

```

import os
from dotenv import load_dotenv
load_dotenv()

# Set API keys for OpenAI and Langsmith from environment variables
os.environ["OPENAI_API_KEY"] = os.getenv("OPENAI_API_KEY")
os.environ["LANGCHAIN_API_KEY"] = os.getenv("LANGCHAIN_API_KEY")
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_PROJECT"] = os.getenv("LANGCHAIN_PROJECT")

# ===== Initialize LLM (GPT-4o via LangChain) =====
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model="gpt-4o")

# ===== Basic Prompt Template Setup =====
from langchain_core.prompts import ChatPromptTemplate

# Define a reusable chat prompt template
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are an expert AI Engineer. Provide me answers based on the questions."),
    ("user", "{input}")
])

# ===== Output Parser Setup =====
from langchain_core.output_parsers import StrOutputParser
output_parser = StrOutputParser()

# ===== Create Final Chain: Prompt -> LLM -> Parser =====
chain = prompt | llm | output_parser

# ===== Invoke the Chain =====
response = chain.invoke({"input": "Can you tell me about Langsmith?"})
print(response)

```



```

pgsql

User Input (e.g., "Tell me about Langsmith") -
Prompt Template -
GPT-4o Model (via LangChain) -
String Parser -
Final Answer

```

## 1. Environment Variables & API Keys

- `dotenv.load_dotenv()` loads API keys from a `.env` file into your Python environment.
- `os.environ[...]` sets the necessary environment variables for OpenAI and LangSmith tracking.

## 2. LLM Initialization

```
python
```

Copy Edit

```
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model="gpt-4o")
```

- This loads the **GPT-4o** model via LangChain's OpenAI integration.

## 3. Basic LLM Call

```
python
```

Copy Edit

```
result = llm.invoke("What is generative AI?")
```

- Calls the model directly with a simple string and prints the output.

## 4. Prompt Template

```
python
```

Copy Edit

```
ChatPromptTemplate.from_messages(...)
```

- You define a **chat template** where the system message sets the behavior ("You are an expert AI Engineer") and the user prompt is dynamic using `{input}`.
- You define a **chat template** where the system message sets the behavior ("You are an expert AI Engineer") and the user prompt is dynamic using `{input}`.

## 5. Chaining Prompt to Model

```
python
```

Copy Edit

```
chain = prompt | llm
response = chain.invoke({"input": "Can you tell me about Langsmith?"})
```

- A LangChain **pipeline is formed**: prompt → model. The input is passed to the template and processed by the model.

## 6. Adding Output Parser

```
python
```

Copy Edit

```
from langchain_core.output_parsers import StrOutputParser
```

- `StrOutputParser()` ensures the output is returned as a plain string, not an object or message type.

## 7. Final Chain

```
python
```

Copy Edit

```
chain = prompt | llm | output_parser
```

- Now the full pipeline is:
  - Fill prompt template
  - Call GPT-4o
  - Extract response text as string

## RAG (Retrieval-Augmented Generation) pipeline using LangChain

```
import os
from dotenv import load_dotenv
load_dotenv()

# Set API keys for OpenAI and Langsmith from environment variables
os.environ["OPENAI_API_KEY"] = os.getenv("OPENAI_API_KEY")
os.environ["LANGCHAIN_API_KEY"] = os.getenv("LANGCHAIN_API_KEY")
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_PROJECT"] = os.getenv("LANGCHAIN_PROJECT")
```

python

```
# ===== Step 1: Load and Chunk Web Data =====
from langchain_community.document_loaders import WebBaseLoader
from langchain_text_splitter import RecursiveCharacterTextSplitter

# Load LangSmith documentation page
loader = WebBaseLoader("https://docs.smith.langchain.com/tutorials/Administrators/manage_
docs = loader.load()

# Split into smaller chunks for embeddings
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
documents = text_splitter.split_documents(docs)

# ===== Step 2: Generate Embeddings and Store in FAISS =====
from langchain_openai import OpenAIEmbeddings
from langchain_community.vectorstores import FAISS

embeddings = OpenAIEmbeddings()
vectorstoredb = FAISS.from_documents(documents, embeddings)

# ===== Step 3: Setup LLM and Retrieval Chain =====
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_chains.combine_documents import create_stuff_documents_chain

llm = ChatOpenAI(model="gpt-4o")
```

```

# Prompt template instructing LLM to answer using only provided context
prompt = ChatPromptTemplate.from_template("""
Answer the following question based only on the provided context:
<context>
{context}
</context>
""")

# Document chain: context + LLM
document_chain = create_stuff_documents_chain(llm, prompt)

# Connect retriever to LLM chain
from langchain.chains import create_retrieval_chain
retriever = vectorstoredb.as_retriever()
retrieval_chain = create_retrieval_chain(retriever, document_chain)

# ===== Step 4: Run the Retrieval-Augmented Query =====
query = "LangSmith has two usage limits: total traces and extended"
response = retrieval_chain.invoke({"input": query})

# Print Final Answer and Context Used
print("Answer:\n", response['answer'])
print("\nContext Used:\n", response['context'][0].page_content)

```

## Summary of the Pipeline

vbnet

Web Page → Document Loader → Text Splitter → Embeddings → FAISS DB →  
Retriever + Prompt + LLM → Answer based on retrieved chunks

When similarity search is already available , why do we need the retrievers over the database, explain top 5 reasons



Thought for a few seconds >

Even though you can call a vector store's `.similarity_search()` method directly, wrapping it in a "retriever" abstraction gives you a much richer, more reliable, and more maintainable retrieval-augmented pipeline. Here are the top 5 reasons why you almost always want to use a Retriever, not raw similarity search:

| Reason                            | What it gives you  | Citation                                  |
|-----------------------------------|--|---|
| 1. Standardized Interface         | A Retriever implements a simple <code>get_relevant_documents(query)</code> API. Your application code no longer needs to know whether you're using FAISS, Chroma, ElasticSearch, or some future store. You can swap backends without touching your business logic. | <a href="#">Introduction</a>    LangChain |
| 2. Pre- and Post-Processing Hooks | You can plug in query-expansion, spelling correction, or reranking logic in one place. For example, you might apply sparse keyword expansion before the vector lookup, or run an expensive cross-encoder only on the top-k results—all inside the Retriever.       | <a href="#">wikipedia</a>                 |
| 3. Batching & Efficiency          | A Retriever can batch multiple queries, cache embeddings, de-duplicate overlapping chunks, and manage asynchronous calls. If you call <code>.similarity_search()</code> yourself, you must reimplement all these optimizations.                                    | <a href="#">Introduction</a>    LangChain |
| 4. Score Thresholding & Filtering | Many Retriever implementations let you specify score thresholds, hybrid (dense + sparse) search, or max-min relevance filtering out of the box. With raw similarity search you'd have to manually filter on the returned distances.                                | <a href="#">Introduction</a>    LangChain |

## 5. Integration in Chains & Tooling

LangChain and other frameworks recognize Retrievers as first-class components—automatically logging to LangSmith, generating explainability traces, wiring into RetrievalQA chains, etc. Raw search calls bypass all of that observability and orchestration.

## In practice

```
python Copy Edit
# ❌ Raw similarity search – you must manage everything yourself
docs = vectorstore.similarity_search(query, k=5)
# You still need to format, re-rank, filter, and feed into your LLM.

# ✅ Retriever – one simple interface, pluggable enhancements,
#   and full integration into RAG chains & monitoring:
retriever = vectorstore.as_retriever(
    search_type="similarity_score_threshold",
    search_kwargs={"k": 5, "score_threshold": 0.8}
)
relevant_docs = retriever.get_relevant_documents(query)
```

By using a Retriever you gain maintainability, performance optimizations, richer retrieval strategies, and seamless integration into higher-level RAG chains.

```
# 1. Load environment variables for API keys
import os
from dotenv import load_dotenv

load_dotenv()
os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY")
os.environ["LANGCHAIN_API_KEY"] = os.getenv("LANGCHAIN_API_KEY")
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_PROJECT"] = os.getenv("LANGCHAIN_PROJECT")

# 2. Initialize the OpenAI LLM (GPT-4o via LangChain wrapper)
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o")

# 3. Create a chat prompt template to define system and user roles
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are an expert AI Engineer. Provide me answers based on the questions."),
    ("user", "{input}")
])

# 4. Set up a full LangChain pipeline: Prompt → LLM → Output Parser
from langchain_core.output_parsers import StrOutputParser

output_parser = StrOutputParser()

# Create a chain that formats the prompt, sends it to the LLM, and parses the output
chain = prompt | llm | output_parser

# 5. Invoke the chain with user input and print the final answer
response = chain.invoke({"input": "Can you tell me about LangSmith?"})
print(response)
```

## Explanation of What's Happening

| Step | Code   | Description   |
|------|--|---|
| 1    | <code>load_dotenv() + os.environ[...]</code>       | Loads environment variables (API keys) securely using <code>.env</code> file. Needed for OpenAI + Langsmith tracking. |
| 2    | <code>ChatOpenAI(...)</code>                       | Initializes the GPT-4o model via LangChain. This wraps OpenAI's chat API so you can use it in LangChain chains.       |
| 3    | <code>ChatPromptTemplate.from_messages(...)</code> | Builds a reusable prompt template with a system message and user input placeholder.                                   |
| 4    | <code>strOutputParser()</code>                     | Ensures the output is returned as a clean string (removes token metadata, formatting issues, etc).                    |
| 5    | <code>chain.invoke(...)</code>                     | Executes the full chain: formats prompt → sends to GPT-4o → parses response → returns final answer.                   |

## RAG - preparation

### Summary of What the Code Does:

- 1. Environment Setup:** Loads API keys securely from `.env`.
- 2. Data Loading:** Scrapes documentation from a website.
- 3. Preprocessing:** Splits documents into manageable chunks.
- 4. Embedding Generation:** Converts text to vectors using OpenAI embeddings.
- 5. Vector Store:** Stores embeddings in a FAISS database for similarity search.
- 6. Querying:** Retrieves relevant documents based on a user query.
- 7. LLM Chain:** Uses GPT-4o to generate answers using context.
- 8. Retrieval Chain:** Combines the retriever and LLM to form a complete RAG pipeline.

```
### --- 1. Load API Keys and Environment Variables --- ###
import os
from dotenv import load_dotenv
load_dotenv() # Loads variables from a .env file into environment

# Set environment variables for OpenAI and LangChain tracking
os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY")
os.environ["LANGCHAIN_API_KEY"] = os.getenv("LANGCHAIN_API_KEY")
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_PROJECT"] = os.getenv("LANGCHAIN_PROJECT")

### --- 2. Load Website Content as Documents --- ###
from langchain_community.document_loaders import WebBaseLoader

# Scrapes content from a given URL and loads it into LangChain document format
loader = WebBaseLoader("https://docs.smith.langchain.com/tutorials/Administrators/manage_")
docs = loader.load()

### --- 3. Split Long Documents into Chunks --- ###
from langchain_text_splitters import RecursiveCharacterTextSplitter

# Splits text into overlapping chunks to preserve context across cuts
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
documents = text_splitter.split_documents(docs)

### --- 4. Generate Embeddings for Each Chunk --- ###
from langchain_openai import OpenAIEMBEDDINGS

# Converts chunks of text into numerical vectors using OpenAI's embedding model
embeddings = OpenAIEMBEDDINGS()
```

```
### --- 5. Store Vectors in a FAISS Vector Store --- ###
from langchain_community.vectorstores import FAISS

# Creates an in-memory FAISS database for efficient similarity search
vectorstoredb = FAISS.from_documents(documents, embeddings)

### --- 6. Perform a Sample Similarity Search --- ###
query = "LangSmith has two usage limits: total traces and extended"
results = vectorstoredb.similarity_search(query)
print("Top Match:\n", results[0].page_content)

### --- 7. Define the Language Model --- ###
from langchain_openai import ChatOpenAI

# Use OpenAI's GPT-4o for answering questions
llm = ChatOpenAI(model="gpt-4o")

### --- 8. Create a Prompt Template and LLM Chain --- ###
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain_core.prompts import ChatPromptTemplate

# This prompt instructs the model to only use the provided context to answer
prompt = ChatPromptTemplate.from_template(
    """
    Answer the following question based only on the provided context:
    <context>
    {context}
    </context>
    """
)

)
```

 Copy  Edit

```
# Combines LLM with the prompt into a chain that takes documents as context
document_chain = create_stuff_documents_chain(llm, prompt)

### --- 9. Test LLM Chain with Hardcoded Context --- ###
from langchain_core.documents import Document

# Manually invoke the LLM with a provided document
test_response = document_chain.invoke({
    "input": query,
    "context": [
        Document(page_content="LangSmith has two usage limits: total traces and extended
    ]
})
print("LLM Output:\n", test_response)

### --- 10. Set Up a Retriever from the Vector Store --- ###
retriever = vectorstoredb.as_retriever()

### --- 11. Create Full Retrieval-Augmented Chain --- ###
from langchain.chains import create_retrieval_chain

# Combines retriever + LLM to automatically find and use relevant documents
retrieval_chain = create_retrieval_chain(retriever, document_chain)

### --- 12. Run Final RAG Pipeline --- ###
response = retrieval_chain.invoke({"input": query})

# Final answer and its source context
print("Answer:\n", response['answer'])
print("\nRetrieved Context:\n", response['context'][0].page_content)
```

 Copy  Edit

## Explanation of Key Libraries:

| Library  | Purpose   |
|--|---|
| <code>dotenv</code>  | Loads environment variables from <code>.env</code> files.             |
| <code>langchain_community.document_loaders.WebBaseLoader</code>              | Scrapes webpage content into LangChain documents.                     |
| <code>langchain_text_splitters.RecursiveCharacterTextSplitter</code>         | Splits long documents into overlapping chunks.                        |
| <code>langchain_openai.OpenAIEmbeddings</code>                               | Converts text into vector embeddings via OpenAI.                      |
| <code>FAISS</code> (from LangChain Community)                                | Stores and searches vector data efficiently.                          |
| <code>langchain.chains.combine_documents.create_stuff_documents_chain</code> | Combines context documents with a language model to answer questions. |
| <code>langchain.chains.create_retrieval_chain</code>                         | Creates a RAG (retrieval-augmented generation) pipeline.              |
| <code>ChatOpenAI</code>  | Interface to OpenAI's GPT-based models like GPT-4o.                   |

# DOCUMENT INGESTION

## What This Code Does:

This script demonstrates how to **ingest documents from various sources** using the LangChain framework:

1. Plain text file
2. PDF file
3. Web page
4. ArXiv (scientific papers)
5. Wikipedia

Each loader returns a list of `Document` objects that can be further processed (split, embedded, queried, etc.).

python

 Copy

 Edit

```
### --- 1. Load a Plain Text File --- ###
from langchain_community.document_loaders import TextLoader

# Load a simple `*.txt` file into LangChain document format
text_loader = TextLoader('speech.txt')
text_documents = text_loader.load()

print(f"Loaded {len(text_documents)} document(s) from text file.")
print(text_documents[0].page_content[:300]) # Print first 300 characters
```

python

 Copy

 Edit

```
### --- 2. Load a PDF File --- ###
from langchain_community.document_loaders import PyPDFLoader

# Load a `*.pdf` file and extract content page by page
pdf_loader = PyPDFLoader('attention.pdf')
pdf_documents = pdf_loader.load()

print(f"Loaded {len(pdf_documents)} document(s) from PDF file.")
print(pdf_documents[0].page_content[:300])
```

```
### --- 3. Load Web Page Content --- ###
from langchain_community.document_loaders import WebBaseLoader
import bs4 # BeautifulSoup is used for parsing HTML content

# Load content from a webpage, restricting to specific HTML classes using
web_loader = WebBaseLoader(
    web_paths=("https://lilianweng.github.io/posts/2023-06-23-agent/"),
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(class_=("post-title", "post-content",
    )))
)

web_documents = web_loader.load()

print(f"Loaded {len(web_documents)} document(s) from the web.")
print(web_documents[0].page_content[:300])
```

```
python
### --- 4. Load Academic Papers from ArXiv --- ###
from langchain_community.document_loaders import ArxivLoader

# Load research papers by ArXiv ID (1706.03762 corresponds to the "Attention
arxiv_loader = ArxivLoader(query="1706.03762", load_max_docs=2)
arxiv_documents = arxiv_loader.load()

print(f"Loaded {len(arxiv_documents)} document(s) from ArXiv.")
print(arxiv_documents[0].page_content[:300])
```

```
python
```

 Copy

 Edit

```
### --- 5. Load Content from Wikipedia --- ###
from langchain_community.document_loaders import WikipediaLoader

# Search and load Wikipedia articles related to "Generative AI"
wiki_loader = WikipediaLoader(query="Generative AI", load_max_docs=2)
wiki_documents = wiki_loader.load()

print(f"Loaded {len(wiki_documents)} document(s) from Wikipedia.")
print(wiki_documents[0].page_content[:300])
```

| Library         | Loader  | Purpose |
|-----------------|---|---------|
| TextLoader      | Loads plain <code>.txt</code> files as documents.                           |         |
| PyPDFLoader     | Extracts text from each page of a PDF file.                                 |         |
| WebBaseLoader   | Scrapes specific content from HTML pages using <code>BeautifulSoup</code> . |         |
| ArxivLoader     | Downloads papers by ID or search from <code>arxiv.org</code> .              |         |
| WikipediaLoader | Fetches summaries of articles from Wikipedia.                               |         |

Each loader returns a list of `Document` objects in LangChain, where each document has:

- `.page_content` : The actual content (string).
- `.metadata` : Source info like file name, URL, page number, etc.

## Reccursive Character Splitter

## ✓ What This Code Does

1. Loads a PDF file and a `.txt` file as documents.
2. Splits these documents into smaller chunks for efficient processing.
3. Shows how to split raw strings directly (not just `Document` objects).
4. Uses LangChain's `RecursiveCharacterTextSplitter` to maintain semantic meaning across chunks.

```
### --- 1. Load and Split PDF Content --- ###
from langchain_community.document_loaders import PyPDFLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter

# Load the content of a PDF file; each page becomes one Document
pdf_loader = PyPDFLoader('attention.pdf')
pdf_docs = pdf_loader.load()

print(f"Loaded {len(pdf_docs)} pages from PDF.")

# Split the PDF content into chunks for embedding or querying
pdf_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
pdf_chunks = pdf_splitter.split_documents(pdf_docs)

print(f"PDF was split into {len(pdf_chunks)} chunks.")
print("First PDF chunk preview:\n", pdf_chunks[0].page_content[:300])
```

```
python

### --- 2. Load and Split Text File Content --- ###
from langchain_community.document_loaders import TextLoader

# Load the entire text file as one Document
text_loader = TextLoader('speech.txt')
text_docs = text_loader.load()

print(f"Loaded {len(text_docs)} document from text file.")
print("Text document preview:\n", text_docs[0].page_content[:300])
```

```

### --- 3. Alternative: Split Raw Text (String Input) --- ###

# Read the entire file content into a raw string
with open("speech.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

# Split raw string into chunks using create_documents
text_splitter = RecursiveCharacterTextSplitter(chunk_size=100, chunk_overlap=20)
split_text_chunks = text_splitter.create_documents([raw_text])

print(f"Raw text split into {len(split_text_chunks)} chunks.")
print("First chunk:\n", split_text_chunks[0].page_content)
print("\nSecond chunk:\n", split_text_chunks[1].page_content)

```

| Component                      | Purpose   | 🔗 |
|--------------------------------|---|---|
| PyPDFLoader                    | Loads each <b>page</b> of a PDF file into a <b>Document</b> object.   |   |
| TextLoader                     | Loads the contents of a <b>.txt</b> file into a single <b>Document</b> .  |   |
| RecursiveCharacterTextSplitter | Splits long documents or raw text into chunks based on character limits while trying to preserve sentence and paragraph boundaries. |   |
| create_documents([text])       | Accepts raw strings and returns <b>Document</b> objects from them.  |   |

## ✓ Notes on Usage:

- **Why split documents?**

Large texts can exceed token limits for embedding models or LLMs, so chunking helps manage this.

- **Why overlapping chunks?**

It helps retain context from the end of one chunk into the beginning of the next.



Like the above, there are many splitters available, go through langchain

# EMBEDDINGS- OPENAI

## ✓ What This Code Does

1. Loads OpenAI API keys from `.env`.
2. Initializes **OpenAI embedding models**, both default and dimension-limited versions.
3. Loads and splits a text file (`speech.txt`) into chunks.
4. Converts the chunks into **vector embeddings**.
5. Stores those vectors in a **Chroma vector store**.
6. Queries the vector store for similar chunks based on a **user query**.

```
### --- 1. Load API Keys from .env --- ###
import os
from dotenv import load_dotenv

# Load environment variables from .env file
load_dotenv()

# Set OpenAI API key in the environment explicitly for libraries that need it
os.environ["OPENAI_API_KEY"] = os.getenv("OPENAI_API_KEY")
```

---

```
python

### --- 2. Initialize OpenAI Embedding Model --- ###
from langchain_openai import OpenAIEMBEDDINGS

# Default full-dimension embedding model
default_embeddings = OpenAIEMBEDDINGS(model="text-embedding-3-large")

# Optional: specify output dimension (e.g., 1024 instead of 3072 for cost or performance)
reduced_embeddings = OpenAIEMBEDDINGS(model="text-embedding-3-large", dimensions=1024)
```

```
### --- 3. Generate Embeddings for a Sample Query --- ###
sample_text = "This is a tutorial on OPENAI embedding"

# Get the embedding vector (3072-dim by default, 1024-dim for the reduced model)
query_vector = reduced_embeddings.embed_query(sample_text)

print(f"Vector dimension: {len(query_vector)}")
print("Sample embedding vector (first 5 values):", query_vector[:5])
```

```
python

### --- 4. Load and Split Text File --- ###
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter

# Load 'speech.txt' as a Document object
loader = TextLoader('speech.txt')
docs = loader.load()

# Split document into 500-character chunks with 50-character overlap
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
chunked_docs = text_splitter.split_documents(docs)

print(f"Total chunks created: {len(chunked_docs)})")
```

```
### --- 5. Embed Documents and Store in Vector DB (Chroma) --- ###
from langchain_community.vectorstores import Chroma

# Create vector store from documents using the 1024-dimension embeddings
vector_db = Chroma.from_documents(chunked_docs, reduced_embeddings)

print("Chroma vector database created successfully.")
```

```
python

### --- 6. Search for Similar Chunks Using a Query --- ###
query = "It will be all the easier for us to conduct ourselves as belligerents"

# Perform similarity search using the query text
results = vector_db.similarity_search(query)

print(f"Top {len(results)} retrieved chunks:")
for i, doc in enumerate(results):
    print(f"\n--- Result #{i+1} ---")
    print(doc.page_content.strip())
```

| Library / Function                          | Purpose  |
|---|--|
| <code>dotenv</code>                         | Loads API keys from <code>.env</code> securely.                                    |
| <code>OpenAIEmbeddings</code>               | Converts text into high-dimensional vectors (embeddings).                          |
| <code>TextLoader</code>                     | Reads a text file and converts it into a LangChain <code>Document</code> object.   |
| <code>RecursiveCharacterTextSplitter</code> | Breaks long text into smaller, overlapping chunks to preserve context.             |
| <code>Chroma</code>                         | Lightweight, fast vector database for storing and retrieving vectorized documents. |
| <code>similarity_search(query)</code>       | Returns documents from the DB most similar to the input query.                     |

## ✓ Why Use Overlapping Chunks?

Overlapping text during chunking ensures that important context at the edge of one chunk carries into the next, reducing loss of semantic meaning.

### OLLAMA EMBEDDINGS

1. Initializes `OllamaEmbeddings` with a chosen local embedding model (e.g., `gemma:2b` , `mxbai-embed-large` ).
2. Generates vector embeddings for both:
  - A **list of documents** (using `embed_documents` )
  - A **search/query string** (using `embed_query` )

```
### --- 1. Import and Initialize Ollama Embeddings --- ###
from langchain_community.embeddings import OllamaEmbeddings

# Initialize Ollama embeddings using the "gemma:2b" model.
# This model runs locally via the Ollama server.
ollama_embeddings = OllamaEmbeddings(model="gemma:2b") # Defaults to "llama2" if model not specified

# Display model object
print("Initialized embedding model:", ollama_embeddings)
```

Copy Edit

```
python
### --- 2. Embed Multiple Documents --- ###

# Input: List of sentences (documents)
documents = [
    "Alpha is the first letter of Greek alphabet",
    "Beta is the second letter of Greek alphabet"
]

# Output: List of embedding vectors (one per document)
document_embeddings = ollama_embeddings.embed_documents(documents)

print(f"\nGenerated {len(document_embeddings)} document embeddings.")
print("First 5 values of first embedding vector:\n", document_embeddings[0][:5])
```

Copy Edit

```
### --- 3. Embed a Query (for Search/Similarity) --- ###

# Query string
query_text = "What is the second letter of Greek alphabet?"

# Output: Single embedding vector
query_embedding = ollama_embeddings.embed_query(query_text)

print("\nQuery embedding (first 5 values):", query_embedding[:5])
```

```
python

### --- 4. Try a Different Embedding Model --- ###

# List of supported models: https://ollama.com/blog/embedding-models

# Use a higher-capacity embedding model
mxbai_embeddings = OllamaEmbeddings(model="mxbai-embed-large")

sample_text = "This is a test document."

# Generate embedding for the sample query text
sample_embedding = mxbai_embeddings.embed_query(sample_text)

print("\nMXBAI model embedding (first 5 values):", sample_embedding[:5])
```

- Ollama must be running locally (`ollama serve`) and the models must be available (`ollama pull gemma:2b`).
- The `embed_query` result and each `embed_documents` result is a list of floats (vectors).
- You can use these vectors to:
  - Populate a vector database (e.g., FAISS, Chroma)
  - Perform similarity search
  - Power a RAG pipeline

## HUGGING FACE EMBEDDINGS

## ✓ What This Script Does

1. Loads your Hugging Face token from a `.env` file.
2. Initializes a Hugging Face embedding model.
3. Embeds a single text query.
4. Embeds multiple documents.
5. Outputs the resulting embedding vectors.

```
### --- 1. Load Environment Variables from `*.env` File --- ###
import os
from dotenv import load_dotenv

# Load variables like HF_TOKEN from .env into the environment
load_dotenv()

# Set Hugging Face token to allow access to models (if required)
os.environ['HF_TOKEN'] = os.getenv("HF_TOKEN")

### --- 2. Initialize Hugging Face Embedding Model --- ###
from langchain_huggingface import HuggingFaceEmbeddings

# Initialize embeddings using the popular 'all-MiniLM-L6-v2' sentence transformer
# This model returns a 384-dimensional vector for each text
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
```

python

 Copy  Edit

```

### --- 3. Embed a Single Query String --- ###
text = "This is a test document"

# Generate a vector representation of the query
query_vector = embeddings.embed_query(text)

print("Query vector (first 5 values):", query_vector[:5])
print("Vector length:", len(query_vector))

```

python

 Copy

```

### --- 4. Embed Multiple Documents --- ###
docs = [
    "This is a test document.",
    "This is not a test document."
]

# Convert each document into a vector
doc_vectors = embeddings.embed_documents(docs)

# Display first 5 values of first document embedding
print("\nDocument 1 vector (first 5 values):", doc_vectors[0][:5])
print("Total documents embedded:", len(doc_vectors))

```

| Library / Function                          | Purpose  |
|---|--|
| <code>dotenv</code>                         | Loads environment variables (like Hugging Face API keys) from <code>.env</code> .          |
| <code>os.environ['HF_TOKEN']</code>         | Explicitly sets the token in the environment for use by <code>transformers</code> .        |
| <code>HuggingFaceEmbeddings</code>          | Wrapper around Hugging Face's sentence-transformer models for easy embedding in LangChain. |
| <code>embed_query(text)</code>              | Converts a <b>single string</b> into a <b>dense embedding vector</b> .                     |
| <code>embed_documents(list_of_texts)</code> | Converts a <b>list of documents</b> into a list of vectors, one per document.              |

## Why Use `all-MiniLM-L6-v2`?

- Small, **fast**, and **efficient**.
- Outputs **384-dimensional** sentence embeddings.
- Great for semantic search, classification, and retrieval-augmented generation (RAG).

## VECTOR STORE - FAISS

### ✓ Objective:

1. Load and preprocess a text file.
2. Embed the text using **Ollama** embeddings.
3. Store embeddings in a **FAISS** vector store.
4. Perform **semantic search** queries on the vector database.
5. Save and reload the FAISS index.

```
### --- Step 1: Import Required Libraries --- ###
from langchain_community.document_loaders import TextLoader
from langchain_community.vectorstores import FAISS
from langchain_community.embeddings import OllamaEmbeddings
from langchain_text_splitters import CharacterTextSplitter
# To load plain text documents
# V Copy Edit
# Local embeddings via Ollama
# To split large texts
```

```
python
### --- Step 2: Load and Split the Document --- ###
loader = TextLoader("speech.txt")           # Load the speech.txt file
documents = loader.load()                   # Load into LangChain Document format

# Split text into chunks of max 1000 characters with 30 characters of overlap
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=30)
docs = text_splitter.split_documents(documents)
```

```
python
### --- Step 3: Generate Embeddings with Ollama --- ###
# Default model is usually "llama2" unless overridden
embeddings = OllamaEmbeddings()
```

```
python
### --- Step 4: Store Documents and Embeddings in FAISS Vector Store --- ###
# Index documents with their embedding vectors
db = FAISS.from_documents(docs, embeddings)
```

```
### --- Step 5: Perform Semantic Search (Querying) --- ###
query = "How does the speaker describe the desired outcome of the war?"
```

[Copy](#) [Edit](#)

```
# Basic semantic search (returns list of relevant documents)
results = db.similarity_search(query)
print("\nMost relevant chunk:\n", results[0].page_content)
```

python

[Copy](#) [Edit](#)

```
# Use FAISS as a retriever
retriever = db.as_retriever()
results = retriever.invoke(query)
print("\nRetriever output:\n", results[0].page_content)
```

python

[Copy](#) [Edit](#)

```
# Get both the document and its similarity score
docs_and_scores = db.similarity_search_with_score(query)
print("\nSimilarity Score:\n", docs_and_scores[0])
```

python

[Copy](#) [Edit](#)

```
### --- Step 6: Manually Embed a Query --- ###
# You can embed a query string into a dense vector manually
embedding_vector = embeddings.embed_query(query)
print("\nQuery embedding vector length:", len(embedding_vector))
```

```
### --- Step 7: Save and Reload FAISS Index --- ###
# Save the vector index locally
db.save_local("faiss_index")

# Load the saved FAISS index with the same embedding model
new_db = FAISS.load_local("faiss_index", embeddings, allow_dangerous_deserialization=True)

# Re-query the reloaded database
results = new_db.similarity_search(query)
print("\nQuery after loading index:\n", results[0].page_content)
```

| Component                           | Role  |
|-------------------------------------|---|
| TextLoader                          | Loads .txt files as LangChain Document objects.                                   |
| CharacterTextSplitter               | Splits large text into overlapping, manageable chunks for better semantic search. |
| OllamaEmbeddings                    | Generates vector embeddings locally (using models like llama2, gemma, etc.).      |
| FAISS                               | Efficient similarity search over embeddings using vector indexing.                |
| similarity_search(query)            | Returns relevant documents based on vector similarity.                            |
| similarity_search_with_score()      | Same as above, but includes a relevance score.                                    |
| embed_query(query)                  | Converts a single query string to a vector (for manual use).                      |
| save_local(path) / load_local(path) | Save/load FAISS indexes to/from disk.   |

## ✓ Summary:

- You built a **semantic retriever system** using LangChain + FAISS + Ollama embeddings.
- It's **local and efficient**, and you can **persist the vector index** for future querying.
- It can power tasks like **question-answering, retrieval-augmented generation (RAG), or chatbots**.

CHROMA –DB and other dbs are available

## VECTOR-RETRIEVER

It demonstrates a simple **RAG pipeline**:

- You create a **vector store** from a list of **Document s**.
- You use **semantic search** (via embeddings) to retrieve relevant documents.
- You plug that into an **LLM** (Groq's LLaMA3) to answer a question based on the retrieved context.

| Library  | Purpose  |
|--|--|
| <code>langchain_core.documents.Document</code>           | Represents text chunks with optional metadata      |
| <code>langchain_groq.ChatGroq</code>                     | Access Groq-hosted LLaMA3 model via LangChain      |
| <code>langchain_huggingface.HuggingFaceEmbeddings</code> | Converts text to vector embeddings using HF models |
| <code>langchain_chroma.Chroma</code>                     | Vector database to store and retrieve documents    |
| <code>langchain_core.prompts.ChatPromptTemplate</code>   | Format the prompt passed to the LLM                |
| <code>RunnablePassthrough</code>                         | Simple passthrough node in the pipeline            |
| <code>Retriever</code>                                   | Retrieves similar documents from vectorstore       |

## ✓ WHAT YOU BUILT

You built a **basic RAG system** that:

1. Stores documents with embeddings.
2. Retrieves relevant context for a question.
3. Sends both question and context to the LLM to generate a grounded, accurate response.

```
# Step 1: Create documents
from langchain_core.documents import Document

documents = [
    Document(page_content="Dogs are great companions, known for their loyalty and friendliness.", metadata={"source": "mammal-pets-doc"}),
    Document(page_content="Cats are independent pets that often enjoy their own space.", metadata={"source": "mammal-pets-doc"}),
    Document(page_content="Goldfish are popular pets for beginners, requiring relatively little care.", metadata={"source": "fish-pets-doc"}),
    Document(page_content="Parrots are intelligent birds capable of mimicking human speech.", metadata={"source": "bird-pets-doc"}),
    Document(page_content="Rabbits are social animals that need plenty of space to hop around.", metadata={"source": "mammal-pets-doc"}),
]
```

```
python
```

 Copy  Edit

```
# Step 2: Load environment variables (Groq API key and HF token)
import os
from dotenv import load_dotenv

load_dotenv()
os.environ["HF_TOKEN"] = os.getenv("HF_TOKEN")
groq_api_key = os.getenv("GROQ_API_KEY")
```

```
python
```

 Copy  Edit

```
# Step 3: Initialize LLM (Groq using LLaMA3 model)
from langchain_groq import ChatGroq

llm = ChatGroq(groq_api_key=groq_api_key, model="Llama3-8b-8192")
```

```
# Step 4: Generate vector embeddings for documents
from langchain_huggingface import HuggingFaceEmbeddings

embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
```

```
python
```

 Copy  Edit

```
# Step 5: Create a Chroma vector store from documents
from langchain_chroma import Chroma

vectorstore = Chroma.from_documents(documents=documents, embedding=embeddings)
```

```
# Step 6: Test similarity search
vectorstore.similarity_search("cat") # returns documents similar to the query
vectorstore.similarity_search_with_score("cat") # with similarity scores
```

python

 Copy

```
# Step 7: Create a retriever with top-k results
retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 1}
)
```

python

```
retriever.batch(["cat", "dog"])
```

python

```
# Step 8: RAG (Retrieval-Augmented Generation) Pipeline
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough

# Prompt template asking the model to only use retrieved context
prompt_template = ChatPromptTemplate.from_messages([
    ("human", """
        Answer this question using the provided context only.

        {question}

        Context:
        {context}
        """
    )
])
```

```
# Step 9: Create the RAG chain by combining retriever, prompt, and LLM
rag_chain = {
    "context": retriever,
    "question": RunnablePassthrough()
} | prompt_template | llm
```

python

Copy

```
# Step 10: Run the RAG pipeline
response = rag_chain.invoke("Tell me about dogs")
print(response.content) # Final response from the model
```

## Building A Chatbot

### ✓ Step-by-Step Guide to Build the Chatbot

#### Step 1: Install Required Packages

Make sure you have these installed:

bash

Copy

Edit

```
pip install streamlit python-dotenv langchain langchain-community langchain-core langchai
```

#### Step 2: Set Up Environment Variables

Create a `.env` file in your project folder:

env

Copy

Edit

```
GROQ_API_KEY=your_groq_api_key
LANGCHAIN_API_KEY=your_langchain_api_key
```

These are needed to authenticate requests to Groq (LLM) and LangChain's tracing tools.

## Step 3: Create Your Main Python File ( app.py )

This will:

- Load environment variables
- Initialize the chatbot (Groq LLM)
- Set up message trimming (to avoid overloading tokens)
- Handle chat history
- Provide a frontend using Streamlit

## Step 4: Code Breakdown

### LLM Initialization (Groq)

```
python
from langchain_groq import ChatGroq
model = ChatGroq(model_name="Compound-Beta-Mini", temperature=0.7, max_tokens=150)
```

Initializes a chatbot with a selected Groq model and parameters.

### Prompt Template

```
python
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant."),
    MessagesPlaceholder(variable_name="messages")
])
```

Defines how the LLM should behave. A system message and user/AI message structure are used.

### Token Trimmer

```
python
from langchain_core.messages import trim_messages
from langchain_community.utils.token_length import get_token_length_function
token_counter_fn = get_token_length_function(model_name="Compound-Beta-Mini")
trimmer = trim_messages(max_tokens=300, strategy="last", token_counter=token_counter_fn)
```

Ensures older messages are trimmed based on token limits.

### LangChain Runnable Pipeline

```
python
chain = (
    RunnablePassthrough.assign(messages=itemgetter("messages") | trimmer)
    | prompt
    | model
)
```

Chains together trimming → prompt formatting → LLM response.

### Message History Handling

```
python
from langchain_community.chat_message_histories import ChatMessageHistory
store = {}
def get_session_history(session_id):
    if session_id not in store:
        store[session_id] = ChatMessageHistory()
    return store[session_id]
```

Each session maintains its own chat history.

### Chat Chain With History

```
python
from langchain_core.runnables.history import RunnableWithMessageHistory
chat_chain = RunnableWithMessageHistory(
    chain,
    get_session_history,
    input_messages_key="messages",
    history_messages_key="messages"
)
```

This allows persistent conversation over time.

## Step 5: Build the Streamlit UI

Streamlit provides the chat frontend.

```
python
st.title("🤖 Smart Chatbot with Groq & LangSmith")

# Sidebar sliders
st.sidebar.slider("Temperature")
st.sidebar.slider("Max Tokens")

# User input
user_input = st.text_input("You:")

# Display chat history
st.markdown("**You:** Hello")
st.markdown("**Bot:** Hi! How can I help?")
```

## Step 6: Add Optional Features

- 🗑️ Clear Chat Button
- 🖉 Save/Restore chat history to disk
- 📁 File upload support (e.g., PDFs/text)
- 🎧 Voice input (using Whisper or similar)

| Component                  | Description                                |
|----------------------------|--|
| ChatGroq                   | Uses Groq LLM for responses                |
| ChatPromptTemplate         | Defines system + user/AI message structure |
| RunnableWithMessageHistory | Maintains session-wise chat history        |
| trim_messages              | Prevents token overflow in long chats      |
| streamlit                  | Provides the interactive UI                |

Would you like me to provide a folder structure or example `.env` file as well?

```
# filename: app.py
import os
import streamlit as st
from typing import Dict
from operator import itemgetter
from dotenv import load_dotenv
from langchain_core.messages import HumanMessage, AIMessage, SystemMessage, trim_messages
from langchain_core.runnables import RunnablePassthrough
from langchain_core.runnables.history import RunnableWithMessageHistory
from langchain_core.chat_history import BaseChatMessageHistory
from langchain_community.chat_message_histories import ChatMessageHistory
from langchain_prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_groq import ChatGroq
from langchain_community.utils.token_length import get_token_length_function
# ✅ Load environment variables
load_dotenv()
os.environ["LANGCHAIN_API_KEY"] = os.getenv("LANGCHAIN_API_KEY")
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_PROJECT"] = "Smart Chatbot With Groq"
os.environ["GROQ_API_KEY"] = os.getenv("GROQ_API_KEY")
# ✅ Streamlit Sidebar Controls
st.sidebar.title("Chat Settings")
temperature = st.sidebar.slider("Temperature", min_value=0.0, max_value=1.0, value=0.7)
max_tokens = st.sidebar.slider("Max Tokens", min_value=50, max_value=300, value=150)
# ✅ Instantiate Groq LLM
model = ChatGroq(model_name="Compound-Beta-Mini", temperature=temperature,
max_tokens=max_tokens)
```

```

# ✅ Get token counter for trimming
token_counter_fn = get_token_length_function(model_name="Compound-Beta-Mini")

# ✅ Trim old messages if they exceed token limit
trimmer = trim_messages(
    max_tokens=300, strategy="last", token_counter=token_counter_fn, include_system=True,
    allow_partial=False, start_on="human" )

# ✅ Prompt Template
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant. Answer all the questions to the best of your ability."),
    MessagesPlaceholder(variable_name="messages")
])

# ✅ In-memory session history store
store: Dict[str, BaseChatMessageHistory] = {}

def get_session_history(session_id: str) -> BaseChatMessageHistory:
    if session_id not in store:
        store[session_id] = ChatMessageHistory()
    return store[session_id]

# ✅ Build the LangChain Runnable
chain = (
    RunnablePassthrough.assign(messages=itemgetter("messages") | trimmer)
    | prompt | model )

chat_chain = RunnableWithMessageHistory(
    chain, get_session_history, input_messages_key="messages",
    history_messages_key="messages" )

# ✅ Streamlit UI
st.title("Smart Chatbot with Groq & LangSmith")

# Session ID Setup
if "session_id" not in st.session_state:
    st.session_state.session_id = "user_456"
session_id = st.session_state.session_id

# Chat history initialization
if "chat_history" not in st.session_state:
    st.session_state.chat_history = []

# ✅ Clear chat button
if st.button("Clear Chat"):
    st.session_state.chat_history = []

```

```

# User input
user_input = st.text_input("You:", key="user_input")

# ✅ Process user input
if user_input:
    full_messages = st.session_state.chat_history + [HumanMessage(content=user_input)]
    response = chat_chain.invoke(input={"messages": full_messages}, config={"configurable": {"session_id": session_id}} )
    st.session_state.chat_history.append(HumanMessage(content=user_input))
    st.session_state.chat_history.append(AIMessage(content=response.content))

# ✅ Display chat history
for msg in st.session_state.chat_history:
    if isinstance(msg, HumanMessage):
        st.markdown(f"**You:** {msg.content}")
    elif isinstance(msg, AIMessage):
        st.markdown(f"**Bot:** {msg.content}")

```

## PDFQuery\_LangChain

### Overview of What the Code Does

This script builds a chatbot that:

1. Reads a PDF document.
2. Splits it into chunks.
3. Stores the chunks in a vector database (Astra DB via Cassandra).
4. Embeds queries using OpenAI.
5. Answers questions by retrieving similar chunks and feeding them to a language model.

### Summary of Steps

1. **Read** and extract text from a PDF.
2. **Split** the text into manageable chunks.
3. **Embed** the chunks using OpenAI embeddings.
4. **Store** them in Astra DB via the Cassandra vector store.
5. **Query** with user input and perform similarity search.
6. **Retrieve** top-k documents and answer using the OpenAI LLM.

## ✓ Explanation of Libraries and Usage

| Library  | Functionality   | Comment on Usefulness                                    |
|--|---|--|
| <code>PyPDF2</code>  | Extracts text from PDFs                                   | Essential for parsing source document                    |
| <code>cassio</code>  | Initializes connection to Astra DB (vector DB)            | <input checked="" type="checkbox"/> Essential            |
| <code>langchain.vectorstores.cassandra</code>                      | Stores document embeddings in Cassandra                   | <input checked="" type="checkbox"/> Essential            |
| <code>langchain.embeddings.OpenAIEmbeddings</code>                 | Converts text to embeddings using OpenAI                  | <input checked="" type="checkbox"/> Essential            |
| <code>langchain.llms.OpenAI</code>                                 | OpenAI LLM for answering user questions                   | <input checked="" type="checkbox"/> Essential            |
| <code>langchain.indexes.vectorstore.VectorStoreIndexWrapper</code> | Wraps the vector store for convenient RAG query interface | <input checked="" type="checkbox"/> Optional but helpful |
| <code>langchain.text_splitter.CharacterTextSplitter</code>         | Splits long documents into chunks                         | <input checked="" type="checkbox"/> Essential            |

## ⚠ Redundant or Unused Parts

- `from datasets import load_dataset` → **Not used** in the script.
- The line `!pip install PyPDF2` → Should **not be included** inside Python code (only needed in notebooks).
- Using `texts[:50]` limits to just 50 chunks. Consider parameterizing or removing that limit unless needed for cost/testing.

## ✓ Cleaned and Well-Commented Version

```
python

# Required Libraries
import os
from PyPDF2 import PdfReader
from langchain.vectorstores.cassandra import Cassandra
from langchain.indexes.vectorstore import VectorStoreIndexWrapper
from langchain.llms import OpenAI
from langchain.embeddings import OpenAIEmbeddings
from langchain.text_splitter import CharacterTextSplitter
import cassio

# 1. Load Environment Variables or Replace With Constants
ASTRA_DB_APPLICATION_TOKEN = "AstraCS:..." # Replace with your actual token
ASTRA_DB_ID = "your-db-id" # Replace with your actual DB ID
OPENAI_API_KEY = "sk-..." # Replace with your actual OpenAI API key

# 2. Read PDF Content
pdf_reader = PdfReader("budget_speech.pdf")
raw_text = ""
for page in pdf_reader.pages:
    content = page.extract_text()
    if content:
        raw_text += content

# 3. Initialize Astra DB connection via CassIO
cassio.init(token=ASTRA_DB_APPLICATION_TOKEN, database_id=ASTRA_DB_ID)
```

```
# 4. Setup Embedding Model and Language Model
embedding = OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY)
llm = OpenAI(openai_api_key=OPENAI_API_KEY)
Copy Edit

# 5. Initialize Cassandra Vector Store
astra_vector_store = Cassandra(
    embedding=embedding,
    table_name="qa_mini_demo",
    session=None,      # Automatically handled by cassio
    keyspace=None       # Automatically handled by cassio
)

# 6. Split Text into Chunks for Vector Embedding
text_splitter = CharacterTextSplitter(
    separator="\n",
    chunk_size=800,
    chunk_overlap=200,
    length_function=len
)
texts = text_splitter.split_text(raw_text)
astra_vector_store.add_texts(texts[:50]) # Insert first 50 chunks into the database

print(f"Inserted {len(texts[:50])} chunks into vector store.")

# 7. Wrap the Vector Store for Querying
astra_vector_index = VectorStoreIndexWrapper(vectorstore=astra_vector_store)

# 8. Chat Loop for Interactive QA
first_question = True
while True:
    prompt = "Enter your question" if first_question else "What's your next question"
    query_text = input(f"\n{prompt} (or type 'quit' to exit): ").strip()

    if query_text.lower() == "quit":
        break
    if not query_text:
        continue
```



```
# 8. Chat Loop for Interactive QA
first_question = True
while True:
    prompt = "Enter your question" if first_question else "What's your next question"
    query_text = input(f"\n{prompt} (or type 'quit' to exit): ").strip()

    if query_text.lower() == "quit":
        break
    if not query_text:
        continue

    first_question = False

    print(f"\nQUESTION: \'{query_text}\'")
    answer = astra_vector_index.query(query_text, llm=llm).strip()
    print(f"ANSWER: {answer}\n")

    print("Top Relevant Chunks:")
    for doc, score in astra_vector_store.similarity_search_with_score(query_text, k=4):
        print(f"    [{score:.4f}] '{doc.page_content[:84]}...'")
```

## LCEL

## ✓ Explanation of Libraries Used

| Library                                    | Purpose   | Used? | Comment                                 |
|--|---|-------|---|
| <code>dotenv</code>                        | Load environment variables like API keys from <code>.env</code> | ✓     | Required                                |
| <code>langchain_groq</code>                | Access open-source LLMs (LLaMA3, Gemma, Mixtral) from Groq      | ✓     | Essential                               |
| <code>langchain_openai</code>              | Interface for OpenAI LLMs (not used here directly)              | ✗     | Can be removed                          |
| <code>langchain_core.messages</code>       | For building chat-style messages                                | ✓     | Used to structure system/human messages |
| <code>langchain_core.output_parsers</code> | Parses model outputs (e.g., extract plain text)                 | ✓     | Useful for chaining                     |
| <code>langchain_core.prompts</code>        | Build flexible prompt templates with variables                  | ✓     | Useful for dynamic input                |
| <code>streamlit</code>                     | For building UI apps (installed but <b>not used</b> )           | ✗     | Can be skipped unless building UI       |

## ✓ Summary of What This Code Does

1. Loads credentials from a `.env` file.
2. Sets up a Groq-hosted open-source model like `Gemma2-9b-It`.
3. Sends chat-formatted input messages (system + user).
4. Parses model response into plain text.
5. Chains together prompt template → model → output parser using LCEL.
6. Allows language-specific dynamic translation using templated prompts.

```
# -----
# 1. Load API Keys and Environment Variables

# -----
import os
from dotenv import load_dotenv
load_dotenv()
```

```
# Load API keys from .env file
openai_api_key = os.getenv("OPENAI_API_KEY")
groq_api_key = os.getenv("GROQ_API_KEY")

# -----
# 2. Set Up the LLM (Groq) - Using Gemma2 Model

# -----
from langchain_groq import ChatGroq
# Choose from: "LLaMA3-8b-8192", "Gemma2-9b-It", "Mixtral-8x7b-32768"
model = ChatGroq(model="Gemma2-9b-It", groq_api_key=groq_api_key)

# -----
# 3. Build Basic Prompt and Test Translation

# -----
from langchain_core.messages import SystemMessage, HumanMessage
# Example prompt to translate English to French
messages = [
    SystemMessage(content="Translate the following from English to French"),
    HumanMessage(content="Hello, how are you?")
]
# Invoke model directly with messages
response = model.invoke(messages)

# -----
# 4. Parse the Response to Get Just the Text

# -----
from langchain_core.output_parsers import StrOutputParser
parser = StrOutputParser()
# Convert the model's output to plain string
parsed_output = parser.invoke(response)
print("✓ Direct Translation:", parsed_output)

# -----
# 5. Chain Components with LCEL (LangChain Expression Language)

# -----
```

```
# Build chain: Model output → Parsed string
chain = model | parser
print("✓ LCEL Chained Output:", chain.invoke(messages))
# -----
# 6. Use Prompt Template for Dynamic Inputs
# -----

from langchain_core.prompts import ChatPromptTemplate
# Prompt template for any language translation
generic_template = "Translate the following into {language}:"
# Create chat prompt template with variables
prompt = ChatPromptTemplate.from_messages([
    ("system", generic_template),  ("user", "{text}") ])

# Generate full prompt message
filled_prompt = prompt.invoke({"language": "French", "text": "Hello"})
print("✓ Filled Prompt Messages:", filled_prompt.to_messages())

# -----
# 7. Final Chain: Prompt → Model → Output Parser
# -----


final_chain = prompt | model | parser
print("✓ Final Translation:", final_chain.invoke({"language": "French", "text": "Hello"}))
```

## RAG ON PDF

### ✓ Summary of Each Component & Library

| Component                                   | Purpose  |
|---|--|
| <code>streamlit</code>                      | UI layer for user interaction  |
| <code>dotenv</code>                         | Load environment variables like API keys from <code>.env</code> file |
| <code>langchain_groq.ChatGroq</code>        | Interface to access open-source LLMs (like LLaMA3) via Groq          |
| <code>HuggingFaceEmbeddings</code>          | Converts text into embeddings using HuggingFace models               |
| <code>RecursiveCharacterTextSplitter</code> | Splits long docs into chunks suitable for embedding                  |
| <code>PyPDFDirectoryLoader</code>           | Loads all PDFs from a folder into text documents                     |
| <code>FAISS</code>                          | Efficient vector similarity search library                           |
| <code>ChatPromptTemplate</code>             | Builds flexible prompts with dynamic variables                       |
| <code>create_stuff_documents_chain</code>   | Chains context documents + question → LLM                            |
| <code>create_retrieval_chain</code>         | Combines retriever and generation chain for RAG                      |
| <code>st.session_state</code>               | Stores persistent state (e.g., vectors) across Streamlit app runs    |

## 1. Setup and Configuration

- **Streamlit** is used to create a simple interactive web app.
  - **Environment variables** are loaded using `dotenv` to securely store API keys for Groq, HuggingFace, and optionally OpenAI.
  - These keys allow access to models hosted on external services without hardcoding them into the script.
- 

## 2. Model and Embeddings Initialization

- A **Groq-hosted LLaMA3-8B model** is selected to serve as the LLM (Large Language Model) for generating answers.
- A **HuggingFace embedding model** (`all-MiniLM-L6-v2`) is used to convert chunks of document text into high-dimensional vector representations. These vectors allow for efficient similarity search.

## 3. Vector Store Creation (Embedding PDFs)

This step is triggered when the user clicks the "Create Document Embedding" button.

- **PDF files are loaded** from a folder named `research_papers/`.
  - Each PDF is **split into smaller text chunks** using a recursive character-based strategy to ensure coherence and manage token limits.
  - These chunks are then **embedded using the HuggingFace model**, converting them into vectors.
  - A **FAISS vector database** is built from these vectors and stored in the app's session state for later retrieval.
- 

## 4. Streamlit UI

- The app displays a title and an input box where the user can type a question.
- A button allows users to generate the vector database from their PDF documents.
- If the vector database has been created and a question is entered, the RAG pipeline is triggered.

## 5. Retrieval-Augmented Generation (RAG) Workflow

Once the user submits a question:

- The app uses a **retriever** to find the most relevant chunks from the FAISS vector store.
  - These chunks are inserted into a **prompt template** that asks the LLM to answer based only on the retrieved context.
  - The **Groq-hosted LLaMA3 model** generates a response using the context + question.
  - The app **displays the answer** and **measures the time taken** for generating it.
- 

## 6. Document Transparency

- The app shows an expandable section where it **lists the actual document chunks** used to generate the answer.
- This builds **trust and transparency**, as users can verify which parts of the documents were used in answering their query.

## Summary of Purpose

This app allows users to **ask questions about a collection of PDF research papers**, and it answers them using a **retrieval-based AI assistant**. It combines:

- Document ingestion (PDF loading + text splitting)
- Semantic search (via HuggingFace + FAISS)
- LLM reasoning (via Groq's LLaMA3)
- User interface (via Streamlit)

```
# -----
```

```
# 1. Imports and Environment Setup
```

```
# -----
```

```
import streamlit as st
```

```
import os
```

```
import time
```

```
from dotenv import load_dotenv
```

```
# LangChain components
```

```
from langchain_groq import ChatGroq
```

```

from langchain_huggingface import HuggingFaceEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain_core.prompts import ChatPromptTemplate
from langchain_chains import create_retrieval_chain
from langchain_community.vectorstores import FAISS
from langchain_community.document_loaders import PyPDFDirectoryLoader

# Load environment variables (API keys etc.)
load_dotenv()

# Set environment variables
os.environ["OPENAI_API_KEY"] = os.getenv("OPENAI_API_KEY") # Optional, not used here
os.environ["GROQ_API_KEY"] = os.getenv("GROQ_API_KEY")
os.environ["HF_TOKEN"] = os.getenv("HF_TOKEN") # For HuggingFace embeddings

# -----
# 2. Initialize LLM and Embedding Model
# -----
# Groq LLM using LLaMA3 8B model
llm = ChatGroq(groq_api_key=os.environ["GROQ_API_KEY"], model_name="Llama3-8b-8192")

# Embedding model (from HuggingFace)
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

# Prompt template for question answering
prompt = ChatPromptTemplate.from_template("""
Answer the question using ONLY the provided context below.
Provide the most accurate response possible.

<context>
{context}
</context>

Question: {input}
""")

# -----
# 3. Vector Embedding Function
# -----
def create_vector_embedding():
    """
    Load and split PDF documents, then embed them using HuggingFace and store in FAISS vector DB.
    """

```

```

if "vectors" not in st.session_state:
    # Load PDF documents from a directory
    st.session_state.loader = PyPDFDirectoryLoader("research_papers")
    st.session_state.docs = st.session_state.loader.load()

    # Split documents into smaller chunks
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
    split_docs = text_splitter.split_documents(st.session_state.docs[:50]) # Use first 50 docs

    # Store embeddings in FAISS vector DB
    st.session_state.vectors = FAISS.from_documents(split_docs, embeddings)

# -----
# 4. Streamlit UI
# -----
st.title("🔍 RAG Q&A over Research Papers (Groq + LLaMA3)")

# User query input
user_prompt = st.text_input("Ask a question based on the research papers:")

# Button to build vector DB from PDF documents
if st.button("📦 Create Document Embedding"):
    create_vector_embedding()
    st.success("✅ Vector database is ready!")

# -----
# 5. Run RAG Chain on User Query
# -----
if user_prompt and "vectors" in st.session_state:
    # Build document retrieval and response generation pipeline
    document_chain = create_stuff_documents_chain(llm, prompt)
    retriever = st.session_state.vectors.as_retriever()
    retrieval_chain = create_retrieval_chain(retriever, document_chain)

    # Time the response
    start = time.process_time()
    response = retrieval_chain.invoke({"input": user_prompt})
    response_time = time.process_time() - start

    # Show answer
    st.markdown(f"### 📄 Answer:\n{response['answer']} ")
    st.caption(f"⌚ Response time: {response_time:.2f} seconds")

    # Expand to show documents used
    with st.expander("📄 Document Similarity Results"):
        for i, doc in enumerate(response['context']):

```

```
st.write(doc.page_content)
st.markdown("---")
```

This was for huggingface , but also you can use with openai

## CONVERSATIONAL – QA-RAG

### What This Application Does — Overview

This Streamlit app creates a **Conversational RAG (Retrieval-Augmented Generation)** system that:

- Lets users upload one or more **PDFs**.
- Converts PDFs into text, splits into chunks, and embeds using HuggingFace embeddings.
- Stores the embeddings in a **Chroma vector store**.
- Uses a **Groq-hosted LLM (Gemma2-9b-It)** for answering user questions.
- Incorporates **chat history** to allow for follow-up questions and contextual understanding.
- Uses **LangChain's RunnableWithMessageHistory** to tie everything together with session management.

## Step-by-Step Explanation (No Code)

### 1. Environment Setup:

- Load necessary API keys (Groq and HuggingFace).
- Import all required libraries from LangChain and Streamlit.

### 2. User Interface (UI):

- Title and description using `Streamlit`.
- Prompt user for Groq API key and optional session ID (used to track conversations).

### 3. PDF Upload:

- User uploads one or more PDFs.
- PDFs are saved temporarily and parsed using `PyPDFLoader` to extract text.

### 4. Text Processing and Embedding:

- The extracted text is split into manageable chunks.
- These chunks are embedded using `HuggingFaceEmbeddings`.
- A vector store is created using `Chroma` to support semantic search.

### 5. Retriever Configuration with History Awareness:

- A special prompt is created to "re-contextualize" follow-up questions using chat history.
- LangChain's `create_history_aware_retriever` wraps the standard retriever to reformulate user input with chat context.

### 6. Q&A Chain:

- Another prompt is created to answer the standalone question using the context retrieved from the documents.
- A RAG chain is assembled using the history-aware retriever and the Q&A logic.

### 7. Session-Based Chat:

- All interactions are tracked per session using `ChatMessageHistory`.
- This allows context-aware multi-turn conversation.

### 8. Display:

- User inputs their query.
- The RAG system returns a concise answer based on the document.
- Both the assistant's response and chat history are shown on the UI.

## Libraries — What They Do

| Library                        | Purpose  |
|--------------------------------|--|
| streamlit                      | Frontend for user interface                          |
| dotenv                         | Loads API keys from <code>.env</code>                |
| ChatGroq                       | Interface to Groq-hosted open-source LLMs like Gemma |
| HuggingFaceEmbeddings          | Embeds document chunks using transformer models      |
| PyPDFLoader                    | Loads and extracts text from PDF files               |
| RecursiveCharacterTextSplitter | Splits long documents into overlapping chunks        |
| Chroma                         | Vector database for semantic search over chunks      |
| ChatPromptTemplate             | Structured prompt formatting                         |
| MessagesPlaceholder            | Inserts chat history into prompts                    |
| ChatMessageHistory             | Stores messages per session                          |
| RunnableWithMessageHistory     | Chains retrieval + chat logic with stateful memory   |

```
# -----
# 1. Imports and Setup
# -----
import os
import streamlit as st
from dotenv import load_dotenv
from langchain_groq import ChatGroq
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_core.chat_history import BaseChatMessageHistory
from langchain_community.chat_message_histories import ChatMessageHistory
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_community.document_loaders import PyPDFLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_chroma import Chroma
from langchain.chains import (
    create_history_aware_retriever,
    create_retrieval_chain,
```

```

)
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain_core.runnables.history import RunnableWithMessageHistory

# Load environment variables
load_dotenv()
os.environ["HF_TOKEN"] = os.getenv("HF_TOKEN")

# Initialize embedding model
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

# -----
# 2. Streamlit UI Setup
# -----
st.title("❑ Conversational RAG with PDFs")
st.write("Upload PDFs and chat with their content across sessions.")

# User enters API key and optional session ID
api_key = st.text_input("❑ Enter your Groq API key:", type="password")
session_id = st.text_input("❑ Session ID:", value="default_session")

# Store per-session chat history
if "store" not in st.session_state:
    st.session_state.store = {}

# -----
# 3. If API key is provided, proceed
# -----
if api_key:
    llm = ChatGroq(groq_api_key=api_key, model_name="Gemma2-9b-It")

    uploaded_files = st.file_uploader("❑ Upload PDFs", type="pdf", accept_multiple_files=True)

    if uploaded_files:
        documents = []

        # Save and load all uploaded PDFs
        for uploaded_file in uploaded_files:
            with open("temp.pdf", "wb") as f:
                f.write(uploaded_file.getvalue())
            loader = PyPDFLoader("temp.pdf")
            documents.extend(loader.load())

        # Split documents and embed
        splitter = RecursiveCharacterTextSplitter(chunk_size=5000, chunk_overlap=500)
        splits = splitter.split_documents(documents)
        vectorstore = Chroma.from_documents(documents=splits, embedding=embeddings)

```

```

retriever = vectorstore.as_retriever()

# -----
# 4. History-aware Retriever & RAG Chain
# -----


# Reformulate follow-up questions
contextualize_prompt = ChatPromptTemplate.from_messages([
    ("system", "Reformulate the question using chat history context if necessary."),
    MessagesPlaceholder("chat_history"),
    ("human", "{input}"),
])
history_aware_retriever = create_history_aware_retriever(llm, retriever,
contextualize_prompt)

# Q&A Prompt with context
qa_prompt = ChatPromptTemplate.from_messages([
    ("system", "Use the following context to answer. Be concise.\n\n{context}"),
    MessagesPlaceholder("chat_history"),
    ("human", "{input}"),
])
qa_chain = create_stuff_documents_chain(llm, qa_prompt)

# Final RAG chain with retriever and QA logic
rag_chain = create_retrieval_chain(history_aware_retriever, qa_chain)

# Track session-specific message history
def get_session_history(session: str) -> BaseChatMessageHistory:
    if session not in st.session_state.store:
        st.session_state.store[session] = ChatMessageHistory()
    return st.session_state.store[session]

conversational_chain = RunnableWithMessageHistory(
    rag_chain,
    get_session_history,
    input_messages_key="input",
    history_messages_key="chat_history",
    output_messages_key="answer",
)
# -----
# 5. User Query Handling
# -----


user_input = st.text_input("Ask your question:")

if user_input:
    session_history = get_session_history(session_id)

```

```

response = conversational_chain.invoke(
    {"input": user_input},
    config={"configurable": {"session_id": session_id}},
)
st.markdown(f"### ☑ Assistant:\n{response['answer']}")  

st.caption("📁 Full Chat History")
for msg in session_history.messages:
    st.markdown(f"- **{msg.type.capitalize()}:** {msg.content}")
else:
    st.warning("⚠ Please enter your Groq API key.")

```

## SEARCH ENGINE WTH TOOLS AND AGENTS

### Step-by-Step Explanation (No Code)

#### 1. Tool Creation (Wikipedia, Arxiv, LangSmith RAG)

- Wikipedia Tool:
    - Uses `WikipediaAPIWrapper` to fetch a short summary (top 1 result, max 250 chars).
  - Arxiv Tool:
    - Uses `ArxivAPIWrapper` to fetch paper metadata from arXiv.org, also limited to 250 chars.
  - Custom RAG Tool:
    - Loads content from LangSmith's website.
    - Splits and embeds it using `OpenAIEmbeddings` and stores it in a `FAISS` vector DB.
    - Wraps the vector retriever as a tool using `create_retriever_tool`.
- 

#### 2. Tool Aggregation

- Combines the above tools into a list: `[WikipediaTool, ArxivTool, LangSmithRetrieverTool]`

### 3. Agent Setup

- Initializes a **Groq-hosted LLM** (`Llama3-8b-8192`) as the brain behind the agent.
  - Loads a **multi-tool-aware prompt** from LangChain's Hub (`openai-functions-agent`) that knows how to reason over tools.
  - Builds the agent using `create_openai_tools_agent()` —this enables it to decide **which tool to call** based on user query.
- 

### 4. Agent Execution

- An `AgentExecutor` is initialized to run this agent, with verbosity enabled.
- The agent is **invoked with natural language queries**, like:
  - "Tell me about LangSmith" → Likely uses your custom retriever tool.
  - "What is machine learning" → Likely uses Wikipedia tool.
  - "What's the paper 1706.03762 about?" → Uses Arxiv tool.

## 📦 Summary of Each Library:

| Library / Component                         | Purpose   |
|---|---|
| <code>langchain_community.tools</code>      | Built-in wrappers for external data sources (Wikipedia, Arxiv). |
| <code>WikipediaAPIWrapper</code>            | Queries Wikipedia programmatically.                             |
| <code>ArxivAPIWrapper</code>                | Queries Arxiv academic papers.                                  |
| <code>WebBaseLoader</code>                  | Scrapes websites and loads text.                                |
| <code>RecursiveCharacterTextSplitter</code> | Breaks long texts into manageable chunks.                       |
| <code>FAISS</code>                          | Local vector DB to store and retrieve document embeddings.      |
| <code>OpenAIEmbeddings</code>               | Converts text to dense vector embeddings.                       |
| <code>create_retriever_tool</code>          | Converts a retriever into a usable tool for agents.             |
| <code>chatGroq</code>                       | Connects to Groq-hosted LLMs like LLaMA3 or Gemma.              |
| <code>hub.pull(...)</code>                  | Pulls pre-built LangChain prompt templates.                     |
| <code>create_openai_tools_agent</code>      | Builds an LLM agent that can use multiple tools.                |
| <code>AgentExecutor</code>                  | Executes the agent with tools and user queries.                 |

```
# -----
# 1. Imports and Environment Setup
# -----
import os
from dotenv import load_dotenv
load_dotenv()

from langchain_community.tools import ArxivQueryRun, WikipediaQueryRun
from langchain_community.utilities import WikipediaAPIWrapper, ArxivAPIWrapper
from langchain_community.document_loaders import WebBaseLoader
from langchain_community.vectorstores import FAISS
from langchain.tools.retriever import create_retriever_tool
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_openai import OpenAIEmbeddings
from langchain_groq import ChatGroq
from langchain import hub
from langchain.agents import create_openai_tools_agent, AgentExecutor

# -----
```

```

# 2. Tool Creation
# ----

# Wikipedia Tool
wiki_wrapper = WikipediaAPIWrapper(top_k_results=1, doc_content_chars_max=250)
wiki_tool = WikipediaQueryRun(api_wrapper=wiki_wrapper)

# ArXiv Tool
arxiv_wrapper = ArxivAPIWrapper(top_k_results=1, doc_content_chars_max=250)
arxiv_tool = ArxivQueryRun(api_wrapper=arxiv_wrapper)

# Custom RAG Tool from LangSmith Docs
web_loader = WebBaseLoader("https://docs.smith.langchain.com/")
raw_docs = web_loader.load()
splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
documents = splitter.split_documents(raw_docs)

vectordb = FAISS.from_documents(documents, OpenAIEmbeddings())
retriever = vectordb.as_retriever()
langsmith_tool = create_retriever_tool(
    retriever,
    name="langsmith-search",
    description="Search any information about LangSmith."
)

# Combine all tools
tools = [wiki_tool, arxiv_tool, langsmith_tool]

# -----
# 3. LLM Initialization (Groq)
# -----

groq_api_key = os.getenv("GROQ_API_KEY")
llm = ChatGroq(groq_api_key=groq_api_key, model_name="Llama3-8b-8192")

# -----
# 4. Agent Setup with Multi-Tool Prompt
# -----

prompt = hub.pull("hwchase17/openai-functions-agent")
agent = create_openai_tools_agent(llm=llm, tools=tools, prompt=prompt)

# -----
# 5. Agent Executor - Handles Inputs Dynamically
# -----

agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

# -----
# 6. Agent Execution - Sample Queries

```

```
# -----
agent_executor.invoke({"input": "Tell me about LangSmith"})
agent_executor.invoke({"input": "What is machine learning?"})
agent_executor.invoke({"input": "What's the paper 1706.03762 about?"})
```

## What's Happening (Conceptual Breakdown):

### 1. Tools Setup:

- You define tools for Wikipedia, arXiv, LangSmith documentation (RAG), and plan to add Google Search and custom PDF retrievers.
- Each tool is wrapped to plug into LangChain's agent framework via a retriever or query interface.

### 2. Retriever Creation:

- Documents are loaded (e.g., from web or PDFs), split into chunks, and embedded using an embedding model.
- These embeddings are stored in FAISS (a vector database) to support semantic search.

### 3. Memory Handling:

- LangChain stores chat history using `ChatMessageHistory`, allowing the system to reference past user inputs and maintain context across messages.

### 4. Agent Formation:

- All tools are given to an OpenAI-function-style agent (via prompt templates).
- The agent decides which tool to use based on the input question.

### 5. Execution:

- Input queries are processed by the `AgentExecutor`, which uses the agent + tools + memory to provide contextual answers.

```
# -----
# Imports and Environment Setup
# -----
import os
from dotenv import load_dotenv
import openai
import streamlit as st
from langchain_groq import ChatGroq
from langchain.agents import create_openai_tools_agent, AgentExecutor
from langchain import hub
from langchain_community.tools import ArxivQueryRun, WikipediaQueryRun
from langchain_community.utilities import WikipediaAPIWrapper, ArxivAPIWrapper
from langchain_community.document_loaders import WebBaseLoader, PyPDFLoader
```

```

from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEMBEDDINGS
from langchain.tools.retriever import create_retriever_tool
from langchain_core.chat_history import BaseChatMessageHistory
from langchain_community.chat_message_histories import ChatMessageHistory
from langchain_core.runnables.history import RunnableWithMessageHistory

# -----
# Load API Keys
# -----
load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")
groq_api_key = os.getenv("GROQ_API_KEY")

# -----
# Initialize LLM
# -----
llm = ChatGroq(groq_api_key=groq_api_key, model_name="Llama3-8b-8192")

# -----
# Define External Query Tools
# -----
wiki = WikipediaQueryRun(api_wrapper=WikipediaAPIWrapper(top_k_results=1,
doc_content_chars_max=250))
arxiv = ArxivQueryRun(api_wrapper=ArxivAPIWrapper(top_k_results=1,
doc_content_chars_max=250))

# -----
# RAG Tool (LangSmith Docs)
# -----
web_docs = WebBaseLoader("https://docs.smith.langchain.com/").load()
web_splits = RecursiveCharacterTextSplitter(chunk_size=1000,
chunk_overlap=200).split_documents(web_docs)
web_vectordb = FAISS.from_documents(web_splits, OpenAIEMBEDDINGS())
web_retriever_tool = create_retriever_tool(web_vectordb.as_retriever(), "langsmith-search",
"Search LangSmith docs")

# -----
# PDF Retriever Tool (Custom PDF Uploads)
# -----
def create_pdf_tool(pdf_path: str, tool_name="pdf-search", description="Search PDF content"):
    pdf_docs = PyPDFLoader(pdf_path).load()
    pdf_splits = RecursiveCharacterTextSplitter(chunk_size=1000,
chunk_overlap=200).split_documents(pdf_docs)
    pdf_vectordb = FAISS.from_documents(pdf_splits, OpenAIEMBEDDINGS())
    return create_retriever_tool(pdf_vectordb.as_retriever(), tool_name, description)

```

```

# Example: create_pdf_tool("research_papers/sample.pdf")

# -----
# Tools Aggregation
# -----
tools = [wiki, arxiv, web_retriever_tool] # Add pdf_tool manually if needed

# -----
# Agent Setup
# -----
prompt = hub.pull("hwchase17/openai-functions-agent")
agent = create_openai_tools_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

# -----
# Conversation History Memory
# -----
def get_session_history(session_id: str) -> BaseChatMessageHistory:
    if "store" not in st.session_state:
        st.session_state.store = {}
    if session_id not in st.session_state.store:
        st.session_state.store[session_id] = ChatMessageHistory()
    return st.session_state.store[session_id]

conversational_agent = RunnableWithMessageHistory(
    agent_executor,
    get_session_history,
    input_messages_key="input",
    history_messages_key="chat_history",
    output_messages_key="output"
)
# -----
# Example Invocation
# -----
# response = conversational_agent.invoke({"input": "What's LangSmith?"}, config={"configurable": {"session_id": "user1"}})

```

## Summary of Libraries and Their Role:

| Library / Tool  | Purpose                                       |
|---|---|
| <code>langchain_groq.ChatGroq</code>                            | Interfaces with Groq's LLaMA3 model           |
| <code>langchain_community.tools</code>                          | Built-in tools like Wikipedia/arXiv           |
| <code>WikipediaAPIWrapper</code> , <code>ArxivAPIWrapper</code> | Query wrappers for fetching external data     |
| <code>FAISS</code>  | Vector DB for storing and querying embeddings |
| <code>OpenAIEMBEDDINGS</code>                                   | Converts text into vector embeddings          |
| <code>WebBaseLoader</code> , <code>PyPDFLoader</code>           | Loads text from web pages and PDFs            |
| <code>RecursiveCharacterTextSplitter</code>                     | Splits documents into overlapping chunks      |
| <code>create_retriever_tool</code>                              | Wraps retrievers into agent-compatible tools  |
| <code>ChatMessageHistory</code>                                 | Maintains chat memory                         |
| <code>RunnableWithMessageHistory</code>                         | Binds memory to agent execution               |
| <code>hub.pull(...)</code>                                      | Loads predefined prompt templates             |
| <code>AgentExecutor</code>                                      | Executes agent decisions using tools          |

## Search engine Agent + Tools

## Understanding the Existing Code

### **Objective:**

Create a Streamlit-based chatbot that leverages LangChain agents to answer user queries using tools like Arxiv, Wikipedia, and DuckDuckGo Search.

### **Workflow Breakdown:**

#### **1. Imports and Environment Setup:**

- **Streamlit:** For building the web interface.
- **LangChain Components:** Including agents, tools, and callback handlers.
- **Environment Variables:** Loaded using `dotenv` for secure API key management.

#### **2. Tool Initialization:**

- **Arxiv:** Fetches academic papers.
- **Wikipedia:** Retrieves encyclopedia articles.
- **DuckDuckGo Search:** Performs web searches.

#### **3. Streamlit Interface:**

- **Title and Description:** Sets up the app's header.
- **Sidebar:** Allows users to input their Groq API key.
- **Chat History:** Maintains a list of messages exchanged during the session.
- **User Input:** Captures user queries via `st.chat_input`.

---

#### **4. Agent Setup:**

- **Language Model:** Initialized using Groq's Llama3 model.
- **Tools:** Combined into a list for the agent's use.
- **Agent Initialization:** Configured with the tools and language model.
- **Callback Handler:** `StreamlitCallbackHandler` displays the agent's thought process in the UI.

#### **5. Response Generation:**

- **Agent Execution:** Processes the user's query and generates a response.
- **Chat History Update:** Appends the assistant's response to the session's message history.

---

## Enhanced Code with Additional Features

Now, let's enhance the application by integrating:

1. **Google Search Tool:** For comprehensive web searches.
2. **Custom PDF Retriever:** Allows users to upload PDFs and query their content.
3. **Conversation Memory:** Maintains context across multiple interactions.

## Summary of Enhancements

1. **Google Search Tool:** Placeholder added for future integration, allowing real-time web searches.
2. **Custom PDF Retrievers:**
  - Users can upload multiple PDFs.
  - Each PDF is processed into a retriever tool, enabling the agent to search and reference their content during conversations.
3. **Conversation Memory:**
  - Maintains a history of interactions, allowing the agent to provide contextually relevant responses.

## Library and Function Overview

| Library / Component  | Functionality   |
|--|---|
| <code>streamlit</code>   | Builds the web interface for user interaction.                |
| <code>langchain_groq.ChatGroq</code>                                       | Interfaces with Groq's Llama3 language model.                 |
| <code>langchain.agents.initialize_agent</code>                             | Initializes the agent with specified tools and LLM.           |
| <code>langchain.callbacks.StreamlitCallbackHandler</code>                  | Displays the agent's thought process in the Streamlit UI.     |
| <code>langchain_community.utilities</code>                                 | Provides wrappers for external APIs like Arxiv and Wikipedia. |
| <code>langchain_community.tools</code>                                     | Offers tools for querying various data sources.               |
| <code>langchain_community.document_loaders.PyPDFLoader</code>              | Loads and processes PDF documents.                            |
| <code>langchain_text_splitters.RecursiveCharacterTextsplitter</code>       | Splits documents into manageable chunks for processing.       |
| <code>langchain_community.vectorstores.FAISS</code>                        | Stores and retrieves document embeddings for semantic search. |
| <code>langchain_openai.OpenAIEMBEDDINGS</code>                             | Generates embeddings for document chunks.                     |
| <code>langchain.tools.retriever.create_retriever_tool</code>               | Converts retrievers into tools usable by the agent.           |
| <code>langchain_community.chat_message_histories.ChatMessageHistory</code> | Maintains the history of chat interactions.                   |

# -----

```

# Imports and Environment Setup
# -----
import os
import streamlit as st
from dotenv import load_dotenv
from langchain_groq import ChatGroq
from langchain.agents import initialize_agent, AgentType
from langchain.callbacks import StreamlitCallbackHandler
from langchain_community.utilities import ArxivAPIWrapper, WikipediaAPIWrapper
from langchain_community.tools import ArxivQueryRun, WikipediaQueryRun,
DuckDuckGoSearchRun
from langchain_community.document_loaders import PyPDFLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEMBEDDINGS
from langchain.tools.retriever import create_retriever_tool
from langchain_community.chat_message_histories import ChatMessageHistory
from langchain_core.chat_history import BaseChatMessageHistory
from langchain_core.runnables.history import RunnableWithMessageHistory

# Load environment variables
load_dotenv()
GROQ_API_KEY = os.getenv("GROQ_API_KEY")
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")

# -----
# Streamlit Interface Setup
# -----
st.title("LangChain - Chat with Enhanced Search")
st.sidebar.title("Settings")
api_key = st.sidebar.text_input("Enter your Groq API Key:", type="password")

# Initialize session state for chat history and PDF retrievers
if "messages" not in st.session_state:
    st.session_state["messages"] = [
        {"role": "assistant", "content": "Hi, I'm a chatbot who can search the web and your uploaded PDFs. How can I help you?"}
    ]
if "pdf_retrievers" not in st.session_state:
    st.session_state["pdf_retrievers"] = {}

# Display chat history
for msg in st.session_state.messages:
    st.chat_message(msg["role"]).write(msg["content"])

# -----
# Tool Initialization

```

```

# -----
# Arxiv Tool
arxiv_wrapper = ArxivAPIWrapper(top_k_results=1, doc_content_chars_max=200)
arxiv = ArxivQueryRun(api_wrapper=arxiv_wrapper)

# Wikipedia Tool
wiki_wrapper = WikipediaAPIWrapper(top_k_results=1, doc_content_chars_max=200)
wiki = WikipediaQueryRun(api_wrapper=wiki_wrapper)

# DuckDuckGo Search Tool
duckduckgo_search = DuckDuckGoSearchRun(name="DuckDuckGo Search")

# Placeholder for Google Search Tool (to be implemented)
# google_search = GoogleSearchRun(name="Google Search")

# -----
# PDF Upload and Retriever Setup
# -----
uploaded_files = st.sidebar.file_uploader("Upload PDF files", type="pdf",
accept_multiple_files=True)
if uploaded_files:
    for uploaded_file in uploaded_files:
        file_name = uploaded_file.name
        if file_name not in st.session_state["pdf_retrievers"]:
            # Save uploaded PDF temporarily
            temp_path = f"./temp_{file_name}"
            with open(temp_path, "wb") as f:
                f.write(uploaded_file.getbuffer())

            # Load and process PDF
            loader = PyPDFLoader(temp_path)
            documents = loader.load()
            text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
            splits = text_splitter.split_documents(documents)
            vectordb = FAISS.from_documents(splits, OpenAIEmbeddings())
            retriever = vectordb.as_retriever()
            retriever_tool = create_retriever_tool(retriever, f'{file_name}_search', f'Search content
in {file_name}')
            st.session_state["pdf_retrievers"][file_name] = retriever_tool

# Combine all tools
tools = [duckduckgo_search, arxiv, wiki] + list(st.session_state["pdf_retrievers"].values())

# -----
# Chat Input and Agent Response
# -----
if prompt := st.chat_input(placeholder="Ask me anything..."):

```

```

st.session_state.messages.append({"role": "user", "content": prompt})
st.chat_message("user").write(prompt)

# Initialize LLM
llm = ChatGroq(groq_api_key=api_key, model_name="Llama3-8b-8192", streaming=True)

# Initialize Agent
agent = initialize_agent(
    tools,
    llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    handle_parsing_errors=True
)

# Callback Handler for Streamlit
with st.chat_message("assistant"):
    st_cb = StreamlitCallbackHandler(st.container(), expand_new_thoughts=False)
    response = agent.run(prompt, callbacks=[st_cb])
    st.session_state.messages.append({"role": "assistant", "content": response})
    st.write(response)

```

## CHAT + SQL

### **Sqlite3**

```

import sqlite3

## connect to sqllite
connection=sqlite3.connect("student.db")

##create a cursor object to insert record,create table
cursor=connection.cursor()

## create the table
table_info=""""
create table STUDENT(NAME VARCHAR(25),CLASS VARCHAR(25),
SECTION VARCHAR(25),MARKS INT)
"""

cursor.execute(table_info)

## Insert some more records

```

```
cursor.execute("Insert Into STUDENT values('Krish','Data Science','A',90)")  
cursor.execute("Insert Into STUDENT values('John','Data Science','B',100)")  
cursor.execute("Insert Into STUDENT values('Mukesh','Data Science','A',86)")  
cursor.execute("Insert Into STUDENT values('Jacob','DEVOPS','A',50)")  
cursor.execute("Insert Into STUDENT values('Dipesh','DEVOPS','A',35)")  
  
## Display all the records  
print("The inserted records are")  
data=cursor.execute("Select * from STUDENT")  
for row in data:  
    print(row)  
  
## Commit your changes in the database  
connection.commit()  
connection.close()
```

## Chat+SQL

## 1. What is Happening Here?

This Streamlit app allows users to **chat with a SQL database** (either a local SQLite DB or a remote MySQL DB) using **LangChain's SQL Agent** powered by the **Groq Llama3 model**. The chatbot interprets natural language questions and generates SQL queries to retrieve the corresponding data.

---

## 2. Step-by-Step Explanation (Without Code)

### 1. User selects the database:

- Can choose between a local SQLite (`student.db`) or a remote MySQL DB.
- Provides connection details (host, user, password, db name) if using MySQL.

### 2. Groq API key is entered to use the LLM (Llama3-8b-8192).

### 3. The selected database is configured:

- SQLite is connected in read-only mode using SQLAlchemy and `sqlite3`.
- MySQL is connected using SQLAlchemy with MySQL connector.

### 4. A LangChain SQL agent is created using:

- The LLM from Groq
- A SQL Toolkit that lets the agent understand the database schema

### 5. User sends a natural language query (like "Show me all students with GPA > 3.5").

### 6. The agent interprets the query, converts it to SQL, runs it on the DB, and returns the result as a response.

### 7. Chat history is maintained and displayed using Streamlit's `chat_input` and session state.

## 3. Overall Process Flow

Input → (LangChain Agent w/ SQL Toolkit + LLM) → Generates SQL → Executes SQL on DB → Returns Answer → Displays in Chat

## Summary of Libraries and Their Purpose

| Library   | Functionality   |
|---|---|
| <code>streamlit</code>  | Builds the chat interface and handles user input/output                   |
| <code>langchain_groq.ChatGroq</code>                            | Connects to Groq's LLM (Llama3) for natural language understanding        |
| <code>langchain.sql_database.SQLDatabase</code>                 | Abstract layer over SQL databases (supports SQLite, MySQL, Postgres etc.) |
| <code>langchain.agents.create_sql_agent</code>                  | Creates an agent that can convert natural language into SQL               |
| <code>langchain.agents.agent_toolkits.SQLDatabaseToolkit</code> | Allows the agent to understand database schema and run queries            |
| <code>sqlalchemy.create_engine</code>                           | Handles actual DB connections for both SQLite and MySQL                   |
| <code>sqlite3</code>  | Provides low-level access to SQLite databases                             |
| <code>StreamlitCallbackHandler</code>                           | Used to show intermediate steps (thoughts, reasoning) in the chat UI      |

```

import streamlit as st
from pathlib import Path
from sqlalchemy import create_engine
import sqlite3
from langchain.sql_database import SQLDatabase
from langchain.agents import create_sql_agent, AgentType
from langchain.agents.agent_toolkits import SQLDatabaseToolkit
from langchain_groq import ChatGroq
from langchain.callbacks import StreamlitCallbackHandler

# Streamlit page settings
st.set_page_config(page_title="LangChain: Chat with SQL DB", page_icon="❑")
st.title("❑ LangChain: Chat with SQL Database")

# Options to select database
DB_LOCAL = "SQLite - student.db"
DB_MYSQL = "MySQL"
db_choice = st.sidebar.radio("Choose your database:", [DB_LOCAL, DB_MYSQL])

# Groq API Key input

```

```

api_key = st.sidebar.text_input("Groq API Key", type="password")

# MySQL DB connection inputs if selected
if db_choice == DB_MYSQL:
    mysql_host = st.sidebar.text_input("MySQL Host")
    mysql_user = st.sidebar.text_input("MySQL User")
    mysql_password = st.sidebar.text_input("MySQL Password", type="password")
    mysql_db = st.sidebar.text_input("MySQL Database Name")

# Ensure API Key is provided
if not api_key:
    st.info("Please enter your Groq API key to continue.")
    st.stop()

# Initialize LLM from Groq (LLaMA3)
llm = ChatGroq(groq_api_key=api_key, model_name="Llama3-8b-8192", streaming=True)

# Cached DB configuration (avoids reconnecting on every rerun)
@st.cache_resource(ttl="2h")
def configure_database():
    if db_choice == DB_LOCAL:
        # Load local SQLite DB in read-only mode
        db_path = (Path(__file__).parent / "student.db").absolute()
        connect = lambda: sqlite3.connect(f"file:{db_path}?mode=ro", uri=True)
        engine = create_engine("sqlite://", creator=connect)
    else:
        # Connect to MySQL using user inputs
        if not all([mysql_host, mysql_user, mysql_password, mysql_db]):
            st.error("Please complete all MySQL connection fields.")
            st.stop()
        engine =
create_engine(f"mysql+mysqlconnector://{mysql_user}:{mysql_password}@{mysql_host}/{mysql_db}")

    return SQLDatabase(engine)

# Initialize SQL database connection
db = configure_database()

# Create SQL Toolkit for the agent
toolkit = SQLDatabaseToolkit(db=db, llm=llm)

# Create SQL Agent with toolkit
agent = create_sql_agent(
    llm=llm,
    toolkit=toolkit,
    agent_type=AgentType.ZERO_SHOT_DESCRIPTION,
)

```

```

verbose=True
)

# Initialize or reset chat messages
if "messages" not in st.session_state or st.sidebar.button("Clear message history"):
    st.session_state["messages"] = [{"role": "assistant", "content": "How can I help you query your database?"}]

# Display chat history
for msg in st.session_state["messages"]:
    st.chat_message(msg["role"]).write(msg["content"])

# Input field for user query
user_query = st.chat_input("Ask a question about your database...")

# Process user query
if user_query:
    st.session_state["messages"].append({"role": "user", "content": user_query})
    st.chat_message("user").write(user_query)

with st.chat_message("assistant"):
    callback = StreamlitCallbackHandler(st.container())
    try:
        response = agent.run(user_query, callbacks=[callback])
    except Exception as e:
        response = f"Error: {e}"
    st.session_state["messages"].append({"role": "assistant", "content": response})
    st.write(response)

```

### Final corrected code:

```

import streamlit as st
from pathlib import Path
from sqlalchemy import create_engine
import sqlite3

from langchain.sql_database import SQLDatabase
from langchain.agents import create_sql_agent, AgentType
from langchain.agents.agent_toolkits import SQLDatabaseToolkit
from langchain_groq import ChatGroq
from langchain.callbacks import StreamlitCallbackHandler

# — UI setup —
st.set_page_config(page_title="Chat with SQL DB", page_icon="❑")

```

```

st.title("□ Chat with SQL Database")

# — Select DB —
DB_LOCAL, DB_MYSQL = "SQLite", "MySQL"
db_choice = st.sidebar.radio("Database:", [DB_LOCAL, DB_MYSQL])

# — Credentials —
api_key = st.sidebar.text_input("Groq API Key", type="password")
if db_choice == DB_MYSQL:
    mysql_host = st.sidebar.text_input("Host")
    mysql_user = st.sidebar.text_input("User")
    mysql_password = st.sidebar.text_input("Password", type="password")
    mysql_db = st.sidebar.text_input("Database")

# — Must have API key —
if not api_key:
    st.info("Enter your Groq API key.")
    # Still render UI so they can enter it
else:
    # — Initialize LLM —
    llm = ChatGroq(groq_api_key=api_key, model_name="Llama3-8b-8192")

    # — Configure DB (cached) —
    @st.cache_resource(ttl="2h")
    def get_db():
        if db_choice == DB_LOCAL:
            path = (Path(__file__).parent / "student.db").absolute()
            conn = lambda: sqlite3.connect(f"file:{path}?mode=ro", uri=True)
            engine = create_engine("sqlite://", creator=conn)
        else:
            if not all([mysql_host, mysql_user, mysql_password, mysql_db]):
                st.error("Complete all MySQL fields.")
                return None
            engine = create_engine(
                f"mysql+mysqlconnector://{mysql_user}:{mysql_password}@{mysql_host}/{mysql_db}"
            )
        return SQLDatabase(engine)

    db = get_db()
    if db:
        # — Build agent —
        toolkit = SQLDatabaseToolkit(db=db, llm=llm)
        agent = create_sql_agent(
            llm=llm,
            toolkit=toolkit,
            agent_type=AgentType.ZERO_SHOT_DESCRIPTION,

```

```

    verbose=True,
)
# — Chat history —
if "messages" not in st.session_state or st.sidebar.button("Clear history"):
    st.session_state.messages = [{"role": "assistant", "content": "How can I help?"}]

for m in st.session_state.messages:
    st.chat_message(m["role"]).write(m["content"])

# — User input —
if user_q := st.chat_input("Ask the database..."):
    st.session_state.messages.append({"role": "user", "content": user_q})
    st.chat_message("user").write(user_q)

with st.chat_message("assistant"):
    cb = StreamlitCallbackHandler(st.container())
    resp = agent.run(user_q, callbacks=[cb])
    st.session_state.messages.append({"role": "assistant", "content": resp})
    st.write(resp)

```

## Connecting to the local Ollama

### 3. Swapping in an Ollama LLM

If you have Ollama running locally and want to use, say, the Gemma2 model, install `langchain-ollama` and replace the LLM initialization:

```

python
from langchain_ollama import Ollama

# Instead of ChatGroq...
llm = Ollama(model="gemma:2b", embedding_model="mxbai-embed-large")

```

| Component                        | Role   |
|----------------------------------|--|
| Streamlit                        | UI framework for chat interface                          |
| SQLDatabase / SQLDatabaseToolkit | Connects to SQLite/MySQL and exposes schema to the agent |
| create_sql_agent                 | Builds an agent to translate NL → SQL → execution        |
| ChatGroq / Ollama                | LLM backends (Groq-hosted or local Ollama)               |
| StreamlitCallbackHandler         | Streams the agent's "thinking" steps into the UI         |

## High-level Purpose

This is a **Streamlit app** where users can:

- Pick a database (`SQLite` or `MySQL`)
  - Enter their credentials (for Groq and optionally MySQL)
  - Ask **natural language** questions about the database
  - Let a **LangChain agent** generate and run SQL queries
  - View results as chat messages
- 

## Line-by-Line Explanation

### Import required libraries

```
python
```

```
import streamlit as st
from pathlib import Path
from sqlalchemy import create_engine
import sqlite3
```

- `streamlit` : For UI building and chat interface.
- `Path` : To handle file paths (e.g., for SQLite DB file).
- `create_engine` : From SQLAlchemy — builds database engine.
- `sqlite3` : Low-level SQLite connector.

```
python
```

 Copy  Edit

```
from langchain.sql_database import SQLDatabase
from langchain.agents import create_sql_agent, AgentType
from langchain.agents.agent_toolkits import SQLDatabaseToolkit
from langchain_groq import ChatGroq
from langchain.callbacks import StreamlitCallbackHandler
```

- `SQLDatabase` : LangChain wrapper for database schema + connection.
- `create_sql_agent` : Builds an agent capable of translating natural language into SQL queries.
- `SQLDatabaseToolkit` : Helps the agent understand table structure and query tools.
- `ChatGroq` : LangChain-compatible wrapper for using Groq's LLMs (like LLaMA3).
- `StreamlitCallbackHandler` : Lets the LLM stream its thought process/output to the UI.

## Set up the Streamlit UI

```
python
```

 Copy  Edit

```
st.set_page_config(page_title="LangChain: Chat with SQL DB", page_icon="💻")
st.title("💻 LangChain: Chat with SQL Database")
```

- Sets browser tab title/icon and shows the app title on top.

## Database selection UI

python

 Copy  Edit

```
DB_LOCAL = "SQLite - student.db"
DB_MYSQL = "MySQL"
db_choice = st.sidebar.radio("Choose your database:", [DB_LOCAL, DB_MYSQL])
```

- Sidebar toggle: lets the user pick between local SQLite and remote MySQL DB.

---

## Groq API key input

python

 Copy  Edit

```
api_key = st.sidebar.text_input("Groq API Key", type="password")
```

- Field for entering the Groq API key (hidden input).

---

## MySQL connection info (only if chosen)

python

 Copy  Edit

```
if db_choice == DB_MYSQL:
    mysql_host = st.sidebar.text_input("MySQL Host")
    mysql_user = st.sidebar.text_input("MySQL User")
    mysql_password = st.sidebar.text_input("MySQL Password", type="password")
    mysql_db = st.sidebar.text_input("MySQL Database Name")
```

## Stop if no API key is given

```
python
```

 Copy  Edit

```
if not api_key:  
    st.info("Please enter your Groq API key to continue.")  
    st.stop()
```

- If the API key is missing, display a message and stop execution of the rest of the app.

## Initialize the LLM from Groq

```
python
```

 Copy  Edit

```
llm = ChatGroq(groq_api_key=api_key, model_name="Llama3-8b-8192", streaming=True)
```

- Uses the Groq LLaMA3 8B model (8192 token context).
- `streaming=True` : allows partial responses (streamed to UI).

## Configure and cache the database

python

 Copy  Edit

```
@st.cache_resource(ttl="2h")
def configure_database():
```

- This decorator caches the database object for 2 hours (avoids reloading on each interaction).

python

 Copy  Edit

```
if db_choice == DB_LOCAL:
    db_path = (Path(__file__).parent / "student.db").absolute()
    connect = lambda: sqlite3.connect(f"file:{db_path}?mode=ro", uri=True)
    engine = create_engine("sqlite://", creator=connect)
```

- For SQLite: builds an SQLAlchemy engine from the `student.db` file, opened read-only.

python

 Copy  Edit

```
else:
    if not all([mysql_host, mysql_user, mysql_password, mysql_db]):
        st.error("Please complete all MySQL connection fields.")
        st.stop()
    engine = create_engine(f"mysql+mysqlconnector://{{mysql_user}}:{{mysql_password}}@{{my:
```

- For MySQL: constructs connection string and engine from user inputs.

```
python
```

 Copy  Edit

```
    return SQLDatabase(engine)
```

- Returns a LangChain-wrapped `SQLDatabase` object for agent use.

## Connect to the DB

```
python
```

 Copy  Edit

```
db = configure_database()
```

- Calls the function to get the configured `SQLDatabase` object.

## Create agent toolkit + agent

```
python
```

 Copy  Edit

```
toolkit = SQLDatabaseToolkit(db=db, llm=llm)
```

- Tells the agent how to access the database via tools.

```
python
```

 Copy  Edit

```
agent = create_sql_agent(  
    llm=llm,  
    toolkit=toolkit,  
    agent_type=AgentType.ZERO_SHOT_REACT_DESCRIPTION,  
    verbose=True  
)
```

- Creates the main LangChain agent:
  - `ZERO_SHOT_REACT_DESCRIPTION` : lets the model plan & act using descriptions of tools.
  - `verbose=True` : prints reasoning/logs to console.

## 💬 Initialize/reset chat history

```
python
```

 Copy  Edit

```
if "messages" not in st.session_state or st.sidebar.button("Clear message history"):  
    st.session_state["messages"] = [{"role": "assistant", "content": "How can I help you ?"}]
```

- On first run (or when the clear button is clicked), a welcome assistant message is added to `st.session_state`.

## Display chat messages so far

python

 Copy  Edit

```
for msg in st.session_state["messages"]:
    st.chat_message(msg["role"]).write(msg["content"])
```

- Iterates through chat history and displays each message in the chat UI.
- 

## Get user input

python

 Copy  Edit

```
user_query = st.chat_input("Ask a question about your database...")
```

- Provides a text box at the bottom where user can type their query.
- 

## Handle input and generate a response

python

 Copy  Edit

```
if user_query:
    st.session_state["messages"].append({"role": "user", "content": user_query})
    st.chat_message("user").write(user_query)
```

- If user has submitted a query, it's saved to history and shown in the chat UI.

```
python
```

Copy Edit

```
with st.chat_message("assistant"):
    callback = StreamlitCallbackHandler(st.container())
    try:
        response = agent.run(user_query, callbacks=[callback])
    except Exception as e:
        response = f"Error: {e}"
    st.session_state["messages"].append({"role": "assistant", "content": response})
    st.write(response)
```

- This block:
  1. Creates a placeholder for the assistant reply.
  2. Streams agent output using the callback handler.
  3. Runs the agent to get SQL and results.
  4. Handles any errors cleanly.
  5. Saves + displays the assistant's reply.

## ✓ Summary: What This App Does

1. User picks a database (SQLite/MySQL) and gives Groq API key.
2. User types **natural language** like:  
 "Show me the names of all students with GPA > 3.5"
3. Agent uses LLaMA3 to:
  - Parse the intent
  - Convert it into an SQL query
  - Run it
  - Return the result to the user

Ollamma Based – SQL  
 import streamlit as st  
 from pathlib import Path

```

from sqlalchemy import create_engine
import sqlite3

from langchain.sql_database import SQLDatabase
from langchain.agents import create_sql_agent, AgentType
from langchain.agents.agent_toolkits import SQLDatabaseToolkit
from langchain_community.chat_models import ChatOllama
from langchain.callbacks import StreamlitCallbackHandler

# — UI setup —
st.set_page_config(page_title="Chat with SQL DB", page_icon="❑")
st.title("❑ Chat with SQL Database (Ollama)")

# — Select DB —
DB_LOCAL, DB_MYSQL = "SQLite", "MySQL"
db_choice = st.sidebar.radio("Database:", [DB_LOCAL, DB_MYSQL])

# — Model choice (optional) —
model_name = st.sidebar.text_input("Ollama Model", value="llama3") # You can change to
# "mistral", "codellama", etc.

# — MySQL credentials if selected —
if db_choice == DB_MYSQL:
    mysql_host = st.sidebar.text_input("Host")
    mysql_user = st.sidebar.text_input("User")
    mysql_password = st.sidebar.text_input("Password", type="password")
    mysql_db = st.sidebar.text_input("Database")

# — Initialize LLM —
llm = ChatOllama(model=model_name)

# — Configure DB (cached) —
@st.cache_resource(ttl="2h")
def get_db():
    if db_choice == DB_LOCAL:
        path = (Path(__file__).parent / "student.db").absolute()
        conn = lambda: sqlite3.connect(f"file:{path}?mode=ro", uri=True)
        engine = create_engine("sqlite:///", creator=conn)
    else:
        if not all([mysql_host, mysql_user, mysql_password, mysql_db]):
            st.error("Complete all MySQL fields.")
            return None
        engine = create_engine(
            f"mysql+mysqlconnector://{mysql_user}:{mysql_password}@{mysql_host}/{mysql_db}"
        )
    return SQLDatabase(engine)

```

```

# — Load DB and run app if valid —
db = get_db()
if db:
    # — Build agent —
    toolkit = SQLDatabaseToolkit(db=db, llm=llm)
    agent = create_sql_agent(
        llm=llm,
        toolkit=toolkit,
        agent_type=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
        verbose=True,
    )

# — Chat history —
if "messages" not in st.session_state or st.sidebar.button("Clear history"):
    st.session_state.messages = [{"role": "assistant", "content": "How can I help?"}]

for m in st.session_state.messages:
    st.chat_message(m["role"]).write(m["content"])

# — User input —
if user_q := st.chat_input("Ask the database..."):
    st.session_state.messages.append({"role": "user", "content": user_q})
    st.chat_message("user").write(user_q)

with st.chat_message("assistant"):
    cb = StreamlitCallbackHandler(st.container())
    try:
        resp = agent.run(user_q, callbacks=[cb])
    except Exception as e:
        resp = f"Error: {e}"
    st.session_state.messages.append({"role": "assistant", "content": resp})
    st.write(resp)

```

## TEXT-SUMMARISATION

## Conceptual Overview (No Code Yet)

### Goal:

We want to:

1. Take a **speech** (text or PDF).
2. Use an **AI model** (like Gemma-7b from Groq/Ollama) to:
  - Summarize it.
  - Optionally, translate the summary.
  - Format it beautifully (with a title, numbered points).
3. Make sure it's accurate, even for large speeches.

## Summary of Everything

|  Step |  What It Does |
|--|--|
| Load API   | Gets your secure key for LLM.  |
| Load Model   | Connects to Groq's Gemma-7b.   |
| Prompt   | Creates customizable instructions.   |
| Chat Summary   | Single-shot summary using chat-style interaction.  |
| Translate Summary  | Summarizes & translates speech to any language.  |
| PDF Loader   | Reads speeches from PDF.   |
| Text Splitter  | Splits long texts into manageable chunks.  |
| Map-Reduce Chain   | Summarizes each chunk → merges results.  |
| Refine Chain   | Iteratively refines one summary with the next.   |

## Libraries & Their Roles

| 📦 Library                                   | 🚀 Purpose  |
|---|--|
| <code>dotenv</code>                         | Load API keys securely from <code>.env</code> file.          |
| <code>langchain_groq</code>                 | Connect to Groq's Gemma-7b model.                            |
| <code>langchain.schema</code>               | Create structured chat messages (human/system/AI).           |
| <code>PromptTemplate</code>                 | Reusable prompts with variables (like speech + language).    |
| <code>LLMChain</code>                       | Chain a model + prompt together for single-step tasks.       |
| <code>PyPDFLoader</code>                    | Read and extract text from PDF files.                        |
| <code>RecursiveCharacterTextSplitter</code> | Split long documents into manageable chunks.                 |
| <code>load_summarize_chain</code>           | Load built-in summarization logic (stuff/map-reduce/refine). |

## Step-by-Step Logic

### 1. Load the AI Model (LLM)

- We load a **language model** (LLM) from Groq (e.g., Gemma-7b).
- It works like a very smart assistant that can read and write.

### 2. Define What We Want (Prompt)

- We give it a **prompt**: a set of instructions like "Please summarize this speech".
- Prompts tell the model how to behave.

### 3. Prepare the Speech

- The speech can come from:
  - A long **text block**.
  - A **PDF document** (using a loader).
- If it's long, we **split** it into small pieces (called "chunks") so the model can handle it.

### 4. Summarize in Different Styles

We use 3 types of summarization **chains**:

1. **Stuff** – Put all chunks together and summarize.
2. **Map-Reduce** – Summarize each chunk → then combine summaries into one.
3. **Refine** – Summarize first chunk, refine it with the next, and so on.

### 5. Translate (Optional)

- After summary, we can ask the model to translate it into another language like Hindi or French.



```

# 1. Load required libraries
import os
from dotenv import load_dotenv
from langchain_groq import ChatGroq
from langchain.schema import SystemMessage, HumanMessage
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.chains.summarize import load_summarize_chain

# 2. Load API key from .env file
load_dotenv()
api_key = os.getenv("GROQ_API_KEY")

# 3. Load LLM (Gemma model from Groq)
llm = ChatGroq(groq_api_key=api_key, model_name="Gemma-7b-It")

# 4. One-shot summary via chat message
speech_text = """<Your full speech here>""" # Replace this with actual speech text
chat_messages = [
    SystemMessage(content="You are an expert speech summarizer."),
    HumanMessage(content=f"Please summarize the following speech:\n{speech_text}")
]
response = llm(chat_messages)
print("□ One-shot Summary:\n", response.content)

# 5. Translate + summarize with template
template = """
Write a summary of the following speech:
Speech: {speech}
Translate the precise summary to {language}
"""

prompt = PromptTemplate(input_variables=["speech", "language"], template=template)
llm_chain = LLMChain(llm=llm, prompt=prompt)
translated_summary = llm_chain.run({"speech": speech_text, "language": "Hindi"})
print("□ Translated Summary:\n", translated_summary)

# 6. Load a PDF and split it into chunks
loader = PyPDFLoader("apjspeech.pdf")
docs = loader.load()
text_splitter = RecursiveCharacterTextSplitter(chunk_size=2000, chunk_overlap=100)
final_docs = text_splitter.split_documents(docs)

# 7. MAP-REDUCE chain: Summarize big documents intelligently
map_prompt = PromptTemplate(
    input_variables=["text"],

```

```

template="Please summarize the below speech:\nSpeech: `{{text}}`\nSummary:"
)
combine_prompt = PromptTemplate(
    input_variables=["text"],
    template=""
)
Provide the final summary of the entire speech with these important points.
Add a motivational title. Start with an introduction. Then provide the summary as bullet points.
Speech: {{text}}
"
)

summary_chain = load_summarize_chain(
    llm=llm,
    chain_type="map_reduce",
    map_prompt=map_prompt,
    combine_prompt=combine_prompt,
    verbose=True
)
mapreduce_summary = summary_chain.run(final_docs)
print("□ Map-Reduce Summary:\n", mapreduce_summary)

# 8. REFINE chain: Improve the summary step-by-step
refine_chain = load_summarize_chain(llm=llm, chain_type="refine", verbose=True)
refined_summary = refine_chain.run(final_docs)
print("□ Refined Summary:\n", refined_summary)

```

## Explanation

### Map-Reduce Summary:

- **Map:** We first summarize each chunk of text independently (from the split document).
- **Reduce:** After summarizing each chunk, we combine them into one cohesive summary.
- We use the `map_prompt` to guide the summarization of individual chunks and the `combine_prompt` to merge them into one summary.

### Refinement Summary:

- **Refinement:** Summarization happens iteratively. For each chunk, it generates a summary and then refines it as new chunks are added.
- This process makes the summary more polished by iterating over the text chunks and improving the output step by step.

## Map\_reduce using t5

```

# Import required libraries
from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.chains.summarize import load_summarize_chain
from langchain.prompts import PromptTemplate
from langchain.llms import T5
from langchain.chains import LLMChain

# 1. Load PDF document locally
loader = PyPDFLoader("apjspeech.pdf") # Replace with your document
docs = loader.load()

# 2. Split document using RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(chunk_size=2000, chunk_overlap=100)
final_docs = text_splitter.split_documents(docs)

# 3. Set up the Map-Reduce chain using T5 model
t5_model = T5.from_pretrained("t5-base") # You can use "t5-large" or any other pre-trained
model

# Create the map prompt (what the LLM should do for each chunk)
map_prompt = PromptTemplate(
    input_variables=["text"],
    template="Please summarize the below speech:\nSpeech: `{{text}}`\nSummary:"
)
# Create the combine prompt (what to do after summarizing each chunk)
combine_prompt = PromptTemplate(
    input_variables=["text"],
    template=""
)
Provide the final summary of the entire speech with these important points.
Add a motivational title. Start with an introduction. Then provide the summary as bullet points.
Speech: {{text}}
"
)

# Set up the map-reduce summarization chain
summary_chain = load_summarize_chain(
    llm=t5_model,
    chain_type="map_reduce",
    map_prompt=map_prompt,
    combine_prompt=combine_prompt,
    verbose=True
)
# Run the chain to get the map-reduced summary

```

```
mapreduce_summary = summary_chain.run(final_docs)
print("□ Map-Reduce Summary:\n", mapreduce_summary)
```

## REFINED - t5

```
# Import required libraries
from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.chains.summarize import load_summarize_chain
from langchain.prompts import PromptTemplate
from langchain.llms import T5

# 1. Load PDF document locally
loader = PyPDFLoader("apjspeech.pdf") # Replace with your document
docs = loader.load()

# 2. Split document using RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(chunk_size=2000, chunk_overlap=100)
final_docs = text_splitter.split_documents(docs)

# 3. Set up the Refine chain using T5 model
t5_model = T5.from_pretrained("t5-base") # You can use "t5-large" or any other pre-trained
model

# Create the prompt for refining summaries iteratively
refine_prompt = PromptTemplate(
    input_variables=["text"],
    template="Please summarize and refine the following speech:\nSpeech: `{{text}}`\nSummary:"
)

# Set up the refine summarization chain
refine_chain = load_summarize_chain(
    llm=t5_model,
    chain_type="refine",
    verbose=True
)

# Run the chain to get the refined summary
refined_summary = refine_chain.run(final_docs)
print("□ Refined Summary:\n", refined_summary)
```

## Streamlit summary

```
import validators, streamlit as st
```

```

from langchain.prompts import PromptTemplate
from langchain_groq import ChatGroq
from langchain.chains.summarize import load_summarize_chain
from langchain_community.document_loaders import YoutubeLoader, UnstructuredURLLoader

## Streamlit APP
st.set_page_config(page_title="LangChain: Summarize Text From YT or Website",
page_icon="❑")
st.title("❑ LangChain: Summarize Text From YT or Website")
st.subheader("Summarize URL")

## Get the Groq API Key and URL to summarize
with st.sidebar:
    groq_api_key = st.text_input("Groq API Key", value="", type="password")

generic_url = st.text_input("URL", label_visibility="collapsed")

## Setup Groq Chat model
llm = ChatGroq(model="gemma-7b-it", groq_api_key=groq_api_key)

prompt_template = """
Provide a summary of the following content in 300 words:
Content: {text}
"""

prompt = PromptTemplate(template=prompt_template, input_variables=["text"])

if st.button("Summarize the Content from YT or Website"):
    if not groq_api_key.strip() or not generic_url.strip():
        st.error("Please provide the information to get started")
    elif not validators.url(generic_url):
        st.error("Please enter a valid URL (YouTube or Website)")
    else:
        try:
            with st.spinner("Waiting..."):
                if "youtube.com" in generic_url:
                    loader = YoutubeLoader.from_youtube_url(generic_url, add_video_info=True)
                else:
                    loader = UnstructuredURLLoader(
                        urls=[generic_url],
                        ssl_verify=False,
                        headers={"User-Agent": "Mozilla/5.0"})
                docs = loader.load()
                chain = load_summarize_chain(llm, chain_type="stuff", prompt=prompt)
                output_summary = chain.run(docs)
                st.success(output_summary)
        except Exception as e:
            st.error(f"An error occurred: {e}")

```

```
st.error(f"An error occurred: {e}")
```

## MATH SOLVER

### Phase 1: Explain Like I'm 5 (ELI5 → PhD)

#### What the app does:

This app is like a **super smart assistant** that can:

-  Solve math problems (like a calculator that talks).
  -  Search the web (via Wikipedia).
  -  Think through tricky reasoning questions (like Sherlock Holmes with math).
- 

### Step-by-Step Logic – No Code Yet

1. You open the app. It asks for a special secret key (Groq API key) to talk to a smart AI brain (Gemma model).
2. You type your math or logic question like:  
*"If I eat 2 apples and buy 12 bananas, how many fruits do I have?"*
3. The AI has **three tools** to help it:
  -  **Calculator**: Solves math.
  -  **Wikipedia**: Looks up facts.
  -  **Reasoning tool**: Explains logic step by step.
4. The smart assistant reads your question and thinks:
  - "Hmm... do I need to calculate something? Or look it up? Or explain reasoning?"
  - It then **chooses the right tool** (or a mix) to answer.
5. Finally, it replies in a **chat-style format**, and the answer shows up on the screen.

---

## Phase 2: Deep Dive into the Whole Process

### 1. Streamlit UI Setup

-  Streamlit makes the app interactive and web-based.
- It shows text input boxes, buttons, and chat messages.

### 2. LLM (Language Model) Setup

- Using **Groq API**, the app connects to **Gemma-2 9B** model. This is a large language model (LLM) capable of understanding and generating text.

### 3. Tool Setup

- **Wikipedia Tool** → Answers general knowledge questions.
- **Math Chain (LLMMathChain)** → Solves math expressions step-by-step.
- **LLMChain with PromptTemplate** → A logical reasoning tool using a prompt like "Explain this question step by step."

### 4. Agent Setup

- Agent is like a smart brain that can choose from tools.
- We use `initialize_agent` with `ZERO_SHOT_REACT_DESCRIPTION`, meaning:
  - The agent reads your input.
  - It reacts and picks the correct tool based on the description.

### 5. Conversation Memory

- `st.session_state["messages"]` stores chat history.
- The agent sees the whole conversation so it can remember what was asked before.

## 📦 Library & Function Summary

| Library   | Function                              | What It Does   | 🔗 |
|---|---------------------------------------|--|---|
| <code>streamlit</code>                          | <code>UI elements</code>              | Builds the web interface.                            |   |
| <code>langchain_groq</code>                     | <code>ChatGroq</code>                 | Connects to Groq LLM like Gemma 2.                   |   |
| <code>langchain.chains</code>                   | <code>LLMMathChain , LLMChain</code>  | Prebuilt logic for math solving and prompt chaining. |   |
| <code>langchain.prompts</code>                  | <code>PromptTemplate</code>           | Makes structured instructions for the LLM.           |   |
| <code>langchain_community.utiliti<br/>es</code> | <code>WikipediaAPIWrapper</code>      | Lets the bot query Wikipedia.                        |   |
| <code>langchain.agents</code>                   | <code>Tool , initialize_agent</code>  | Smart agent that can choose the right tool to use.   |   |
| <code>langchain.callbacks</code>                | <code>StreamlitCallbackHandler</code> | Show live feedback while the agent works.            |   |

## ✓ Summary

- The app connects to Gemma-2 via Groq to answer questions.
- It intelligently chooses between math solving, reasoning, and web search.
- The UI is clean, interactive, and conversational.
- The code is now modular, commented, and professional.

```
import streamlit as st
from langchain_groq import ChatGroq
from langchain.chains import LLMMathChain, LLMChain
from langchain.prompts import PromptTemplate
from langchain_community.utilities import WikipediaAPIWrapper
from langchain.agents import Tool, initialize_agent
from langchain.agents.agent_types import AgentType
from langchain.callbacks import StreamlitCallbackHandler

# ----- Streamlit App Setup -----
st.set_page_config(page_title="Text to Math Solver and Assistant", page_icon="□")
st.title("□ Text to Math Problem Solver using Gemma 2")
groq_api_key = st.sidebar.text_input("□ Enter your Groq API Key", type="password")
```

```

# Block the app unless API key is provided
if not groq_api_key:
    st.info("Please enter your Groq API Key to continue.")
    st.stop()

# ----- Initialize LLM -----
llm = ChatGroq(model="gemma2-9b-it", groq_api_key=groq_api_key)

# ----- Tools Setup -----

# 1. Wikipedia Tool for general knowledge
wikipedia_tool = Tool(
    name="Wikipedia",
    func=WikipediaAPIWrapper().run,
    description="Search Wikipedia for topic-based information."
)

# 2. Calculator Tool for solving math problems
math_chain = LLMMathChain.from_llm(llm=llm)
calculator_tool = Tool(
    name="Calculator",
    func=math_chain.run,
    description="Use for solving math expressions or word problems involving math."
)

# 3. Reasoning Tool using a custom prompt
reasoning_prompt = PromptTemplate(
    input_variables=["question"],
    template="""
You're a helpful math and logic assistant. Break down the user's question into clear logical steps
and explain the answer clearly:
Question: {question}
Answer:
"""
)
reasoning_chain = LLMChain(llm=llm, prompt=reasoning_prompt)
reasoning_tool = Tool(
    name="Reasoning Tool",
    func=reasoning_chain.run,
    description="Handles reasoning and logic-based questions."
)

# ----- Agent Setup -----
agent = initialize_agent(
    tools=[wikipedia_tool, calculator_tool, reasoning_tool],
    llm=llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
)

```

```

verbose=False,
handle_parsing_errors=True
)

# ----- Chat Session Setup -----
# Create chat history
if "messages" not in st.session_state:
    st.session_state["messages"] = [
        {"role": "assistant", "content": "Hi! I can solve math, reasoning, and lookup questions. Ask me anything!"}
    ]

# Display chat history
for msg in st.session_state["messages"]:
    st.chat_message(msg["role"]).write(msg["content"])

# ----- User Input Area -----
question = st.text_area("❑ Enter your question:",
    "I have 5 bananas and 7 grapes. I eat 2 bananas and give away 3 grapes. Then I buy a dozen apples and 2 packs of blueberries. Each pack has 25 berries. How many fruits do I have at the end?")
)

# ----- Trigger Response -----
if st.button("❑ Find My Answer"):
    if question:
        with st.spinner("Thinking..."):
            # Store user message
            st.session_state["messages"].append({"role": "user", "content": question})
            st.chat_message("user").write(question)

            # Streamlit callback handler to show intermediate thoughts
            streamlit_cb = StreamlitCallbackHandler(st.container(), expand_new_thoughts=False)

    # Run agent and get response
    try:
        response = agent.run(question, callbacks=[streamlit_cb])
    except Exception as e:
        response = f"Sorry, an error occurred: {e}"

        # Store and show assistant response
        st.session_state["messages"].append({"role": "assistant", "content": response})
        st.chat_message("assistant").write(response)
    else:
        st.warning("Please enter a question to get started.")

```

## LINK BASED SUMMARISER



## ELI5: What is this app doing?

This app is like a **magical summarizer**. You give it a **YouTube link** or a **webpage URL**, and it reads it, understands it, and tells you what it's about in **300 words** — like a mini book report!



## Step-by-Step (Simple Version First)

1. **You open the app.**

It asks for:

- Hugging Face API Key (so it can use a smart AI brain from Hugging Face)
- A link (either YouTube video or webpage)

2. **You paste a link** (like a YouTube video of a lecture or a news article).

3. **You press the summarize button.**

4. The app checks:

- Did you give it an API key?
- Did you paste a valid link?

5. It then **reads the content** of the YouTube video or webpage.

6. It asks the **Mistral 7B model** (a very smart AI hosted on Hugging Face) to write a short summary.

7. You get the summary on your screen!

## Technical Process (Full Explanation)

Let's go step-by-step through the entire logic behind the code:

---

### 1. Libraries and Tools Used

| Library                                 | Purpose   |
|---|---|
| <code>streamlit</code>                  | Creates the web interface (buttons, inputs, layout)     |
| <code>validators</code>                 | Checks if the link entered is a proper web URL          |
| <code>langchain.prompts</code>          | Helps define custom prompts for the AI model            |
| <code>langchain_groq.chatGroq</code>    | (Commented out here) Connects to Groq models like Gemma |
| <code>langchain.chains.summarize</code> | Loads a summarization chain using the LLM               |
| <code>YoutubeLoader</code>              | Extracts text data from a YouTube video                 |
| <code>UnstructuredURLLoader</code>      | Extracts text from a webpage                            |
| <code>HuggingFaceEndpoint</code>        | Connects to Hugging Face-hosted models like Mistral     |

## 2. User Interface Setup

- `st.set_page_config(...)` sets up the page title and emoji.
  - `st.sidebar.text_input(...)` asks user for Hugging Face API key securely.
  - `st.text_input(...)` asks for a link (YT or website).
- 

## 3. LLM Setup: Mistral 7B Model via Hugging Face

```
python
repo_id="mistralai/Mistral-7B-Instruct-v0.3"
llm=HuggingFaceEndpoint(repo_id=repo_id, ...)
```

 Copy

This sets up the Mistral model (a powerful text-understanding model) to be used for summarizing.

---

## 4. Prompt Setup: Tell AI What to Do

```
python
prompt_template = """
Provide a summary of the following content in 300 words:
Content:{text}
"""
```

 Copy

This gives a clear instruction to the model to summarize whatever content we provide.

## 5. Loading Content From URL

- If YouTube link:

```
python  
  
loader = YoutubeLoader.from_youtube_url(...)
```

 Copy  Edit

It pulls transcript and video info.

- Else:

```
python  
  
loader = UnstructuredURLLoader(...)
```

 Copy  Edit

It grabs the main content of the webpage (skipping ads and clutter).

Then:

```
python  
  
docs = loader.load()
```

 Copy  Edit

This gives you the content to summarize.

---

## 6. Summarization Chain

```
python  
  
chain = load_summarize_chain(llm, chain_type="stuff", prompt=prompt)
```

 Copy  Edit

This builds a pipeline that:

- Feeds the content to the LLM
  - Applies our prompt
  - Returns the summary
- 

## ✓ 7. Display Result

```
python  
  
output_summary = chain.run(docs)  
st.success(output_summary)
```

Finally, the app:

- Runs the chain
- Shows you the summary on screen!

## 🧠 Summary

| Feature          | Purpose                                      |
|------------------|--|
| Hugging Face API | Accesses a hosted Mistral model to summarize |
| Prompt Template  | Tells the model to create a 300-word summary |
| LangChain Tools  | Used to load data from YouTube or websites   |
| Streamlit        | Makes a friendly UI with inputs and outputs  |
| Validations      | Ensures the user gives real links and keys   |

import validators, streamlit as st

```

from langchain.prompts import PromptTemplate
from langchain_community.document_loaders import YoutubeLoader, UnstructuredURLLoader
from langchain.chains.summarize import load_summarize_chain
from langchain_huggingface import HuggingFaceEndpoint

# ---- Streamlit UI Setup ----
st.set_page_config(page_title="LangChain: Summarize Text From YT or Website",
page_icon="❑")
st.title("❑ LangChain: Summarize Text From YT or Website")
st.subheader("Summarize a YouTube video or website URL")

# ---- Sidebar Input for Hugging Face Token ----
with st.sidebar:
    hf_api_key = st.text_input("❑ Hugging Face API Token", value="", type="password")

# ---- Input URL ----
generic_url = st.text_input("Paste the URL here", label_visibility="collapsed")

# ---- Define Hugging Face LLM (Mistral) ----
repo_id = "mistralai/Mistral-7B-Instruct-v0.3"
llm = HuggingFaceEndpoint(
    repo_id=repo_id,
    max_length=150,
    temperature=0.7,
    token=hf_api_key
)

# ---- Prompt Template for Summary ----
prompt_template = """
Provide a summary of the following content in 300 words:
Content: {text}
"""

prompt = PromptTemplate(template=prompt_template, input_variables=["text"])

# ---- When Button Clicked ----
if st.button("Summarize the Content from YT or Website"):
    if not hf_api_key.strip() or not generic_url.strip():
        st.error("! Please provide both Hugging Face API Token and a URL.")
    elif not validators.url(generic_url):
        st.error("! The URL is invalid. Please enter a valid YouTube or website URL.")
    else:
        try:
            with st.spinner("❑ Summarizing... please wait"):
                # Choose loader based on URL type
                if "youtube.com" in generic_url:
                    loader = YoutubeLoader.from_youtube_url(generic_url, add_video_info=True)

```

```
else:  
    loader = UnstructuredURLLoader(  
        urls=[generic_url],  
        ssl_verify=False,  
        headers={"User-Agent": "Mozilla/5.0"}  
    )  
    docs = loader.load()  
  
    # Create summary chain  
    chain = load_summarize_chain(llm, chain_type="stuff", prompt=prompt)  
    output_summary = chain.run(docs)  
  
    # Display result  
    st.success(output_summary)  
  
except Exception as e:  
    st.exception(f"✖ An error occurred: {e}")
```

## HYBRID SEARCH

## Simple Story (For a 5th Grader)

Imagine you have a diary with memories of your past trips — like "I went to Paris in 2023" and "I went to New York in 2022." Now, a friend asks:

| "What city did you visit first?"

You want a smart way to search through your diary to find the right answer. But instead of flipping pages, you use a robot assistant  who's really good at understanding meanings (not just words) and who remembers how words are related to each other. This robot uses two brains:

1. A **math brain**  (dense vector) to understand the meanings.
2. A **keyword brain**  (BM25 - sparse vector) to focus on important words like "visit" or "2021."

You put your memories into a magic library  (Pinecone). Then you ask it questions like "What city did I visit first?" and it uses both brains to find the best match.

---

## PhD-Level Understanding

This project builds a **hybrid search retriever** using:

- **Dense semantic search** with HuggingFace sentence embeddings.
- **Sparse keyword-based search** using BM25 (TF-IDF-like).
- **Pinecone vector database** to store both kinds of data.
- **LangChain's Retriever** to combine these signals for improved question answering.

## Step-by-Step Process (Conceptual)

| Step | Action   | Purpose  |
|------|--|--|
| 1    | Install Pinecone and related packages                | So you can use Pinecone's vector DB and text encoding tools        |
| 2    | Initialize Pinecone client                           | To connect to your Pinecone environment                            |
| 3    | Create an index                                      | Like creating a new library where your data will go                |
| 4    | Load a <b>dense embedding model</b> from HuggingFace | So you can convert sentences into meaning-based number vectors     |
| 5    | Load and fit a <b>BM25 sparse encoder</b>            | To get keyword-based importance weights                            |
| 6    | Convert and store both vectors                       | Prepares your data to be inserted into Pinecone                    |
| 7    | Add your memory-like texts to the index              | Stores both dense and sparse vectors                               |
| 8    | Ask a question to the retriever                      | The hybrid retriever uses both vector types to find the best match |

## What Each Library Does

| Library                                     | Function   |
|---|--|
| <code>pinecone</code>                       | Hosts the vector database to store and search embeddings       |
| <code>pinecone-text</code>                  | Provides the BM25Encoder to generate sparse keyword vectors    |
| <code>langchain_community.retrievers</code> | Lets you combine sparse and dense vectors in one retriever     |
| <code>langchain_huggingface</code>          | Pulls a HuggingFace model and converts text into dense vectors |
| <code>dotenv</code> (optional)              | Loads environment variables like API keys                      |

## Summary

This project builds a **hybrid search engine** that:

- Understands **word meanings** using HuggingFace embeddings.
- Understands **important keywords** using BM25 (like TF-IDF).
- Combines both to **accurately find answers** in your data.
- Uses Pinecone as a high-speed memory that can be queried with LangChain tools.

```
# 📦 Install required Pinecone packages
!pip install --upgrade --quiet pinecone-client pinecone-text pinecone-notebooks

# 🔑 Pinecone API Key
api_key = "your-pinecone-api-key"

# ----- STEP 1: Set up Pinecone index -----
from pinecone import Pinecone, ServerlessSpec

index_name = "hybrid-search-langchain-pinecone"
pc = Pinecone(api_key=api_key)

# Create index only if it doesn't already exist
if index_name not in pc.list_indexes().names():
```

```

pc.create_index(
    name=index_name,
    dimension=384, # matches size of MiniLM embeddings
    metric="dotproduct",
    spec=ServerlessSpec(cloud="aws", region="us-east-1"),
)
index = pc.Index(index_name) # Connect to the index

# ----- STEP 2: Set up vector encoders -----
from langchain_huggingface import HuggingFaceEmbeddings
from pinecone_text.sparse import BM25Encoder

# Load dense vector model (MiniLM)
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

# Load and fit BM25 sparse encoder
bm25_encoder = BM25Encoder().default()
sentences = [
    "In 2023, I visited Paris",
    "In 2022, I visited New York",
    "In 2021, I visited New Orleans",
]
bm25_encoder.fit(sentences)

# Optionally save and reload BM25 values
bm25_encoder.dump("bm25_values.json")
bm25_encoder = BM25Encoder().load("bm25_values.json")

# ----- STEP 3: Store in Pinecone using Hybrid Retriever -----
from langchain_community.retrievers import PineconeHybridSearchRetriever

# Create retriever that uses both dense & sparse encoders
retriever = PineconeHybridSearchRetriever(
    embeddings=embeddings,
    sparse_encoder=bm25_encoder,
    index=index
)

# Add sentences to the hybrid retriever
retriever.add_texts(sentences)

# ----- STEP 4: Ask a Question -----
# Hybrid search based on semantic and keyword overlap
result = retriever.invoke("What city did I visit first?")
print("Q Answer:", result)

```

**CODE LAMMA**

## 5th Grader-Friendly Explanation

Imagine you have a super smart robot friend named **CodeGuru** 🧠 living in your computer. You want to ask it questions like:

“How do I write a Python program to add two numbers?”

You write your question in a notebook (Gradio app), and every time you ask something new, your robot remembers all your past questions — like a diary — so it answers better.

You talk to your robot using a tunnel (API call). This tunnel sends your question to CodeGuru, who replies back with an answer.

The Gradio app is like a fancy notebook where you write your questions and read the robot's answers.

---

## Expert/PhD-Level Explanation

This is a **local LLM (Language Model) interface** using:

- A REST API endpoint (likely from [Ollama](#) or similar at `localhost:11434`)
- A custom model named `"codeguru"` which is queried using JSON
- `Gradio` to create a browser UI for chatting with this model
- A **persistent chat history** so context builds up like a real conversation

## Step-by-Step Breakdown

| Step | Action  | Purpose                               |
|------|---|---------------------------------------|
| 1    | Import libraries  | Needed to make API calls and build UI |
| 2    | Define API info ( <code>url</code> , <code>headers</code> ) | Where and how to talk to the LLM      |
| 3    | Store chat history  | So future messages have context       |
| 4    | Build a function to call the API                            | Send prompt, receive response         |
| 5    | Build a UI with Gradio                                      | Lets users interact in a browser      |
| 6    | Launch the app  | Opens a local server for interaction  |

## What Each Library Does

| Library               | Purpose  |
|-----------------------|--|
| <code>requests</code> | Sends HTTP requests (your prompt) to a local AI API          |
| <code>json</code>     | Converts Python data into JSON for API communication         |
| <code>gradio</code>   | Creates a web UI so users can interact easily with the model |

## Summary

- You built a **chat UI using Gradio** to talk to a **local LLM** via REST API.
- The model ("codeguru") **remembers past inputs** to build smarter answers.
- It uses `requests` and `json` to send and receive data from the LLM.
- Gradio wraps it into a clean web-based chatbot interface.

Would you like to upgrade this to include **streaming output** or **reset history support**?

```
import requests
import json
import gradio as gr

# Define the API endpoint (usually local Ollama or similar LLM server)
url = "http://localhost:11434/api/generate"

# Set headers to send JSON data
headers = {
    'Content-Type': 'application/json'
}

# Store previous prompts to maintain conversation history
history = []
```

```

def generate_response(prompt):
    """
    Sends the user's prompt along with the full chat history
    to the local LLM model and returns its response.
    """

    # Add the current user prompt to history
    history.append(prompt)

    # Combine entire conversation into one string
    final_prompt = "\n".join(history)

    # Prepare the payload to send to the LLM
    data = {
        "model": "codeguru", # model name running locally
        "prompt": final_prompt,
        "stream": False      # receive full response, not stream
    }

    # Make the POST request to the LLM API
    response = requests.post(url, headers=headers, data=json.dumps(data))

    # Check if response is OK
    if response.status_code == 200:
        # Parse the JSON and extract the actual model response
        data = json.loads(response.text)
        return data['response']
    else:
        # Show error if something went wrong
        return f"Error: {response.text}"

# Create a simple Gradio UI with a text input and output
interface = gr.Interface(
    fn=generate_response, # Function to call when user submits input
    inputs=gr.Textbox(lines=4, placeholder="Enter your Prompt here..."), # User input area
    outputs="text", # Show text output
    title="Local LLM Chat with CodeGuru",
    description="Ask your questions and get intelligent responses from a locally hosted model."
)

# Launch the interface
interface.launch()

```

**NVIDIA**

## What Is This App Doing?

You're building a chatbot using **NVIDIA's LLaMA-3 model** to answer questions from PDF documents (like reports, articles, or study material). The app reads the documents, understands their content using embeddings (like turning words into smart numbers), stores them in a searchable database (FAISS), and when a user asks a question, it finds relevant content and generates an answer using LLaMA-3.

---

## Step-by-Step Explanation (No Code Yet)

1. **Environment Setup:** Load the NVIDIA API key from a `.env` file.
2. **Document Loading:** Load PDF files from a directory (here, `./us_census`).
3. **Text Chunking:** Break big PDFs into smaller overlapping pieces so they can be searched effectively.
4. **Embeddings:** Convert each text chunk into a vector (numeric representation using NVIDIAEmbeddings).
5. **Store Vectors:** Save these vectors in a special database called FAISS so they can be searched quickly.
6. **User Input:** The user types a question.
7. **Search:** The app finds the most relevant document pieces for the question.
8. **Answer:** It gives that information to the LLaMA-3 model and generates a smart answer.
9. **Show Answer & Sources:** Display both the answer and the original document chunks used.

## Summary

- This app uses **NVIDIA LLM (LLaMA-3)** to answer questions about **PDF documents**.
- It turns documents into **vector form**, stores them in **FAISS**, and uses **retrieval-augmented generation (RAG)**.
- Streamlit powers the UI, while LangChain handles logic like loading, embedding, and prompting.

## What Each Library Does

| Library   | Role  |
|---|---|
| <code>streamlit</code>                            | Builds the web interface                                |
| <code>langchain_nvidia_ai_endpoints</code>        | Allows using NVIDIA's LLM & embeddings                  |
| <code>langchain_community.document_loaders</code> | Loads documents (PDFs, web pages)                       |
| <code>langchain.embeddings</code>                 | Handles vector creation                                 |
| <code>langchain.text_splitter</code>              | Splits text into chunks                                 |
| <code>langchain_core.prompts</code>               | Creates structured prompt templates                     |
| <code>langchain.chains</code>                     | Chains tasks like search + answer                       |
| <code>FAISS</code>                                | Stores and retrieves vector data fast                   |
| <code>dotenv</code>                               | Loads environment variables from <code>.env</code> file |

```
import streamlit as st
import os
import time
from dotenv import load_dotenv

# LangChain + NVIDIA
from langchain_nvidia_ai_endpoints import NVIDIAEmbeddings, ChatNVIDIA
from langchain_community.vectorstores import FAISS
from langchain_community.document_loaders import PyPDFDirectoryLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain.chains import create_retrieval_chain
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

# Load .env API keys
load_dotenv()
os.environ['NVIDIA_API_KEY'] = os.getenv("NVIDIA_API_KEY")

# Streamlit UI
st.title("❑ Nvidia NIM Demo with LLaMA-3")
user_question = st.text_input("Enter your question from the documents:")
```

```

# Function to embed and store PDF document chunks
def vector_embedding():
    if "vectors" not in st.session_state:
        st.session_state.embeddings = NVIDIAEmbeddings()
        st.session_state.loader = PyPDFDirectoryLoader("./us_census") # Load PDF files
        docs = st.session_state.loader.load()

    # Split large documents into chunks for easier searching
    splitter = RecursiveCharacterTextSplitter(chunk_size=700, chunk_overlap=50)
    split_docs = splitter.split_documents(docs[:30]) # Limit to 30 PDFs to avoid overload

    # Create FAISS vector DB from these chunks
    st.session_state.vectors = FAISS.from_documents(split_docs, st.session_state.embeddings)
    st.success("✓ Document embeddings created!")

# Button to trigger document processing
if st.button("🖨 Embed Documents"):
    vector_embedding()

# LLaMA-3 model from NVIDIA
llm = ChatNVIDIA(model="meta/llama3-70b-instruct")

# Prompt template for answering questions from context
prompt = ChatPromptTemplate.from_template("""
Answer the question based on the provided context only.
<context>
{context}
</context>
Question: {input}
""")

# Handle user question
if user_question:
    if "vectors" not in st.session_state:
        st.warning("Please embed documents first.")
    else:
        # Build the document-answering chain
        document_chain = create_stuff_documents_chain(llm, prompt)
        retriever = st.session_state.vectors.as_retriever()
        retrieval_chain = create_retrieval_chain(retriever, document_chain)

        # Get answer
        start = time.process_time()
        response = retrieval_chain.invoke({'input': user_question})
        duration = time.process_time() - start

# Display result

```

```
st.subheader("✍ Answer")
st.write(response['answer'])
st.caption(f"⌚ Response Time: {duration:.2f} seconds")

# Show document sources
with st.expander("🔍 Relevant Documents"):
    for doc in response["context"]:
        st.write(doc.page_content)
        st.write("---")
```

## CONVERSATIONAL QA



You're building a **Conversational Retrieval-Augmented Generation (RAG)** chatbot using:

- **LangChain** as the orchestration framework.
  - **Groq's LLaMA-3 8B** model for answering questions.
  - **Chroma** for vector storage and document retrieval.
  - **A real blog post** (on agents) as the document source.
  - **Message history awareness** to maintain context in multi-turn conversations.
- 

## Step-by-Step Concept (5th Grade to PhD)

### 1. Loading Environment and LLM

You're fetching your secrets like API keys from a `.env` file. Then you're loading the **Groq LLaMA-3 8B model** using LangChain.

### 2. Document Ingestion

You're using `WebBaseLoader` to fetch content from a blog post and extract only the readable content (no sidebars/ads) using `bs4`.

### 3. Prompt Engineering

You define:

- A **system prompt** that says: "You are an assistant. Use only the context below."
- A **prompt template** that combines this system message with a user question.

## 4. RAG Setup

You split the document into smaller chunks and embed them using HuggingFace (`Minilm-L6-v2`). Then you create a **retriever** from these embeddings using Chroma. A **retrieval chain** is built to:

- Fetch relevant chunks.
- Feed them into the model along with the prompt.
- Return a concise answer.

## 5. Contextualization (History-Aware RAG)

You go next-level by making the chatbot **remember past messages**. It does this by:

- Reformulating follow-up questions (like "Tell me more") into full ones using a custom prompt.
- Storing Q&A in `chat_history`.

## 6. Multi-Turn Chat with Session Memory

You use `RunnableWithMessageHistory` to persist chat history across a session. This enables coherent conversations.

## ■ Full Process Summary

| Step                     | What Happens   | Why It Matters                      |
|--------------------------|--|-------------------------------------|
| 💡 Load <code>.env</code> | Access <code>GROQ_API_KEY</code> , <code>HF_TOKEN</code>               | Keeps your secrets safe             |
| 🌐 Load Model             | Use <code>chatGroq</code> with LLaMA3-8B                               | Large model for intelligent answers |
| 🌐 Load Web Data          | Scrape blog content with <code>webBaseLoader</code> + <code>bs4</code> | Use real-world data                 |
| ✂️ Chunking              | Split text into 700-character chunks                                   | Helps LLM handle long docs          |
| 🔍 Embedding              | Convert chunks into vectors using <code>HuggingFaceEmbeddings</code>   | Enable semantic search              |
| 🧠 RAG                    | Combine retriever with LLM   | Accurate, grounded answers          |
| 💡 History-Aware RAG      | Reformulate follow-up questions  | Supports natural dialogue           |
| ⌚ Session Store          | Track conversation using session ID                                    | Makes bot memory persistent         |

## Library Summary

| Library                                     | Purpose                                 |
|---|---|
| <code>dotenv</code>                         | Load API keys and environment variables |
| <code>langchain_groq</code>                 | Use LLaMA models via Groq API           |
| <code>langchain_chroma</code>               | Store and search vector data            |
| <code>langchain_huggingface</code>          | Get text embeddings                     |
| <code>bs4</code>                            | Parse HTML from blogs                   |
| <code>WebBaseLoader</code>                  | Scrape blog content                     |
| <code>RecursiveCharacterTextSplitter</code> | Chunk long text                         |
| <code>ChatPromptTemplate</code>             | Create structured LLM prompts           |
| <code>create_retrieval_chain</code>         | Chain retriever + LLM                   |
| <code>RunnableWithMessageHistory</code>     | Track multi-turn chats                  |
| <code>ChatMessageHistory</code>             | Store chat logs in memory               |

```
# 1. Import required packages
import os
from dotenv import load_dotenv
import bs4
from langchain_groq import ChatGroq
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_chroma import Chroma
from langchain_community.document_loaders import WebBaseLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_chains.combine_documents import create_stuff_documents_chain
from langchain_chains import create_retrieval_chain, create_history_aware_retriever
from langchain_core.messages import AIMessage, HumanMessage
from langchain_community.chat_message_histories import ChatMessageHistory
from langchain_core.chat_history import BaseChatMessageHistory
```

```

from langchain_core.runnables.history import RunnableWithMessageHistory

# 2. Load environment
load_dotenv()
os.environ["HF_TOKEN"] = os.getenv("HF_TOKEN")
groq_api_key = os.getenv("GROQ_API_KEY")

# 3. Initialize Groq LLM
llm = ChatGroq(groq_api_key=groq_api_key, model_name="Llama3-8b-8192")

# 4. Load blog content using WebBaseLoader and BeautifulSoup
loader = WebBaseLoader(
    web_paths=("https://lilianweng.github.io/posts/2023-06-23-agent/"),
    bs_kw_args={"parse_only": bs4.SoupStrainer(class_=("post-content", "post-title", "post-header"))},
)
docs = loader.load()

# 5. Chunk and embed content
splitter = RecursiveCharacterTextSplitter(chunk_size=700, chunk_overlap=50)
chunks = splitter.split_documents(docs)
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
retriever = Chroma.from_documents(chunks, embeddings).as_retriever()

# 6. Define prompt
system_prompt = (
    "You are an assistant for question-answering tasks.\n"
    "Use the following pieces of retrieved context to answer the question.\n"
    "If unknown, say 'I don't know'. Use 3 sentences max.\n"
    "\n\n{context}"
)
qa_prompt = ChatPromptTemplate.from_messages([("system", system_prompt), ("human", "{input}")])
qa_chain = create_stuff_documents_chain(llm, qa_prompt)

# 7. Setup basic RAG chain
rag_chain = create_retrieval_chain(retriever, qa_chain)

# 8. Add history-aware retriever
contextualize_prompt = ChatPromptTemplate.from_messages([
    ("system", "Reformulate follow-up question to be standalone."),
    MessagesPlaceholder("chat_history"),
    ("human", "{input}")
])
history_aware_retriever = create_history_aware_retriever(llm, retriever, contextualize_prompt)

# 9. Create full conversational RAG chain

```

```

chat_prompt = ChatPromptTemplate.from_messages([
    ("system", system_prompt),
    MessagesPlaceholder("chat_history"),
    ("human", "{input}")
])
qa_chain_history = create_stuff_documents_chain(llm, chat_prompt)
rag_chain_with_history = create_retrieval_chain(history_aware_retriever, qa_chain_history)

# 10. Simulate chat history (manual)
chat_history = []
question = "What is Self-Reflection?"
response1 = rag_chain_with_history.invoke({"input": question, "chat_history": chat_history})
chat_history.extend([HumanMessage(content=question),
                     AIMessage(content=response1["answer"])]

question2 = "Tell me more about it?"
response2 = rag_chain_with_history.invoke({"input": question2, "chat_history": chat_history})
print(response2["answer"])

# 11. Store chat in sessions using session_id
store = {}
def get_session_history(session_id: str) -> BaseChatMessageHistory:
    if session_id not in store:
        store[session_id] = ChatMessageHistory()
    return store[session_id]

conversational_chain = RunnableWithMessageHistory(
    rag_chain_with_history,
    get_session_history,
    input_messages_key="input",
    history_messages_key="chat_history",
    output_messages_key="answer",
)
# 12. Make queries with memory
print(conversational_chain.invoke(
    {"input": "What is Task Decomposition?"},
    config={"configurable": {"session_id": "abc123"}}
)["answer"])

print(conversational_chain.invoke(
    {"input": "What are common ways of doing it?"},
    config={"configurable": {"session_id": "abc123"}}
)["answer"])

```



You're building a **Conversational Retrieval-Augmented Generation (RAG)** chatbot using:

- **LangChain** as the orchestration framework.
- **Groq's LLaMA-3 8B** model for answering questions.
- **Chroma** for vector storage and document retrieval.
- **A real blog post** (on agents) as the document source.
- **Message history awareness** to maintain context in multi-turn conversations.

## Crew AI

### Tool

**In the context of AI Agents (like CrewAI or LangChain):**

A **tool** is:

- A **function** or **module** designed to **do one thing really well**.
- Used by an **AI agent** when it needs to solve a task.
- Helps **extend the capabilities** of the AI. For example:
  - Searching YouTube
  - Getting weather data
  - Calculating math problems
  - Searching Google
  - Talking to databases

## ✓ What is YoutubeChannelSearchTool ?

It's a tool from the `crewai_tools` package that allows your AI agents or scripts to **search for videos from a specific YouTube channel**, based on keywords or topics.

## ✓ What is happening in the code?

python

 Copy  Edit

```
from crewai_tools import YoutubeChannelSearchTool
yt_tool = YoutubeChannelSearchTool(youtube_channel_handle='@krishnaik06')
```

| Line   | What It Does  |
|--|---|
| <code>from crewai_tools import</code><br><code>YoutubeChannelSearchTool</code> | Imports the tool class that enables YouTube channel searches.                                       |
| <code>YoutubeChannelSearchTool(...)</code>                                     | Initializes the search tool for a specific channel.   |
| <code>youtube_channel_handle='@krishnaik06'</code>                             | Targets the <b>Krish Naik's</b> YouTube channel (a popular AI/ML educator).                         |
| <code>yt_tool</code>   | This object can now be used to <b>search videos</b> from Krish Naik's channel using search queries. |

## ✓ Why is this useful?

- It allows your AI assistant to **search only one channel** rather than all of YouTube.
- Great for **domain-specific knowledge** (like ML from Krish Naik).
- You can use it in **agents, chatbots, or RAG apps** to extract or summarize YouTube video content from trusted sources.

## 🧠 Example Use Case

Later, you might call something like:

python

 Copy  Edit

```
results = yt_tool.search("LangChain tutorial")
```

This would return videos **from @krishnaik06's channel** that match "LangChain tutorial."

## What's Going On?

You're building **AI agents with tasks** using [CrewAI](#)—a framework that lets you automate complex workflows using smart agents, tools, and tasks. You're assigning specific **goals (tasks)** to agents that will use specific **tools** to complete them.

---

## Step-by-Step Understanding (No Code)

1. **Agent:** Think of this like a character or employee with a special skill (e.g., a researcher or a writer).
2. **Tool:** Just like a hammer helps a carpenter, a tool here helps an agent do something—like search YouTube videos.
3. **Task:** This is a job or a mission. It tells the agent:
  - What to do (description)
  - What the result should look like (expected output)
  - What tools they can use
  - Where to save the result (optional)
4. **Execution:**
  - You first **research** using one agent and tool.
  - Then you **write** using the findings from that task with another agent.

## Full Process Overview

| Step | What Happens   | Who/What Does It   |
|------|--|--|
| 1    | You create a <b>task</b> to research YouTube videos about a topic. | <code>blog_researcher</code> agent uses <code>yt_tool</code> . |
| 2    | You specify the output (3 paragraphs of insights).                 | The <code>Task</code> object.                                  |
| 3    | Then you create another <b>task</b> to write a blog post.          | <code>blog_writer</code> uses the same <code>yt_tool</code> .  |
| 4    | The final content is saved in a markdown file.                     | The <code>write_task</code> .                                  |

## What Each Library Does

| Library | Purpose   |
|---------|---|
| crewai  | Main framework to build agent-task-tool workflows.  |
| tools   | Custom or built-in tools (e.g., YouTube search).    |
| agents  | Your predefined AI agents (like roles or personas). |

### Summary (For Future PhDs)

You're using CrewAI to build a chain of smart tasks. One AI agent gathers video data using a tool. Another uses that data to write a blog. These steps mirror real-world job delegation—research → writing—automated by AI.

```
from crewai import Task # Core task structure from CrewAI
from tools import yt_tool # YouTube search tool
from agents import blog_researcher, blog_writer # Two AI agents

# □ Task 1: Research YouTube content about a topic
research_task = Task(
    description=(
        "Identify the video about {topic} "
        "and gather detailed information from it."
    ),
    expected_output='A comprehensive 3-paragraph report on the video content related to
{topic}.',
    tools=[yt_tool], # Tool to search videos from a YouTube channel
    agent=blog_researcher, # Agent assigned for research
)

# ↗ Task 2: Write a blog post based on the research
write_task = Task(
    description=(
        "Use information from the YouTube channel on the topic {topic} "
        "to write a blog post."
    ),
    expected_output='A blog post summarizing the YouTube video content about {topic}.',
    tools=[yt_tool], # Reuse the same tool
    agent=blog_writer, # Agent assigned for writing
```

```
    async_execution=False, # Run this synchronously
    output_file='new-blog-post.md', # Save result here
)
```

## Orchestration

Great question!

---

### 🧠 What is an Orchestration Layer or Tool?

Think of the **orchestration layer** as the “**director**” of a movie 🎬.

- The **actors** are your **agents** (like `blog_writer` and `blog_researcher` ).
  - The **script** is your **tasks** (like “research a topic”, “write a blog”).
  - The **orchestration tool** (like `CrewAI`, `LangGraph`, or `LangChain`) is the **director** who makes sure everyone plays their part, in the right order, with the right tools, and at the right time.
- 

### 🎯 Formal Definition

Orchestration is the process of **coordinating and managing multiple agents, tools, and tasks** to work together efficiently and intelligently toward a common goal.

## Why Is It Called a “Tool”?

A **tool** in this context is **any utility or component** that helps an agent perform a specific function, such as:

- Searching YouTube → `YoutubeChannelSearchTool`
- Doing math → `CalculatorTool`
- Accessing documents → `PDFLoader`, `Retriever`
- Calling an API → `RequestsTool`

The **orchestration layer uses tools to empower agents**, just like giving a chef a knife or a carpenter a hammer.

---

## Example in Real Life

Imagine you're building an AI Assistant that:

1. Searches YouTube for video content.
2. Summarizes it.
3. Writes a blog post.
4. Posts it to WordPress.

You need **orchestration** to:

- Sequence each of those steps.
- Pass data from one to the next.
- Handle errors.
- Remember what was said.
- Use the right tool at the right time.

## Summary Table

| Concept                    | What it means                              | Examples                              |
|----------------------------|--|---------------------------------------|
| <b>Tool</b>                | A utility that performs a task             | YouTube search, calculator            |
| <b>Agent</b>               | An AI with a goal, persona, and tools      | <code>blog_writer</code> , researcher |
| <b>Task</b>                | A specific job assigned to an agent        | "Summarize video", "Write blog"       |
| <b>Orchestration Layer</b> | The director managing agents, tools, tasks | CrewAI, LangChain, LangGraph          |

You're now putting together the final **orchestration layer** for your AI agents using **CrewAI**—this is where your agents and tasks are connected and executed as a real workflow.

Let's break it down like a **PhD-level guide for a curious 5th grader**, then summarize the entire flow and code functionality.

---

## What's Happening in Simple Terms

### 1. You've built 2 intelligent agents:

- One to **research** YouTube videos.
- Another to **write** a blog based on that research.

### 2. You gave each one a job (task):

- **Task 1:** Research content from YouTube about a topic.
- **Task 2:** Write a blog post using the research.

### 3. Now, you're **assembling your team (crew)** using `Crew()`:

- It includes the agents and the tasks.
- You tell it to **run the tasks one after another** (sequential).
- You turn on memory so it remembers the conversation.
- You enable caching so it reuses past results if needed.
- You limit the rate (`max_rpm`) to 100 requests per minute.
- You set `share_crew=True` so the output is reusable elsewhere.

### 4. Finally, you **start the execution** using `.kickoff()` and pass in the **topic** (like: "AI vs ML vs DL vs Data Science").

---

## Step-by-Step Process (No Code)

| Step | What Happens                                | Who/What Does It                                     |
|------|---|--|
| 1    | Form a team (crew) with 2 agents.           | <code>crew()</code>                                  |
| 2    | Assign tasks to each agent.                 | <code>research_task</code> , <code>write_task</code> |
| 3    | Configure how they work together.           | Sequential, with memory and caching                  |
| 4    | Run the entire process on a specific topic. | <code>.kickoff(inputs={...})</code>                  |
| 5    | Get final output (blog post).               | Print result   |

```

from crewai import Crew, Process
from agents import blog_researcher, blog_writer
from tasks import research_task, write_task

# □□ Form the AI-powered crew with two smart agents and their tasks
crew = Crew(
    agents=[blog_researcher, blog_writer], # List of AI agents
    tasks=[research_task, write_task], # Their assigned tasks
    process=ProcessSEQUENTIAL, # Tasks will run one after the other
    memory=True, # Enable memory between agents
    cache=True, # Cache results to avoid redundant work
    max_rpm=100, # Limit to 100 requests/minute
    share_crew=True # Allow sharing output between agents
)

# □ Start the execution of the crew with the given input topic
result = crew.kickoff(inputs={'topic': 'AI VS ML VS DL vs Data Science'})

# 📄 Print the final result (the blog post)
print(result)

```

## GRAPH DB

---

You're working on an advanced AI system that connects a **Graph Database (Neo4j)** with a **Large Language Model (LLM)** using **LangChain**, and you're asking it questions about **movie data**. Let's explain this entire project from scratch in a way even a 5th-grader can eventually become a PhD-level expert!

---

## What's Happening? (No Code, Just Concept)

Imagine you have a **big notebook (database)** filled with pages listing **movies**, their **actors**, **directors**, **genres**, and **release dates**. But it's not written like a regular notebook. It's written in a **web of connections**, like a **mind map** — this is what **Neo4j**, a **graph database**, stores.

Now, let's say you want to ask:

"Who directed the movie Casino?"

Instead of reading the notebook yourself, you ask a **very smart robot (LLM)**, and the robot replies by looking into the notebook, understanding how things connect, and writing a special query language (Cypher) to fetch the answer.

---

## Step-by-Step Breakdown of the Process

| Step | What Happens   | Why It's Important                             |
|------|--|--|
| 1    | Load movie data into Neo4j from a CSV file.              | Fills the database with real-world movie info. |
| 2    | Create connections like "Tom Hanks ACTED_IN Cast Away".  | Graph databases are about relationships!       |
| 3    | Use LangChain to link a language model (LLM) with Neo4j. | So we can ask questions in plain English.      |

- 
- |   |   |   |
|---|---|---|
| 4 | Add example questions + answers (called <b>few-shot examples</b> ). | Trains the robot how to form the right query. |
| 5 | Ask questions and let the model convert them into Cypher queries.   | LLM becomes your translator to Neo4j.         |
| 6 | Execute query → get results → return readable answer.               | Automates expert-level data querying.         |
- 



## What Each Tool/Library Does

| Tool/Library                       | Purpose   |
|------------------------------------|---|
| <code>Neo4jGraph</code>            | Connects LangChain to Neo4j graph database          |
| <code>ChatGroq</code>              | Uses Groq's blazing-fast LLMs like Gemma2-9b-It     |
| <code>GraphCypherQACChain</code>   | Translates English into Cypher queries using LLM    |
| <code>FewShotPromptTemplate</code> | Shows examples so the LLM can learn by imitation    |
| <code>PromptTemplate</code>        | Template to format the question + Cypher query      |
| <code>load_dotenv</code>           | Loads secret API keys from a <code>.env</code> file |
| <code>os.environ</code>            | Sets environment variables like DB credentials      |
| <code>chain.invoke(...)</code>     | Executes the QA pipeline to answer questions        |

```
# 1. Load secrets and connect to the Neo4j database
import os
from dotenv import load_dotenv
load_dotenv()

from langchain_community.graphs import Neo4jGraph

# Connect using environment variables
graph = Neo4jGraph(
    url=os.getenv("NEO4J_URI"),
    username=os.getenv("NEO4J_USERNAME"),
    password=os.getenv("NEO4J_PASSWORD")
```

```

)

# 2. Load movie dataset from CSV into Neo4j
movie_query = """
LOAD CSV WITH HEADERS FROM
'https://raw.githubusercontent.com/tomasonjo/blog-datasets/main/movies/movies_small.csv' AS
row

MERGE (m:Movie {id: row.movieId})
SET m.title = row.title,
    m.released = date(row.released),
    m.imdbRating = toFloat(row.imdbRating)

FOREACH (director IN split(row.director, '|') |
    MERGE (p:Person {name: trim(director)})
    MERGE (p)-[:DIRECTED]->(m)
)

FOREACH (actor IN split(row.actors, '|') |
    MERGE (p:Person {name: trim(actor)})
    MERGE (p)-[:ACTED_IN]->(m)
)

FOREACH (genre IN split(row.genres, '|') |
    MERGE (g:Genre {name: trim(genre)})
    MERGE (m)-[:IN_GENRE]->(g)
)
"""

# Run the query to populate the graph
graph.query(movie_query)
graph.refresh_schema()
print(graph.schema) # See structure of the data

# 3. Load the LLM from Groq
from langchain_groq import ChatGroq
llm = ChatGroq(groq_api_key=os.getenv("GROQ_API_KEY"), model_name="Gemma2-9b-It")

# 4. Set up few-shot examples to help the LLM learn how to write Cypher queries
from langchain_core.prompts import FewShotPromptTemplate, PromptTemplate

examples = [
    {
        "question": "How many artists are there?",
        "query": "MATCH (a:Person)-[:ACTED_IN]->(:Movie) RETURN count(DISTINCT a)",
    },
    {
        "question": "Which actors played in the movie Casino?",
```

```

    "query": "MATCH (m:Movie {title: 'Casino'})<-[ACTED_IN]-(a) RETURN a.name",
},
# Add more examples as needed
]

# Format each example
example_prompt = PromptTemplate.from_template(
    "User input: {question}\nCypher query: {query}"
)

# Create a few-shot prompt for the LLM to learn from
prompt = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    prefix="You are a Neo4j expert. Given an input question, create a Cypher query.",
    suffix="User input: {question}\nCypher query: ",
    input_variables=["question", "schema"]
)

# Optional: test the prompt
print(prompt.format(question="How many artists are there?", schema="movie schema"))

# 5. Create the Graph Cypher QA Chain
from langchain.chains import GraphCypherQACChain

chain = GraphCypherQACChain.from_llm(
    graph=graph,
    llm=llm,
    cypher_prompt=prompt,
    verbose=True
)

# 6. Ask Questions!
chain.invoke("Which actors played in the movie Casino?")
chain.invoke("Which actors acted in multiple movies?")

```

## Summary

- You built a **question-answering system** powered by Neo4j + LLM.
- It converts natural language to Cypher queries, thanks to LangChain.
- Using **few-shot examples**, the LLM learns how to write queries correctly.
- The result is a **smart assistant** that can navigate complex graph data.

You're building a **chatbot using LangGraph**, a library that helps design stateful and reactive language model applications, combined with **Groq's Gemma-2 9B It** model via LangChain. Let's break this down in the clearest, most structured way possible—so even a curious 5th grader could eventually earn a PhD in AI engineering. We'll go step by step:

---

## Conceptual Explanation

### • What Are You Building?

A chatbot that:

- Takes user input.
  - Sends it to a large language model (Gemma-2 9B).
  - Returns the model's response.
  - Uses a **state machine** to manage the conversation flow.
- 

## Step-by-Step Explanation (No Code Yet)

### 1. API Keys Setup

You're storing sensitive keys (like Groq and LangSmith) securely using Colab's `userdata` dictionary and `os.environ` to make them accessible in your code.

---

## 2. Define the Conversation Memory (State)

You're setting up a `State` that holds a list of all messages so far. This acts like the **memory of your chatbot**.

## 3. Create a Graph (Workflow)

You're using `LangGraph` to define a simple **state graph** with:

- A starting point.
- A node named `"chatbot"` where the actual message is processed by the model.
- An end point.

## 4. Node Logic (The Chatbot Brain)

You define a function (`chatbot`) that:

- Receives all previous messages.
- Sends them to the language model (`lm.invoke(...)`).
- Appends the response back to the message list.

## 5. Compile the Graph

You "compile" the state machine to get it ready for execution.

## 6. Run Chatbot in Loop

The bot runs in a `while True` loop:

- Takes user input.
- Feeds it to the graph.
- Streams the model response.
- Keeps running until the user types "quit".

## What Each Library Does

| Library   | Purpose   |
|---|---|
| <code>os</code>                                 | Sets environment variables for secure API access.                         |
| <code>google.colab.userdata</code>              | Retrieves secret values in Google Colab.                                  |
| <code>langchain_groq</code>                     | Connects to Groq's fast LLMs like Gemma-2 9B.                             |
| <code>langgraph</code>                          | Builds the <b>stateful flow</b> of conversations (start → chatbot → end). |
| <code>TypedDict</code> , <code>Annotated</code> | Used to define structured state memory and update behavior.               |
| <code>add_messages</code>                       | Automatically appends messages to the history in the state.               |

## Summary

You're building a **chatbot with memory** using:

- **LangChain** to interface with Groq's powerful LLM.
- **LangGraph** to design the conversation as a flexible graph.
- A loop that **streams responses** and maintains chat history.

This is a production-grade architecture in an elegant and minimal form. Great work setting this up!

Would you like to visualize the graph using Mermaid or see how to expand this with tool usage or multiple nodes?

```
# ✓ Step 1: Setup - Access keys securely from Colab userdata
```

```
from google.colab import userdata
```

```
import os
```

```
# Store environment variables securely
```

```
groq_api_key = userdata.get('groq_api_key')
```

```
langsmith = userdata.get('LANGSMITH_API_KEY')
```

```
# Set LangChain tracing environment
```

```
os.environ["LANGCHAIN_API_KEY"] = langsmith
```

```
os.environ["LANGCHAIN_TRACING_V2"] = "true"
```

```
os.environ["LANGCHAIN_PROJECT"] = "CourseLanggraph"
```

```
# ✓ Step 2: Import and load the Groq-powered Gemma model
```

```

from langchain_groq import ChatGroq
llm = ChatGroq(groq_api_key=groq_api_key, model_name="Gemma2-9b-It")

# ✅ Step 3: Define the chatbot memory (state)
from typing import Annotated
from typing_extensions import TypedDict
from langgraph.graph.message import add_messages

class State(TypedDict):
    # This holds a list of all messages and appends new ones
    messages: Annotated[list, add_messages]

# ✅ Step 4: Build the state machine graph
from langgraph.graph import StateGraph, START, END

# Initialize the graph builder
graph_builder = StateGraph(State)

# Define the main chatbot logic
def chatbot(state: State):
    # Sends the latest conversation to the LLM and appends the response
    return {"messages": llm.invoke(state['messages'])}

# Add chatbot logic to the graph
graph_builder.add_node("chatbot", chatbot)

# Connect the nodes: Start -> Chatbot -> End
graph_builder.add_edge(START, "chatbot")
graph_builder.add_edge("chatbot", END)

# ✅ Step 5: Compile the graph
graph = graph_builder.compile()

# ✅ Step 6: Chat loop
while True:
    user_input = input("User: ")
    if user_input.lower() in ["quit", "q"]:
        print("Goodbye!")
        break

    # Run the graph with the latest user input
    for event in graph.stream({'messages': ("user", user_input)}):
        # Print each update from the graph
        for value in event.values():
            print("Assistant:", value["messages"].content)

```

---

## Step-by-Step Understanding (No Code)

1. **Goal:** You're building a chatbot that can talk to you and use tools like:

-  **Wikipedia** (for general knowledge)
-  **Arxiv** (for scientific papers)

2. **What is a Tool?**

A tool is like giving your chatbot a “superpower.” Without tools, the chatbot only talks. With tools, it can search Wikipedia or read research papers to answer better.

3. **What's a LangGraph?**

Think of it like a **map** or **flowchart** showing:

- Where the chatbot starts
- When it should use a tool
- When it should respond to you
- How to repeat this loop if needed

4. **LLM + Tools**

You give the chatbot (LLM from Groq) the ability to:

- Decide if it needs help (tool condition)
- Use the right tool
- Come back and give you an answer

## Full Process Flow (Plain Language)

| Step | What Happens   |
|------|--|
| 1.   | You input a message like “Who is Shahrukh Khan?”           |
| 2.   | The chatbot thinks: “Can I answer this or do I need help?” |
| 3.   | If it needs help, it uses the Wikipedia tool.              |
| 4.   | It gets the answer and continues talking to you.           |
| 5.   | This cycle continues for each question.                    |

## Summary of Each Library

| Library  | Purpose  |
|--|--|
| <code>langchain_community.utilities</code>           | Lets you search Wikipedia, Arxiv, etc.                         |
| <code>langchain_community.tools</code>               | Wraps those utilities as usable tools                          |
| <code>langchain_groq.ChatGroq</code>                 | Connects to Groq's fast LLM (Gemma-2 9B It)                    |
| <code>langgraph.graph</code>                         | Builds a flowchart of chatbot logic                            |
| <code>ToolNode</code> , <code>tools_condition</code> | Add tools to the graph and let the LLM decide when to use them |
| <code>google.colab.userdata</code>                   | Gets secret keys stored securely in Colab                      |
| <code>add_messages</code>                            | Handles how messages are stored and passed between nodes       |

## ✓ Code Explanation (Cleaned + Explained)

python

 Copy  Edit

```
# Import tool interfaces
from typing import Annotated
from typing_extensions import TypedDict

# Tool APIs from LangChain
from langchain_community.utilities import ArxivAPIWrapper, WikipediaAPIWrapper
from langchain_community.tools import ArxivQueryRun, WikipediaQueryRun

# Create Arxiv tool with 1 result and 300 characters max
arxiv_wrapper = ArxivAPIWrapper(top_k_results=1, doc_content_chars_max=300)
arxiv_tool = ArxivQueryRun(api_wrapper=arxiv_wrapper)

# Create Wikipedia tool similarly
wiki_wrapper = WikipediaAPIWrapper(top_k_results=1, doc_content_chars_max=300)
wiki_tool = WikipediaQueryRun(api_wrapper=wiki_wrapper)

# All tools available to the chatbot
tools = [wiki_tool, arxiv_tool] # You can add more tools here
```

## Define the LangGraph State

python

 Copy  Edit

```
# Define the conversation state structure
from langgraph.graph.message import add_messages
class State(TypedDict):
    messages: Annotated[list, add_messages]
```

## Setup Graph and Groq Model

python

 Copy  Edit

```
from langgraph.graph import StateGraph, START, END
from langchain_groq import ChatGroq
from google.colab import userdata

# Load API key for Groq
groq_api_key = userdata.get("groq_api_key")
llm = ChatGroq(groq_api_key=groq_api_key, model_name="Gemma2-9b-It")

# Bind tools to the LLM (give it superpowers)
llm_with_tools = llm.bind_tools(tools=tools)
```

## Define Chat Node

python

 Copy  Edit

```
# This function will be run when the chatbot node is activated
def chatbot(state: State):
    # It invokes the LLM with the current messages
    return {"messages": [llm_with_tools.invoke(state["messages"])]}
```

## Build Graph Flow

python

 Copy  Edit

```
from langgraph.prebuilt import ToolNode, tools_condition

# Create graph blueprint
graph_builder = StateGraph(State)

# Add chatbot and tool node
graph_builder.add_node("chatbot", chatbot)
graph_builder.add_node("tools", ToolNode(tools=tools))

# Set chatbot to decide: continue talking or use tool
graph_builder.add_conditional_edges("chatbot", tools_condition)

# If tools are used, loop back to chatbot
graph_builder.add_edge("tools", "chatbot")
graph_builder.add_edge(START, "chatbot")

# Compile final graph
graph = graph_builder.compile()
```

## Run the Chatbot

```
python

# First user message
user_input = "Hi there!, My name is John"
events = graph.stream({"messages": [("user", user_input)]}, stream_mode="values")
for event in events:
    event["messages"][-1].pretty_print()

# Second user message (uses Wikipedia)
user_input = "what is RLHF."
events = graph.stream({"messages": [("user", user_input)]}, stream_mode="values")
for event in events:
    event["messages"][-1].pretty_print()
```

## 🔍 Summary of Each Library

| Library  | Purpose  |
|--|--|
| <code>langchain_community.utilities</code>           | Lets you search Wikipedia, Arxiv, etc.                         |
| <code>langchain_community.tools</code>               | Wraps those utilities as usable tools                          |
| <code>langchain_groq.ChatGroq</code>                 | Connects to Groq's fast LLM (Gemma-2 9B It)                    |
| <code>langgraph.graph</code>                         | Builds a flowchart of chatbot logic                            |
| <code>ToolNode</code> , <code>tools_condition</code> | Add tools to the graph and let the LLM decide when to use them |
| <code>google.colab.userdata</code>                   | Gets secret keys stored securely in Colab                      |
| <code>add_messages</code>                            | Handles how messages are stored and passed between nodes<br>↓  |

## FINE - TUNING - Lamini

Absolutely! Let's turn this code and process into a **crystal-clear, easy-to-follow journey**, and by the end, you'll understand **how Lamini fine-tunes large language models (LLMs)** with simple data.

---

### 1. 5th Grader to PhD — Core Idea

Imagine you have a super-smart robot that can talk. But this robot speaks very generally, and you want it to **specialize in answering questions about your product or website**, like a customer support agent. How do you teach it?

You **give it examples** — questions and answers — and say:

“Hey, this is how / answer these questions. Learn from it!”

Lamini helps you do that — it's like a **tutor** that trains the robot using those examples (data). You don't need to write complex code or use giant servers — Lamini takes care of all that.

## 2. Step-by-Step Process (Without Code)

| Step | What Happens  | Why We Do It  |
|------|---|---|
| 1    | Define Q&A examples (training data)                       | So the model can learn patterns from specific questions and answers |
| 2    | Import Lamini tools and set your API key                  | This lets you use Lamini's cloud services                           |
| 3    | Pick the model you want to fine-tune (e.g., LLaMA-3)      | This is your smart robot's brain                                    |
| 4    | Feed the Q&A data to the model using <code>.tune()</code> | This fine-tunes the model's brain based on your examples            |
| 5    | Customize with hyperparameters (like learning rate)       | These control how fast and how long the robot learns                |

## 3. Detailed Explanation of the Full Process

### 1. Your Training Data

You create a list of **question-answer pairs**, like:

- **Q:** Can I use Lamini offline?
- **A:** Yes, if you run a local instance...

This tells the model **exactly how to respond** to similar questions.

## 2. Lamini Platform

You use the `lamini` library and `Lamini` class to:

- Connect to the Lamini engine via an **API key**
  - Select which model you want to fine-tune (like `Meta-Llama-3-8B-Instruct`)
  - Call `.tune()` to **upload your data and train the model**
- 

## 3. What `.tune()` Does

Lamini takes:

- Your question-answer pairs
- Your **settings** (learning rate, optimizer, etc.)

And starts fine-tuning in the cloud — meaning:

- You **don't need your own GPU**
- It handles **hyperparameter tuning**
- The model learns how **you** want it to behave



## Explanation of Each Library and Function

| Item                       | What It Does   |
|----------------------------|--|
| <code>lamini</code>        | Python SDK to access Lamini's fine-tuning and inference features                 |
| <code>Lamini</code>        | Class that represents a model (like LLaMA) you want to fine-tune                 |
| <code>.api_key</code>      | Authenticates your access to Lamini's cloud service                              |
| <code>.tune()</code>       | Fine-tunes the selected model using your dataset and tuning settings             |
| <code>learning_rate</code> | Controls how fast the model updates its brain — too high = messy, too low = slow |



## Summary

You are using **Lamini**, a powerful tool to fine-tune LLMs like **LLaMA-3** using **simple question-answer data**. This process:

- **Trains your model** to respond exactly how you want
- **Requires no GPUs or model hacking**
- **Supports advanced tuning options**

```
# -----
# STEP 1: Define training data
# -----
# These are example question-answer pairs to fine-tune the model
def get_data():
    return [
        {
            "input": "Are there any step-by-step tutorials or walkthroughs available in the documentation?",
            "output": "Yes, there are step-by-step tutorials and walkthroughs available in the documentation section. Here's an example for using Lamini: https://lamini-ai.github.io/example/",
        },
        {
            "input": "Is the Lamini type system similar to a python type system?",
            "output": "Yes, the Lamini type system is built using Pydantic BaseModel.",
        },
    ]
```

```

    "input": "Does Lamini have a limit on the number of API requests I can make?",  

    "output": "Lamini provides each user with free tokens up front.",  

  },  

  {  

    "input": "What does it mean to cancel a job using the `cancel_job()` function? Can we  

stop the machine from doing its task?",  

    "output": "The `cancel_job()` function is used to stop a tuning job that is currently  

running.",  

  },  

  {  

    "input": "Can Lamini automatically handle hyperparameter tuning during the  

customization process?",  

    "output": "Yes, Lamini can automatically handle hyperparameter tuning using intelligent  

search techniques like grid search or Bayesian optimization.",  

  },  

  {  

    "input": "What is Lamini? Is it like a robot or a computer program?",  

    "output": "Lamini is a software tool called an LLM engine — it runs and customizes  

large language models. It's not a robot, but a powerful language program.",  

  },  

]  

  

# -----  

# STEP 2: Import Lamini and configure it  

# -----  

import lamini  

from lamini import Lamini  

  

# Set your Lamini API key  

lamini.api_key = "your_api_key_here" # 🚫 Replace with your actual key  

  

# Choose which model to fine-tune (Meta's LLaMA 3 in this case)  

llm = Lamini(model_name="meta-llama/Meta-Llama-3-8B-Instruct")  

  

# -----  

# STEP 3: Fine-tune the model on your data  

# -----  

data = get_data()  

  

# Start fine-tuning with optional hyperparameters  

llm.tune(  

  data_or_dataset_id=data,  

  finetune_args={  

    "learning_rate": 1e-4, # You can also specify: max_steps, early_stopping, optim, etc.  

  }  

)

```

## Finetune gemini

Perfect! Let's break this down in **super simple terms** — as if you're starting in 5th grade and will graduate as a PhD in fine-tuning language models using Hugging Face, QLoRA, and Gemma 🚀

---

### 🧠 1. Step-by-Step (No Code) — Like a Story

#### ✳️ Step 1: Gather Your Tools

We install tools like:

- **Transformers** (the brain)
  - **Datasets** (the memory)
  - **TRL & PEFT** (for custom fine-tuning)
  - **BitsAndBytes** (makes big brains fit into small computers)
- 

#### 🧠 Step 2: Pick a Brain (LLM)

We choose **Gemma-2B**, a smart language model from Google. It's like picking a student to train.

But this student is *huge*, so we shrink it with **quantization** (4-bit), meaning it uses less memory.

---

## Step 3: Teach It to Write Quotes

We take famous English quotes like:

“Imagination is more important than knowledge.”

We want our model to **complete** quotes or generate new ones in the same style.

---

## Step 4: Use a Teaching Method (QLoRA)

QLoRA is a technique to teach a model **efficiently** with fewer resources.

- We use **LoRAConfig** to set how it learns: small updates only in specific parts of the brain.
- 

## Step 5: Feed the Model Examples

We load a **dataset of English quotes** and teach the model:

- “This is how a quote looks.”
  - “This is how a famous person writes.”
- 

## Step 6: Train the Model

We train it for 100 steps using **SFTTrainer**:

- Small batches
- Accumulate gradients
- Use an optimizer called **paged\_adamw\_8bit** for memory-efficient training

## Step 7: Test It

We give the model half-written quotes like:

"A woman is like a tea bag;"

It tries to **complete it** using what it has learned!

---

## Detailed Explanation of the Libraries

| Library                   | Purpose   |
|---------------------------|---|
| <code>transformers</code> | Brings pre-trained language models (like Gemma) and tools to use them |
| <code>torch</code>        | Backend engine (like the electricity for your robot brain)            |
| <code>datasets</code>     | Loads ready-made datasets (e.g. quotes)                               |
| <code>trl</code>          | For fine-tuning models using Trainer with special features            |
| <code>peft</code>         | Enables <b>parameter-efficient fine-tuning</b> (like LoRA)            |
| <code>bitsandbytes</code> | Helps run big models using less memory (4-bit quantization)           |



## Summary

| Step                          | Purpose  |
|-------------------------------|--|
| Install libraries             | Sets up all tools ( <code>transformers</code> , <code>peft</code> , <code>bitsandbytes</code> , <code>trl</code> ) |
| Load quantized Gemma-2B model | Saves memory and loads faster  |
| Format and load quote dataset | Prepares data for fine-tuning  |
| Configure QLoRA               | Trains model with minimal compute cost   |
| Fine-tune with SFTTrainer     | Actually teaches the model with your data  |
| Generate completions          | Test how well it learned to finish quotes  |

# -----

```

# STEP 1: Install Tools
# -----
!pip3 install -q -U bitsandbytes==0.42.0 peft==0.8.2 trl==0.7.10 accelerate==0.27.1
datasets==2.17.0 transformers==4.38.0

# -----
# STEP 2: Import Modules
# -----
import os
import torch
import transformers
from datasets import load_dataset
from trl import SFTTrainer
from peft import LoraConfig
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig

# -----
# STEP 3: Auth Setup
# -----
# Google Colab secure access for HuggingFace token
from google.colab import userdata
os.environ["HF_TOKEN"] = userdata.get('HF_TOKEN') # Add your HF Token to Colab secrets
# -----
# STEP 4: Load Quantized Model
# -----
model_id = "google/gemma-2b"

# Configure model to load in 4-bit (saves memory)
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)

# Load tokenizer and model
tokenizer = AutoTokenizer.from_pretrained(model_id, token=os.environ['HF_TOKEN'])
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    quantization_config=bnb_config,
    device_map={"/": 0},
    token=os.environ['HF_TOKEN']
)

# -----
# STEP 5: Test Before Tuning
# -----
text = "Quote: Imagination is more"

```

```

inputs = tokenizer(text, return_tensors="pt").to("cuda:0")

# Generate output
outputs = model.generate(**inputs, max_new_tokens=20)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))

# -----
# STEP 6: Configure LoRA (QLoRA)
# -----
lora_config = LoraConfig(
    r=8,
    target_modules=["q_proj", "o_proj", "k_proj", "v_proj", "gate_proj", "up_proj",
    "down_proj"],
    task_type="CAUSAL_LM"
)

# -----
# STEP 7: Load Quote Dataset
# -----
data = load_dataset("Abirate/english_quotes")

# Tokenize all quotes
data = data.map(lambda samples: tokenizer(samples["quote"]), batched=True)

# -----
# STEP 8: Format Each Example
# -----
def formatting_func(example):
    # Format as: "Quote: ....\nAuthor: ...."
    return f"Quote: {example['quote'][0]}\nAuthor: {example['author'][0]}"

# -----
# STEP 9: Fine-tune with TRL
# -----
trainer = SFTTrainer(
    model=model,
    train_dataset=data["train"],
    args=transformers.TrainingArguments(
        per_device_train_batch_size=1,
        gradient_accumulation_steps=4,
        warmup_steps=2,
        max_steps=100,
        learning_rate=2e-4,

```

```
        fp16=True,
        logging_steps=1,
        output_dir="outputs",
        optim="paged_adamw_8bit"
    ),
    peft_config=lora_config,
    formatting_func=formatting_func,
)

# -----
# STEP 10: Try New Completions
# -----
texts = [
    "Quote: A woman is like a tea bag;",
    "Quote: Outside of a dog, a book is man's"
]
for text in texts:
    inputs = tokenizer(text, return_tensors="pt").to("cuda:0")
    outputs = model.generate(**inputs, max_new_tokens=20)
    print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

S