memory** so chats persist between restarts, or
* replace the charting with **Plotly** to get fully interactive charts in the browser, or
* show **how to deploy** this on a small cloud VM (Ubuntu) with systemd + uvicorn.

Which one shall we add next?

**Full Stack Project 3**

**Fast API full stack project –**

below is a **complete full-stack project** you can drop into a folder and run. It’s the same RAG + ConversationalRetrievalChain app from earlier, but packaged as a clean project with:

* FastAPI backend (app.py)
* Jinja2 HTML frontend (templates/index.html)
* Persistent Chroma vector store (disk)
* Optional Redis-backed persistent conversation memory (via redis + langchain memory) with a docker-compose for convenience
* requirements.txt, .env.example, and my\_notes.txt sample doc
* Clear run & deploy instructions

I’ll show the directory layout, then full contents for each file, and finally step-by-step run instructions. Copy & paste — it’s runnable.

**Project layout (create these files / folders)**

arduino

CopyEdit

my\_langchain\_rag\_web/

├── app.py

├── requirements.txt

├── my\_notes.txt

├── .env.example

├── docker-compose.yml # optional (for running Redis)

├── templates/

│ └── index.html

└── chroma\_persist/ # will be created by Chroma at runtime

**1) requirements.txt**

Save this and install with pip install -r requirements.txt.

nginx

CopyEdit

fastapi

uvicorn[standard]

jinja2

langchain

openai

chromadb

matplotlib

numpy

python-multipart

redis

langchain[redis] # provides RedisMemory support; optional depending on your langchain install

python-dotenv

Note: langchain[redis] is optional but convenient if you want Redis-backed memory. If you hit install issues, remove the optional extras and install redis separately.

**2) .env.example**

Create .env from this (or export env vars in your shell).

ini

CopyEdit

OPENAI\_API\_KEY=sk-REPLACE\_WITH\_YOUR\_KEY

CHROMA\_PERSIST\_DIR=chroma\_persist

PORT=8000

# If you want persistent memory using Redis (optional)

REDIS\_URL=redis://localhost:6379/0

USE\_REDIS\_MEMORY=false

**3) my\_notes.txt (sample document)**

Put this in the project root. Replace with your own documents later.

vbnet

CopyEdit

In Q1 2024, our company launched two new products that exceeded sales expectations.

In Q2 2024, we expanded into three new markets: India, Brazil, and South Korea.

Customer satisfaction increased by 15% due to better support response times.

In Q3 2024, we plan to focus on AI-powered product recommendations.

In Q4 2024, our target is to increase revenue by 20% compared to Q4 2023.

**4) templates/index.html (frontend)**

Create templates/ folder and put this file inside.

html

CopyEdit

<!doctype html>

<html>

<head>

<meta charset="utf-8" />

<title>LangChain RAG Chat (Web)</title>

<style>

body { font-family: Arial, sans-serif; margin: 16px; }

.container { display: flex; gap: 20px; }

.left { flex: 1; min-width: 380px; }

.right { width: 480px; }

.chat { border: 1px solid #ddd; padding: 10px; height: 420px; overflow-y: auto; background:#fafafa; }

.message.user { text-align: right; color: #222; margin: 6px 0; }

.message.bot { text-align: left; color: #004; margin: 6px 0; }

.controls { margin-top: 10px; display:flex; gap:8px; }

textarea { width:100%; height: 70px; }

button { padding: 8px 12px; }

.sources { font-size: 0.9em; color:#333; margin-top:6px; background:#fff; padding:8px; border-radius:6px; border:1px solid #eee }

.chart { border:1px solid #eee; padding:8px; background:#fff; margin-bottom:10px; }

</style>

</head>

<body>

<h2>LangChain RAG Chat (Web)</h2>

<div class="container">

<div class="left">

<div id="chat" class="chat"></div>

<div class="controls">

<textarea id="input" placeholder="Ask something from your documents..."></textarea>

<div style="display:flex;flex-direction:column;">

<button id="send">Send</button>

<button id="clear">Clear</button>

</div>

</div>

<div id="sources" class="sources"></div>

</div>

<div class="right">

<div class="chart">

<h4>Chunk Lengths</h4>

<img id="chunks\_img" src="/chart/chunks.png" alt="chunks" style="width:100%;height:auto"/>

</div>

<div class="chart">

<h4>Similarity Timeline</h4>

<img id="sim\_img" src="/chart/similarity.png" alt="similarity" style="width:100%;height:auto"/>

</div>

</div>

</div>

<script>

async function postQuestion(q) {

const resp = await fetch('/ask', {

method: 'POST',

headers: {'Content-Type':'application/json'},

body: JSON.stringify({question: q})

});

return resp.json();

}

function addMessage(role, text){

const chat = document.getElementById('chat');

const div = document.createElement('div');

div.className = 'message ' + (role==='user' ? 'user' : 'bot');

div.innerText = (role==='user' ? 'You: ' : 'Bot: ') + text;

chat.appendChild(div);

chat.scrollTop = chat.scrollHeight;

}

document.getElementById('send').addEventListener('click', async ()=>{

const ta = document.getElementById('input');

const q = ta.value.trim();

if(!q) return;

addMessage('user', q);

ta.value = '';

document.getElementById('sources').innerText = 'Thinking...';

const data = await postQuestion(q);

addMessage('bot', data.answer || '(no answer)');

// show sources

const srcDiv = document.getElementById('sources');

if(data.source\_snippets && data.source\_snippets.length){

srcDiv.innerHTML = '<b>Top source snippets:</b><ol>' + data.source\_snippets.map(s => `<li>${s}</li>`).join('') + '</ol>';

} else {

srcDiv.innerText = '(no sources returned)';

}

// refresh similarity chart image (add timestamp to bust cache)

document.getElementById('sim\_img').src = '/chart/similarity.png?ts=' + Date.now();

});

document.getElementById('clear').addEventListener('click', async ()=>{

// clear chat in UI and on server

await fetch('/clear', {method:'POST'});

document.getElementById('chat').innerHTML = '';

document.getElementById('sources').innerHTML = '';

document.getElementById('sim\_img').src = '/chart/similarity.png?ts=' + Date.now();

});

</script>

</body>

</html>

**5) docker-compose.yml (optional: for Redis)**

If you want Redis for persistent memory, this docker-compose spins one up quickly.

yaml

CopyEdit

version: "3.8"

services:

redis:

image: redis:7-alpine

ports:

- "6379:6379"

volumes:

- redis\_data:/data

volumes:

redis\_data:

Run with:

bash

CopyEdit

docker-compose up -d

**6) app.py — full FastAPI backend (copy exactly)**

This file wires everything together. It supports optional Redis memory (toggle via env var).

python

CopyEdit

# app.py

"""

FastAPI + LangChain RAG Web UI (full project)

Run:

1) create .env from .env.example or export env vars

2) pip install -r requirements.txt

3) optional: docker-compose up -d # if using Redis for memory

4) uvicorn app:app --reload

5) open http://127.0.0.1:8000/

"""

import os

import io

import time

import numpy as np

import matplotlib.pyplot as plt

from fastapi import FastAPI, Request

from fastapi.responses import HTMLResponse, JSONResponse, StreamingResponse

from fastapi.templating import Jinja2Templates

from dotenv import load\_dotenv

# LangChain imports

from langchain.chat\_models import ChatOpenAI

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.document\_loaders import TextLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain.vectorstores import Chroma

from langchain.chains import ConversationalRetrievalChain

from langchain.memory import ConversationBufferMemory

# Optional Redis memory

USE\_REDIS = False

try:

from langchain.memory import RedisChatMessageHistory

from langchain.memory import ChatMessageHistory

from langchain.memory import RedisMemory

USE\_REDIS = True

except Exception:

# If langchain version doesn't have Redis memory, we will keep in-memory fallback

USE\_REDIS = False

# Load .env if present

load\_dotenv()

# -------------------------

# Config

# -------------------------

OPENAI\_API\_KEY = os.environ.get("OPENAI\_API\_KEY")

if not OPENAI\_API\_KEY:

raise RuntimeError("Set OPENAI\_API\_KEY in environment (or create .env with OPENAI\_API\_KEY).")

CHROMA\_DIR = os.environ.get("CHROMA\_PERSIST\_DIR", "chroma\_persist")

PORT = int(os.environ.get("PORT", 8000))

REDIS\_URL = os.environ.get("REDIS\_URL", "redis://localhost:6379/0")

USE\_REDIS\_MEMORY\_FLAG = os.environ.get("USE\_REDIS\_MEMORY", "false").lower() in ("1", "true", "yes")

# -------------------------

# FastAPI + templates

# -------------------------

app = FastAPI()

templates = Jinja2Templates(directory="templates")

# -------------------------

# Documents -> chunks -> vectorstore (persisted)

# -------------------------

DOC\_PATH = "my\_notes.txt"

if not os.path.exists(DOC\_PATH):

raise RuntimeError(f"Document file not found: {DOC\_PATH}")

# 1) Load docs

loader = TextLoader(DOC\_PATH)

docs = loader.load()

# 2) Split into chunks

splitter = RecursiveCharacterTextSplitter(chunk\_size=120, chunk\_overlap=20)

chunks = splitter.split\_documents(docs)

# 3) Create embeddings & Chroma vectorstore (persist)

embeddings = OpenAIEmbeddings()

vectorstore = Chroma.from\_documents(documents=chunks, embedding=embeddings, persist\_directory=CHROMA\_DIR)

vectorstore.persist() # writes to disk in CHROMA\_DIR

# 4) Retriever

retriever = vectorstore.as\_retriever(search\_kwargs={"k": 3})

# -------------------------

# LLM, Memory, Chain

# -------------------------

llm = ChatOpenAI(temperature=0.0, model="gpt-4o-mini")

# Memory: choose Redis if requested and available

if USE\_REDIS\_MEMORY\_FLAG and USE\_REDIS:

# Redis-backed memory

try:

# RedisChatMessageHistory and RedisMemory interfaces may vary by langchain version

chat\_history = RedisChatMessageHistory(url=REDIS\_URL)

memory = RedisMemory(chat\_memory=chat\_history)

print("Using Redis-backed memory:", REDIS\_URL)

except Exception as e:

print("Redis memory setup failed, falling back to ConversationBufferMemory:", e)

memory = ConversationBufferMemory(memory\_key="chat\_history", return\_messages=True)

else:

memory = ConversationBufferMemory(memory\_key="chat\_history", return\_messages=True)

chain = ConversationalRetrievalChain.from\_llm(

llm=llm,

retriever=retriever,

memory=memory,

return\_source\_documents=True

)

# -------------------------

# Precompute chunk vectors for similarity plotting

# -------------------------

chunk\_vecs = [np.array(embeddings.embed\_query(d.page\_content)) for d in chunks]

similarity\_timeline = [] # list of lists; append sims for each query

# -------------------------

# Helper: figure -> PNG bytes

# -------------------------

def fig\_to\_png\_bytes(fig):

buf = io.BytesIO()

fig.savefig(buf, format="png", bbox\_inches="tight")

buf.seek(0)

return buf

# -------------------------

# Routes / endpoints

# -------------------------

@app.get("/", response\_class=HTMLResponse)

async def home(request: Request):

return templates.TemplateResponse("index.html", {"request": request})

@app.get("/chart/chunks.png")

def chart\_chunks():

lengths = [len(d.page\_content) for d in chunks]

fig, ax = plt.subplots(figsize=(6, 2.8))

ax.bar(range(len(lengths)), lengths)

ax.set\_title("Chunk Lengths (characters)")

ax.set\_xlabel("Chunk index")

ax.set\_ylabel("Length (chars)")

fig.tight\_layout()

buf = fig\_to\_png\_bytes(fig)

plt.close(fig)

return StreamingResponse(buf, media\_type="image/png")

@app.get("/chart/similarity.png")

def chart\_similarity():

fig, ax = plt.subplots(figsize=(6, 4))

if not similarity\_timeline:

ax.text(0.5, 0.5, "No queries yet.\nAsk something to see similarity timeline.",

ha='center', va='center', fontsize=12)

ax.set\_axis\_off()

else:

sim\_array = np.array(similarity\_timeline)

num\_queries, num\_chunks = sim\_array.shape

for chunk\_idx in range(num\_chunks):

ax.plot(range(num\_queries), sim\_array[:, chunk\_idx], marker='o', label=f"chunk {chunk\_idx}")

ax.set\_title("Chunk Similarity Over Conversation")

ax.set\_xlabel("Query number (0 = first question)")

ax.set\_ylabel("Cosine similarity")

ax.set\_ylim(0, 1)

ax.legend(fontsize='small', ncol=2, loc='upper right')

fig.tight\_layout()

buf = fig\_to\_png\_bytes(fig)

plt.close(fig)

return StreamingResponse(buf, media\_type="image/png")

@app.post("/ask")

async def ask(request: Request):

body = await request.json()

question = body.get("question", "").strip()

if not question:

return JSONResponse({"error": "empty question"}, status\_code=400)

# compute similarity (for timeline chart)

q\_vec = np.array(embeddings.embed\_query(question))

def cosine(a,b): return float(np.dot(a,b)/(np.linalg.norm(a)\*np.linalg.norm(b)+1e-12))

sims = [cosine(q\_vec, cv) for cv in chunk\_vecs]

similarity\_timeline.append(sims)

# call chain (memory is attached)

res = chain({"question": question})

answer = res.get("answer") or res.get("output\_text") or str(res)

source\_docs = res.get("source\_documents", [])

snippets = []

for sd in source\_docs:

text = sd.page\_content.replace("\n", " ").strip()

snippets.append(text[:300] + ("..." if len(text) > 300 else ""))

return JSONResponse({"answer": answer, "source\_snippets": snippets})

@app.post("/clear")

async def clear\_memory():

global memory, chain, similarity\_timeline

# reset memory depending on configured backend

if USE\_REDIS\_MEMORY\_FLAG and USE\_REDIS:

try:

chat\_history = RedisChatMessageHistory(url=REDIS\_URL)

memory = RedisMemory(chat\_memory=chat\_history)

except Exception:

memory = ConversationBufferMemory(memory\_key="chat\_history", return\_messages=True)

else:

memory = ConversationBufferMemory(memory\_key="chat\_history", return\_messages=True)

chain = ConversationalRetrievalChain.from\_llm(llm=llm, retriever=retriever, memory=memory, return\_source\_documents=True)

similarity\_timeline = []

return JSONResponse({"ok": True})

**How to run (step-by-step)**

1. Clone/copy this project folder (my\_langchain\_rag\_web).
2. Create & activate a Python virtual environment:

bash

CopyEdit

python -m venv venv

# mac/linux

source venv/bin/activate

# windows (cmd)

venv\Scripts\activate

1. Install dependencies:

bash

CopyEdit

pip install -r requirements.txt

1. Create .env from .env.example and set your OpenAI key:

bash

CopyEdit

cp .env.example .env

# edit .env and set OPENAI\_API\_KEY=sk-...

Or set environment variables directly in your shell.

1. (Optional) Run Redis if you want persistent memory across restarts:

bash

CopyEdit

docker-compose up -d

# set USE\_REDIS\_MEMORY=true in .env

1. Start the app:

bash

CopyEdit

uvicorn app:app --reload --port 8000

1. Open the UI:

cpp

CopyEdit

http://127.0.0.1:8000/

Type questions and watch the similarity timeline update; use Clear to reset memory.

**Notes, tips & troubleshooting**

* **Chroma persistence:** embeddings are stored in the CHROMA\_DIR (default chroma\_persist). If you change my\_notes.txt you should delete the folder to force recompute, or implement upsert logic (advanced).
* **OpenAI costs:** embeddings and LLM calls cost tokens. For development, keep prompts small and test with fewer queries.
* **Model availability:** replace model "gpt-4o-mini" with a model you have access to (e.g., "gpt-3.5-turbo").
* **Redis Memory:** RedisMemory APIs can vary across LangChain versions. If Redis memory fails to import, we gracefully fall back to in-memory ConversationBufferMemory.
* **Matplotlib + headless servers:** the code uses matplotlib to generate PNGs. If you run on a server without a display, set the backend to Agg by adding near top of app.py:

python

CopyEdit

import matplotlib

matplotlib.use("Agg")

**Next steps I can help with (pick any)**

* Make the UI prettier (React/Bootstrap) and show interactive Plotly charts instead of PNGs.
* Add authentication + rate limiting and deploy to a small VM.
* Add persistable chat memory to disk (file/SQLite) if you prefer not to use Redis.
* Add streaming tokens to the UI so answers appear as the model generates.

Which of these would you like me to implement next?