Table of Contents

[LANGCHAIN 3](#_Toc208752507)

[UNDERSTANDING LANGCHAIN COMPONENTS USING RAG EXAMPLE 3](#_Toc208752508)

[KEY FEATURES OF LANGCHAIN 4](#_Toc208752509)

[CONCEPTS 5](#_Toc208752510)

[CHAINING 5](#_Toc208752511)

[COMPONENTS OF LANGCHAIN 6](#_Toc208752512)

[CHAT MODELS 6](#_Toc208752513)

[LANGCHAIN IN ACTION 6](#_Toc208752514)

[PROMPT TEMPLATE 7](#_Toc208752515)

[EXAMPLE 8](#_Toc208752516)

[Partial() 8](#_Toc208752517)

[CHAT PROMPT TEMPLATE 9](#_Toc208752518)

[WHEN TO USE WHAT? 10](#_Toc208752519)

[OUTPUT PARSER 10](#_Toc208752520)

[COMMON OUTPUT PARSERS IN LANGCHAIN 11](#_Toc208752521)

[CREATING SIMPLE GENAI APPLICATION 12](#_Toc208752522)

[RETRIEVAL CHAIN 12](#_Toc208752523)

[CODE 14](#_Toc208752524)

[CHAINS 15](#_Toc208752525)

[MESSAGE TYPES (langchain\_core.messages) 15](#_Toc208752526)

[CLASS HIERARCHY 16](#_Toc208752527)

[MESSAGE TYPES 16](#_Toc208752528)

[FUNCTION MESSAGE EXAMPLE 19](#_Toc208752529)

[LANGCHAIN EXPRESSION LANGUAGE (LCEL) – SIMPLE DEFINITION 20](#_Toc208752530)

[SEQUENTIAL PROMPTS 21](#_Toc208752531)

[SIMPLE SEQUENTIAL CHAIN 22](#_Toc208752532)

[REGULAR SEQUENTIAL CHAIN 22](#_Toc208752533)

[LANGSERVE 22](#_Toc208752534)

[DEPLOYMENT 22](#_Toc208752535)

[CHAT HISTORY 22](#_Toc208752536)

[MAINTAINING HISTORY 22](#_Toc208752537)

[EMBEDDING MODELS 22](#_Toc208752538)

[EMBEDDINGS 22](#_Toc208752539)

[HOW EMBEDDINGS ARE GENERATED 23](#_Toc208752540)

[GENERATING EMBEDDINGS USING LANGCHAIN 24](#_Toc208752541)

[SIMILARITY FINDER 24](#_Toc208752542)

[HOW IT WORKS? 25](#_Toc208752543)

[VECTORS 26](#_Toc208752544)

[WHY ARE VECTORS USEFUL? 26](#_Toc208752545)

[VECTOR DATABASE 27](#_Toc208752546)

[RAG (RETRIEVER AUGUMENTED GENERATION) 31](#_Toc208752547)

[RAG WORKFLOW 31](#_Toc208752548)

[WHY RAG? 32](#_Toc208752549)

[BENEFITS 32](#_Toc208752550)

[RAG ARCHITECTURE 33](#_Toc208752551)

[RAG PROCESS- BUILDING RAG PIPELINE 33](#_Toc208752552)

[STEP 1: DATA INGESTION (USING DOCUMENT LOADERS) 34](#_Toc208752553)

[STEP 2- DATA CHUNKING 36](#_Toc208752554)

[STEP 3: CREATING EMBEDDINGS 43](#_Toc208752555)

[STEP 4: STORING AND QUERYING VECTOR DATA 46](#_Toc208752556)

[STEP 5: DATA RETRIEVAL (QUERING VECTOR DB) 50](#_Toc208752557)

[EXAMPLES – RAG PIPELINES 55](#_Toc208752558)

[EXAMPLE 1 – TEXT DOCUMENT DATA SOURCE 55](#_Toc208752559)

[EXAMPLE-2 – PDF DOCUMENT DATASOURCE 57](#_Toc208752560)

[EXAMPLE- 3 – WORD DOCUMENT DATASOURCE 58](#_Toc208752561)

[LANGSMITH 59](#_Toc208752562)

[STEP TO SET UP LANGSMITH 59](#_Toc208752563)

[CREATE API KEY 60](#_Toc208752564)

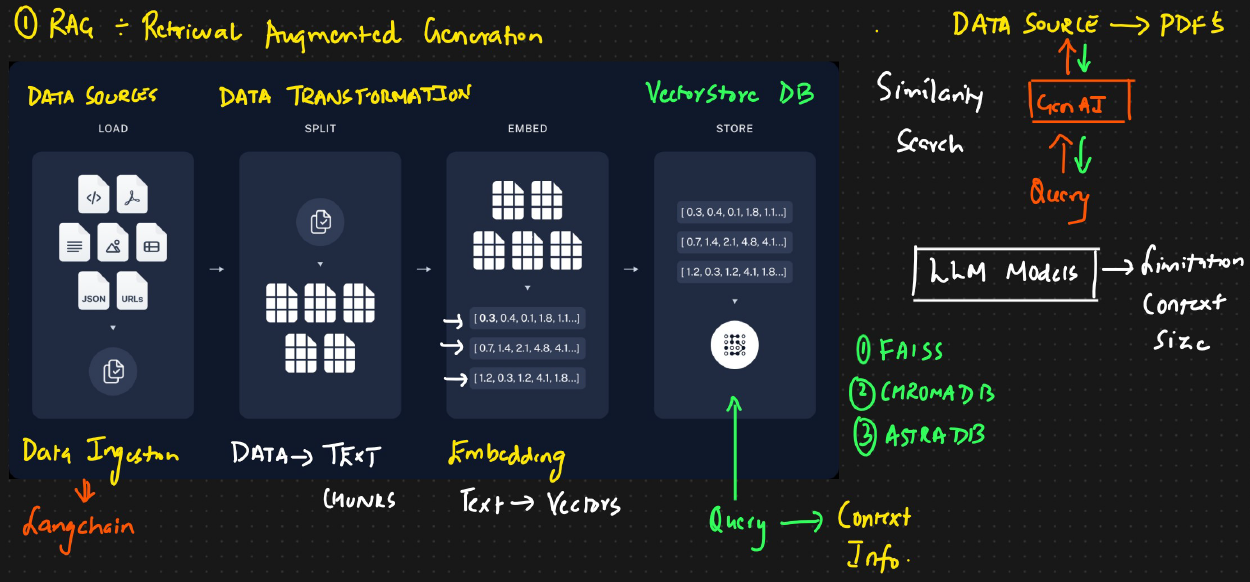
[CODE 60](#_Toc208752565)

[TRACING PROJECT 61](#_Toc208752566)

# LANGCHAIN

* **LangChain** is an open-source framework designed to help developers build applications powered by **large language models (LLMs)** like GPT-4.
* It provides tools to connect LLMs with external data sources, APIs, and memory, enabling more dynamic, context-aware, and interactive applications.

### UNDERSTANDING LANGCHAIN COMPONENTS USING RAG EXAMPLE

To build a RAG based application below are the steps we follow . In below examples we will see how langchain helps in performing those steps

Step 1 – Data Ingestion

1. Loading the data source

Step 2 – Data Chunking/Transformation

1. Splitting the data source into small text chunks

Step 3 –Creating Embeddings (Text Embeddings)

* The data chunks are broken into vectors
* These vectors are used to search using similarity search

Step 4 – Storing Embedding

* The generated embedding are used the stored in a DB called Vector Store DB
* Example – Chroma DB, FAISS, Astra DB

RETRIVING THE DATA

A screenshot of a computer

AI-generated content may be incorrect.

In this retrieval flow when user ask **any question we will see how** Lang Chain handles it behind the scenes.

Step 1: Designing the Prompt

* First, we create a **custom prompt**.  
  Think of it like giving instructions to your AI assistant.
* For example, we might say:  
  *“Hey, act like an AI researcher and answer all user questions.”*
* This prompt tells the language model **how to behave** when responding.

Step 2: Combining Prompt + User Question

* When a user asks a question, we **combine** it with our prompt.
* This combo becomes the input that we send to the system.

Step 3: Searching the Vector Store

* Before answering, the system checks the **Vector Store DB**.
* It looks for **similar content** or context related to the question.

Step 4: Understanding the Retrieval Chain

* The **Retrieval Chain** is the engine that queries the Vector Store.
* It helps us fetch the **right context** from our documents.

|  |
| --- |
| 1. The **Retrieval Chain** is a pipeline that:    * Takes user query.    * Searches a **Vector Store** (a database of embedded documents).    * Finds **relevant context** (like parts of PDFs, docs, or articles).   It’s like asking a librarian to find the best book pages that match your question.   1. The chain pulls out the most relevant chunks of information. |

Step 5: Getting the Final Answer

* Once we have the context from the vector store(on similarity search -vector DB return context from the vector DB), we **combine it with the prompt**.
* This full package is sent to the **Language Model (LM)**.
* The LM then gives us a **final response** based on both the context and the prompt.

## KEY FEATURES OF LANGCHAIN

1. LLM Integration

* Abstracts interaction with various LLMs (e.g., GPT, Claude, Mistral).
* Allows **easy switching between models** using a unified interface(Abstraction).
* Prevents **vendor lock-in** by supporting multiple providers.

2. Prompt Management

* Supports **prompt templates** and **dynamic prompt creation**.
* Helps with **prompt optimization** and **serialization**.

3. Document Loaders

* Load data from sources like PDFs, Notion, emails, etc.
* Converts them into a **standard LangChain document format** for easy processing.

4. Memory

* Maintains **conversation history** for context-aware interactions.

5. Tool & Agent Ecosystem

* Enables LLMs to use tools like:
  + Google Search
  + APIs
  + Databases
* Supports **agentic behavior** (reasoning + tool use).
* Includes components like **agent executors**, **tools**, and **LangGraph**.

## CONCEPTS

### CHAINING

* LangChain is a framework designed to bridge the gap between large language models and real-world applications.
* It works by **chaining** together various components called **links** to provide a flow. Each link in the chain represents a step in the process, from input to output.
* This modular approach allows for flexibility and customization, making LangChain adaptable to a wide range of applications.

A diagram of a software development process

AI-generated content may be incorrect.

Example

Consider a chatbot application.

1. The first link in the chain might be a language model that understands the user’s input.
2. The next link could be a decision-making model that determines the best response.
3. The final link might be another language model that generates the chatbot’s reply.

Each link is independent but interconnected, creating a seamless user experience.

### COMPONENTS OF LANGCHAIN



### CHAT MODELS

## LANGCHAIN IN ACTION

|  |  |
| --- | --- |
| **.env** | |
| **AZURE\_OPENAI\_API\_KEY=**  **AZURE\_OPENAI\_ENDPOINT=https://ai-dojo-open-ai.openai.azure.com/**  **AZURE\_OPENAI\_API\_VERSION=2024-12-01-preview**  **AZURE\_OPENAI\_DEPLOYMENT=gpt-4o-mini** | |
| **from langchain\_openai import AzureChatOpenAI**  **import os**  **from dotenv import load\_dotenv**  **load\_dotenv()**  **# Set your Azure OpenAI credentials**  **subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]**  **endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]**  **api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]**  **deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]**  **# Initialize the Azure OpenAI LLM**  **llm = AzureChatOpenAI(**  **azure\_deployment=deployment,**  **api\_version=api\_version,**  **azure\_endpoint=endpoint,**  **api\_key=subscription\_key,**  **)**  **# Example usage**  **response = llm.invoke("Capital of India!")**  **print(response.content)** |  |

## PROMPT TEMPLATE

* A **Prompt Template** in LangChain is a way to **structure and reuse prompts** for LLMs. It’s like a template string with placeholders you can fill with variables.
* Think of it as a blueprint for generating prompts dynamically.
* Used when we want to pass inputs to a single LLM call.

A white background with black text

AI-generated content may be incorrect.

Example

Let’s say we want to ask the LLM to summarize a topic.

Step 1: Create a Prompt Template

|  |
| --- |
| **from langchain.prompts import PromptTemplate**  **template = PromptTemplate(**  **input\_variables=["topic"],**  **template="Summarize the following topic in simple terms: {topic}"**  **)** |

Step 2: Format the Prompt with User Input

|  |
| --- |
| **prompt = template.format(topic="Quantum Computing")**  **print(prompt)** |

Output:

**Summarize the following topic in simple terms: Quantum Computing**

Now – We can now send this prompt to an LLM like GPT-4 to get a response.

Real-World Use Case

Imagine a chatbot that:

1. Takes user questions
2. Formats them into a prompt like:\ "Answer the following question clearly and concisely: {question}"
3. Sends it to the LLM
4. This keeps the logic clean and lets us swap out prompts easily without changing the whole app.

### EXAMPLE

|  |
| --- |
| import os  from langchain\_openai import AzureChatOpenAI  from langchain.prompts import PromptTemplate  from dotenv import load\_dotenv  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  **api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]**  **deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]**  **# Initialize the Azure OpenAI LLM**  **llm = AzureChatOpenAI(**  **azure\_deployment=deployment,**  **api\_version=api\_version,**  **azure\_endpoint=endpoint,**  **api\_key=subscription\_key,**  **)**  **prompt\_template = PromptTemplate(**  **input\_variables=["country", "no\_of\_paras", "language"],**  **template="""You are an expert in traditional cuisines.**  **You provide information about a specific dish from a specific country.**  **Avoid giving information about fictional places. If the country is fictional**  **or non-existent answer: I don't know.**  **Answer the question: What is the traditional cuisine of {country}?**  **Answer in {no\_of\_paras} short paras in {language}**  **""",**  **)**  **country = input("Enter the country:")**  **no\_of\_paras = int(input("Enter the number of paras"))**  **language = input("Enter the language:")**  **question = prompt\_template.format(**  **country=country, no\_of\_paras=no\_of\_paras, language=language**  **)**  **# Example usage**  **response = llm.invoke(question)**  **print(response.content)** |

### Partial()

In **LangChain**, the partial() method of a PromptTemplate is used to **pre-fill some variables** in the template, allowing you to reuse the prompt with fewer arguments later.

Example

* Imagine we have a prompt with multiple variables, but some of them stay constant across many uses. Instead of passing all variables every time, you can "lock in" some values using partial().

|  |  |
| --- | --- |
| from langchain.prompts import PromptTemplate  # Define a prompt with two variables  prompt = PromptTemplate.from\_template("Translate the following {language} sentence: {sentence}")  # Partially fill in the language  partial\_prompt **= prompt.partial(language="French")**  # Now you only need to provide 'sentence'  final\_prompt = partial\_prompt.format(sentence="Bonjour, comment ça va?")  print(final\_prompt) | **Output:**  Translate the following French sentence: Bonjour, comment ça va? |

## CHAT PROMPT TEMPLATE

* A **ChatPromptTemplate** is designed for **multi-turn chat messages** (system, user, assistant roles).
* Useful when you want structured conversations (like ChatGPT style).
* It maintains **role-based formatting**.

|  |
| --- |
| from langchain\_core.prompts import ChatPromptTemplate  template = ChatPromptTemplate([  **("system", "You are a helpful AI assistant."), 🡨 Tuples**  **("human", "{user\_input}")**  ])  prompt\_value = template.invoke({"user\_input": "What is the weather today?"}) |

The main purpose of ChatPromptTemplate is to:

* **Structure conversations**: We can define how the AI should behave (e.g., “You are a helpful assistant”) and how it should respond to user inputs.
* **Reuse and customize prompts**: Templates can be reused across different applications and customized with variables like {user\_input} or {name}.
* **Support multi-turn interactions**: It enables chaining multiple messages (system, human, AI) to simulate realistic dialogue flows

Example

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  **from langchain\_core.prompts import ChatPromptTemplate**  import os  from dotenv import load\_dotenv  from langchain\_core.output\_parsers import StrOutputParser  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  os.environ["LANGSMITH\_TRACING"] = os.getenv("LANGSMITH\_TRACING")  os.environ["LANGSMITH\_API\_KEY"] = os.getenv("LANGSMITH\_API\_KEY")  os.environ["LANGSMITH\_PROJECT"] = os.getenv("LANGSMITH\_PROJECT")  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,      temperature=0)  prompt = ChatPromptTemplate.from\_messages(      [          (              "system", "You are an expert AI Engineer. Provide me answers based on my questions."),          ("user", "{input}"),      ]  )  ###NOTE – We can get output we can use the following syntax too  # chain = chat\_prompt | llm | StrOutputParser()  # result = chain.invoke({"input": "What is LangChain?"})  # print(result.content)  ###NOTE – But it is recommended to use the some output Parser like StrOutputParser() to parse the AI Message  **chain = prompt | llm | StrOutputParser()**  **res = chain.invoke({"input": "Explain the LangChain in detail."})**  ### NOTE : The output message type is AIMessage  **print(type(res)) 🡪 AI Message**  **print(res.content)** |

### MESSAGEPLACE HOLDER IN CHATPROMPT TEMPLATE

## WHEN TO USE WHAT?

|  |  |  |
| --- | --- | --- |
| Aspect | Prompt Template | Chat Prompt Template |
| Interaction Type | Single-turn (one question → one answer) | Multi-turn (conversation with context) |
| Context Handling | No context preservation | Maintains conversation history and roles |
| Structure | One text block with placeholders | Role-based messages (system, user, assistant) |
| Use Case Examples | Summarization, email generation, translation | Chatbots, virtual assistants, RAG pipelines |
| When to Use | Task is static and predictable | Task requires dynamic context and dialogue flow |
| Example | "Write a {tone} email about {topic}" | system: "You are helpful", user: "{question}" |

## OUTPUT PARSER

* An **output parser** is a component responsible for **interpreting and structuring the raw output** from a language model (like GPT) into a more usable format.
* This is especially useful when we are expecting structured data (like JSON, lists, or specific fields) from a model that typically returns plain text.

|  |
| --- |
| Why It's Important?  Language models like GPT often return **free-form text**, which can be inconsistent or hard to process programmatically. Output parsers help in :   1. Extracting specific fields or values 2. Enforceing a schema (like JSON or Pydantic models) 3. Handle errors or malformed outputs 4. Make chaining between components more robust |
| Why Use Output Parsers?   * Language models often return unstructured text. If you're building a pipeline or application that needs structured responses (e.g., for chaining tasks, storing results, or triggering actions), output parsers help enforce consistency and reliability. |
| What It Does?  An output parser typically performs these steps:   * **Receives the raw output** from the model (usually a string). * **Validates and parses** it into a structured format (like a dictionary, list, or custom object). * **Returns** the parsed result for further use in your LangChain pipeline. |

### COMMON OUTPUT PARSERS IN LANGCHAIN

|  |  |
| --- | --- |
| Parser | Description |
| **StrOutputParser** | * Returns the raw string output. * Useful when we just want the plain text. |
| **CommaSeparatedListOutputParser** | * Parses output into a list of strings separated by commas. * Example: "apple, banana, cherry" → ["apple", "banana", "cherry"] |
| **JsonOutputParser** | * Parses output that is expected to be in JSON format. * Ensures the output is valid JSON and converts it to a Python dictionary. |
| **PydanticOutputParser** | * Parses output into a Pydantic model. * Great for enforcing schema and validation. |
| **StructuredOutputParser** | * Uses a schema (like JSON) to parse structured outputs. * Often used with function calling or OpenAI tools. |
| **OutputFixingParser** | * Wraps another parser and tries to fix malformed outputs using the model itself. * Useful when the model output is close to correct but needs minor adjustments. |

EXAMPLE

|  |  |
| --- | --- |
| from langchain\_openai import AzureChatOpenAI  **from langchain\_core.prompts import ChatPromptTemplate**  import os  from dotenv import load\_dotenv  from langchain\_core.output\_parsers import StrOutputParser  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  os.environ["LANGSMITH\_TRACING"] = os.getenv("LANGSMITH\_TRACING")  os.environ["LANGSMITH\_API\_KEY"] = os.getenv("LANGSMITH\_API\_KEY")  os.environ["LANGSMITH\_PROJECT"] = os.getenv("LANGSMITH\_PROJECT")  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,      temperature=0)  prompt = ChatPromptTemplate.from\_messages(      [          ( "system", "You are an expert AI Engineer. Provide me answers based on my questions."),      ("user", "{input}"),      ]  )  chain = prompt | llm | StrOutputParser()  res = chain.invoke({"input": "Explain the LangChain in detail."})  print(res) |  |

## CREATING SIMPLE GENAI APPLICATION

Application Objective

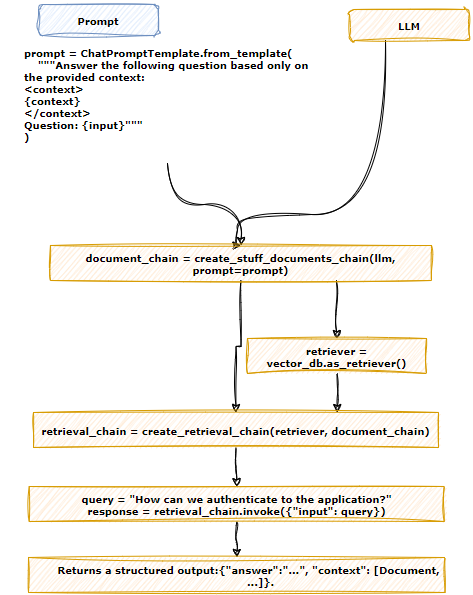
1. Scraping content from a website (e.g. <https://docs.smith.langchain.com/administration/tutorials/manage_spend>
   1. We need to beautifulsoup4 library to scrap the webpage
   2. INSTALL the Beauitifulsoup Library: **pip install beautifulsoup4**
2. Chunking the scraped text
3. Converting chunks into vector embeddings
4. Using a Language Model (LM) with prompt engineering to query or analyze the content

### RETRIEVAL CHAIN

A **Retrieval Chain** is a ready‑made pipeline that:

1. Takes your user input,
2. **Retrieves** the most relevant documents via a retriever, and
3. **Combines** those documents with the input in a prompt to the LLM, returning an **answer** (and the retrieved **context**).
4. It’s exposed as an LCEL Runnable and typically returns at least two keys: answer and context.

#### STEPS IN RETRIEVAL CHAIN



Step 1: User Question → Input

* The user provides a query, e.g., *“What is a Retrieval Chain?”*.
* This is passed as the input key in the chain.

Step 2: Retriever

* The retriever (e.g., vectorstore.as\_retriever()) takes the input and finds the most relevant documents from the knowledge base.
* Output: A List[Document] objects (each has page\_content + metadata).

Step 3: Combine-Docs Chain

|  |
| --- |
| llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,      temperature=0,  )  prompt = ChatPromptTemplate.from\_template(      """Answer the following question based only on the provided context:  <context>  **{context}**  </context>  Question: {input}"""  )  document\_chain = create\_stuff\_documents\_chain(llm, prompt=prompt) |

**create\_stuff\_documents\_chain**

* It formats the context and input into a single prompt.
* It then calls the LLM to generate an answer.
* Without this step, the LLM wouldn’t know how to use the retrieved docs effectively

Step 4: create retrieval chain

|  |
| --- |
| retriever = vector\_db.as\_retriever()  retrieval\_chain = **create\_retrieval\_chain**(retriever, document\_chain) |

* This is the orchestration layer which wires the retriever and the combine-docs chain together.
* It **assigns** the retrieved List[Document] to the key **context**, and forwards that context to the next step. i.e. it handles passing **input → retriever → context → LLM.**
* Returns a structured output:**{"answer": "...", "context": [Document, ...]}.**

Step 5: Final Output

The chain returns:

* answer: The LLM’s response based on the retrieved context.
* context: The actual documents used (for citations or debugging).

### CODE

* The details of the below is explained in RAG section

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  from langchain\_core.prompts import ChatPromptTemplate  import os  from dotenv import load\_dotenv  from langchain\_core.output\_parsers import StrOutputParser  from langchain\_community.document\_loaders import WebBaseLoader  from langchain\_text\_splitters import RecursiveCharacterTextSplitter  from langchain\_openai import AzureOpenAIEmbeddings  from langchain\_community.vectorstores import FAISS  from langchain.chains.combine\_documents import create\_stuff\_documents\_chain  from langchain.chains import create\_retrieval\_chain  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  os.environ["LANGSMITH\_TRACING"] = os.getenv("LANGSMITH\_TRACING")  os.environ["LANGSMITH\_API\_KEY"] = os.getenv("LANGSMITH\_API\_KEY")  os.environ["LANGSMITH\_PROJECT"] = os.getenv("LANGSMITH\_PROJECT")  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,      temperature=0)  # Initialize the Azure OpenAI Embeddings  embeddings\_client = AzureOpenAIEmbeddings(      azure\_deployment=embedding\_deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  loader = WebBaseLoader(web\_paths=("https://docs.langchain.com/langsmith/faq",))  docs = loader.load()  text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)  text\_chunks = text\_splitter.split\_documents(docs)  vector\_db = FAISS.from\_documents(text\_chunks, embeddings\_client)  query = "How can we authenticate to the application?"  prompt = ChatPromptTemplate.from\_template(      """Answer the following question based only on the provided context:  <context>  {context}  </context>  Question: {input}"""  )  document\_chain = create\_stuff\_documents\_chain(llm, prompt=prompt)  retriever = vector\_db.as\_retriever()  retrieval\_chain = create\_retrieval\_chain(retriever, document\_chain)  response = retrieval\_chain.invoke({"input": query})  print(f"Response: {response['answer']}") |

## CHAINS

### MESSAGE TYPES (langchain\_core.messages)

|  |  |
| --- | --- |
| INSTALL LANGCHAIN CORE | **pip install langchain\_core** |

* In LangChain, when interacting with a Language Model (LLM), we don’t just send raw text. Instead, we send a **structured list of messages** that helps the model understand:
  + **Who is speaking** (user, assistant, system, tool, etc.)
  + **What role each message plays**
  + **How to use the context** to generate accurate and consistent responses

|  |  |  |
| --- | --- | --- |
| **Type** | **Role** | **Used By** |
| **SystemMessage** | Sets the assistant's behavior/persona | Developer |
| **HumanMessage** | Represents user input | End user |
| **AIMessage** | Represents model output | Language model |
| **ToolMessage** | Returns results from external tools | Tools (e.g., search) |
| **FunctionMessage** | Returns results from function calls | Functions |

### CLASS HIERARCHY

|  |  |
| --- | --- |
| A diagram of text and words  AI-generated content may be incorrect. | BaseMessage  ├── HumanMessage (user input)  ├── AIMessage (LLM output)  ├── SystemMessage (instructions/context)  ├── ToolMessage (results returned from tools)  └── FunctionMessage (specialized outputs) |

### MESSAGE TYPES

A diagram of a medical procedure

AI-generated content may be incorrect.

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  import os  from dotenv import load\_dotenv  from langchain\_core.messages import SystemMessage, HumanMessage  from langchain\_core.output\_parsers import StrOutputParser  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  def get\_response(user\_input):      messages = [          SystemMessage(              content="You are an expert AI engineer. Provide me the answer for "          ),          HumanMessage(content=user\_input),      ]      return llm.invoke(messages)  user\_input = "What is Langchain?"  result = get\_response(user\_input)  print(type(result)) 🡨 AI Message  parser = StrOutputParser()  parsed\_result = parser.parse(result.content)  print(parsed\_result) |

Using Chaining

|  |
| --- |
| from unittest import result  from langchain\_openai import AzureChatOpenAI  from langchain\_core.prompts import ChatPromptTemplate  import os  from dotenv import load\_dotenv  from langchain\_core.messages import SystemMessage, HumanMessage  from langchain\_core.output\_parsers import StrOutputParser  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  prompt = ChatPromptTemplate.from\_messages(      [          ("system", "You are an expert AI engineer. Provide me the answer for "),          ("user", "{input}"),      ]  )  ################# CHAINING ########################  chain = prompt | llm | StrOutputParser()  user\_input = "What is Langchain?"  final\_result = chain.invoke({"input": user\_input})  print(final\_result)  print(type(final\_result)) |

#### HUMAN MESSAGE

* It represents a **message authored by the user (human)** in a conversation with a language model.
* It’s a lightweight **data structure**, not the text itself.

Why is it needed?

When we call an LLM, we don’t just send raw text. Instead, we send a **list of structured messages** so the model knows:

* who is speaking,
* what role each message has,
* how to use that context.

|  |
| --- |
| Example:  from langchain\_core.messages import HumanMessage, SystemMessage  messages = [  SystemMessage(content="You are a helpful assistant."),  HumanMessage(content="What are the symptoms of diabetes?")  ] |

How it works?

|  |  |
| --- | --- |
| Internally, HumanMessage inherits from BaseMessage.  It has attributes:   * content: the actual text or structured payload the user typed. * additional\_kwargs: optional dict for things like {"name": "Alice"}. * type: always "human" for this class. | msg = HumanMessage(content="Hello doctor!")  print(msg.type) # "human"  print(msg.content) # "Hello doctor!" |

#### AI MESSAGE

* Represents a message authored by the AI (LLM) in a conversation.
* It’s the counterpart of HumanMessage.

Why we need it?

* Let’s us distinguish “assistant’s reply” from user input and system instructions.
* Crucial in multi-turn conversations, where the model needs to “see its own past outputs” to stay consistent.

|  |
| --- |
| from langchain\_core.messages import AIMessage  msg = AIMessage(content="The main symptoms of diabetes include increased thirst, frequent urination, and fatigue.")  print(msg.type) # "ai"  print(msg.content) # The assistant's reply text  **Usage in a conversation**  conversation = [  HumanMessage(content="What are symptoms of diabetes?"),  AIMessage(content="Some common symptoms include thirst, urination, fatigue...")  ] |

#### SYSTEM MESSAGE

* Represents a system-level instruction that sets context for the AI.
* Think of it as the rules of the game or the persona of the assistant.

Why we need it

* Guides the model’s behavior without being “user text.”
* System messages often set tone, scope, or constraints (e.g., “You are a doctor assistant who always gives polite answers”).

|  |
| --- |
| from langchain\_core.messages import SystemMessage  msg = SystemMessage(content="You are a helpful healthcare assistant. Answer truthfully and cite sources.")  print(msg.type) # "system"  print(msg.content) # "You are a helpful healthcare assistant..."  **#Usage in a conversation**  conversation = [  SystemMessage(content="You are a medical research assistant."),  HumanMessage(content="What are the latest treatments for asthma?"),  AIMessage(content="Latest treatments include...") |

#### EXAMPLE: SIMPLE CONVERSATION

|  |
| --- |
| from langchain\_core.messages import HumanMessage, AIMessage, SystemMessage  # Define the conversation  messages = [  SystemMessage(content="You are a helpful assistant."),  HumanMessage(content="What are the symptoms of diabetes?"),  AIMessage(content="Common symptoms include increased thirst, frequent urination, and fatigue.")  ] |
| * **SystemMessage**: Sets the assistant’s tone and behavior. * **HumanMessage**: User asks a health-related question. * **AIMessage**: Assistant replies with relevant information. |

#### TOOL MESSAGE EXAMPLE

Suppose the assistant uses a **search tool** to answer a question.

|  |
| --- |
| from langchain\_core.messages import ToolMessage  tool\_result = ToolMessage(content="Search results: Diabetes symptoms include thirst, fatigue, and blurred vision.") |

This message is injected into the conversation so the model can use it to generate a final response.

#### FUNCTION MESSAGE EXAMPLE

* Used when the assistant calls a **function** (e.g., to calculate BMI).

|  |
| --- |
| from langchain\_core.messages import FunctionMessage  function\_output = FunctionMessage(content="BMI is 24.5, which is considered normal.") |

* This helps the model incorporate structured outputs from backend logic.

#### SUMMARY

Here’s a full conversation flow:

conversation = [

SystemMessage(content="You are a fitness assistant."),

HumanMessage(content="Calculate my BMI. I weigh 70kg and I'm 1.75m tall."),

FunctionMessage(content="BMI is 22.86, which is considered healthy."),

AIMessage(content="Your BMI is 22.86. That's within the healthy range!")

]

LangChain message types allow for **structured, multi-turn conversations** with clear roles:

* 🧑 **HumanMessage**: What the user says
* 🤖 **AIMessage**: What the assistant replies
* ⚙️ **SystemMessage**: How the assistant should behave
* 🔍 **ToolMessage**: What external tools return
* 🧮 **FunctionMessage**: What functions compute

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Message Type** | **Description** | **Purpose** | **Attributes** | **Example Usage** |
| **BaseMessage** | Abstract base class for all message types. | Provides a common structure for all messages. | Inherited by all other message types. | Used internally as a superclass. |
| **HumanMessage** | Message authored by the user. | Identifies user input in a structured format. | content, additional\_kwargs, type="human" | HumanMessage(content="Hello doctor!") |
| **AIMessage** | Message authored by the AI (LLM). | Distinguishes assistant replies from other messages. | content, type="ai" | AIMessage(content="Symptoms include thirst...") |
| **SystemMessage** | System-level instruction or context. | Sets tone, rules, or persona for the assistant. | content, type="system" | SystemMessage(content="You are a helpful assistant.") |
| **ToolMessage** | Message containing results returned from tools. | Communicates tool outputs back to the model. | Typically includes tool name and result payload. | Used when tools like search or code execution return data. |
| **FunctionMessage** | Message containing specialized outputs from function calls. | Enables structured function call responses. | Includes function name and output. | Used in advanced workflows involving function calling. |

### LANGCHAIN EXPRESSION LANGUAGE (LCEL) – SIMPLE DEFINITION

* **LCEL** is a way to build AI workflows in LangChain using a clean and modular style.
* It lets us connect components like prompts, models, and output parsers using a simple syntax, making the code easier to read and maintain.

#### EXAMPLE

|  |  |
| --- | --- |
| **import os**  **from langchain\_openai import AzureChatOpenAI**  **from langchain.prompts import PromptTemplate**  **from dotenv import load\_dotenv**  **from langchain\_core.output\_parsers import StrOutputParser**  **load\_dotenv()**  **# Set your Azure OpenAI credentials**  **subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]**  **endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]**  **api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]**  **deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]**  **# Initialize the Azure OpenAI LLM**  **llm = AzureChatOpenAI(**  **azure\_deployment=deployment,**  **api\_version=api\_version,**  **azure\_endpoint=endpoint,**  **api\_key=subscription\_key,**  **)**  **prompt\_template = PromptTemplate(**  **input\_variables=["language", "text"],**  **template="Translate the following text to {language}: {text}",**  **)**  **text = input("Enter the text to translate: ")**  **language = input("Enter the target language: ")**  **parser = StrOutputParser()**  **# Example usage**  **chain = prompt\_template | llm | parser**  **response = chain.invoke({"text": text, "language": language})**  **print(response)** | **OUTPUT**     * **Note : The chain.invoke() takes dictionary of prompt parameters** |

Explanation

* prompt formats the input.
* llm generates the response.
* parser extracts the final output.
* The | operator connects them in sequence.

### SEQUENTIAL PROMPTS

* A **sequential prompt** in LangChain refers to a **workflow where multiple prompts are executed in a specific order**, **with the output of one prompt serving as the input to the next**.
* This is implemented using **Sequential Chains**, which are ideal for tasks that require **step-by-step reasoning or multi-stage processing**.

Example

|  |
| --- |
| **import os**  **from langchain\_openai import AzureChatOpenAI**  **from langchain.prompts import PromptTemplate**  **from dotenv import load\_dotenv**  **from langchain\_core.output\_parsers import StrOutputParser**  **load\_dotenv()**  **# Set your Azure OpenAI credentials**  **subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]**  **endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]**  **api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]**  **deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]**  **# Initialize the Azure OpenAI LLM**  **llm = AzureChatOpenAI(**  **azure\_deployment=deployment,**  **api\_version=api\_version,**  **azure\_endpoint=endpoint,**  **api\_key=subscription\_key,**  **)**  **topic\_prompt = PromptTemplate(**  **input\_variables=["topic"], template="Create a speech on following topic: {topic}"**  **)**  **speech\_prompt = PromptTemplate(**  **input\_variables=["no\_of\_words"],**  **template="keep the speech under {no\_of\_words} words. "**  **"Make it engaging and informative. ",**  **)**  **first\_chain = topic\_prompt | llm | StrOutputParser()**  **second\_chain = speech\_prompt | llm | StrOutputParser()**  **final\_chain = first\_chain | second\_chain**  **response = final\_chain.invoke({"topic": "Artificial Intelligence", "no\_of\_words": 1000})**  **print(response)** |

### SIMPLE SEQUENTIAL CHAIN

1. **Linear flow**: Each step takes the output of the previous step as its input.
2. **No variable mapping**: You don’t specify input/output keys.
3. **Best for quick prototyping** with simple chains.

### REGULAR SEQUENTIAL CHAIN

## LANGSERVE

* **LangServe** is a **deployment framework for LangChain applications**.
* It allows us to take any LangChain **Runnable** (chains, agents, retrievers, etc.) and **expose it as a REST API** using FastAPI, with minimal boilerplate.

### DEPLOYMENT

|  |  |
| --- | --- |
| INSTALL REQUIRED LIBRARIES | pip install langserve  pip install fastapi  pip install uvucorn |

## CHAT HISTORY

|  |  |
| --- | --- |
| INSTALL REQUIRED LIBRARIES | pip install langchain\_community |

### MAINTAINING HISTORY

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  import os  from dotenv import load\_dotenv  from langchain\_community.chat\_message\_histories import ChatMessageHistory  from langchain\_core.chat\_history import BaseChatMessageHistory  from langchain\_core.runnables.history import RunnableWithMessageHistory  from langchain\_core.messages import HumanMessage  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  store = {}  def get\_session\_history(session\_id: str) -> BaseChatMessageHistory:      if session\_id not in store:          store[session\_id] = ChatMessageHistory()      return store[session\_id]  with\_message\_history = RunnableWithMessageHistory(llm, get\_session\_history)  config = {"configurable": {"session\_id": "chat1"}}  response = with\_message\_history.invoke(      [HumanMessage(content="Hi , My name is Krish and I am a Chief AI Engineer")],      config=config,  )  print(response.content)  response = with\_message\_history.invoke(      [HumanMessage(content="Can you please tell me my name?")], config=config  )  print(response.content)  #### NEW CONFIG######  config2 = {"configurable": {"session\_id": "chat2"}}  response2 = with\_message\_history.invoke(      [HumanMessage(content="Can you please tell me my name?")], config=config2  )  print(response2.content) |
| OUTPUT  Hello Krish! It's great to meet you. As a Chief AI Engineer, you must have a wealth of knowledge and experience in the field. What specific areas of AI are you currently focused on, or is there something particular you'd like to discuss today?  Your name is Krish. How can I assist you today?  I'm sorry, but I don't have access to personal information about you unless you share it with me. How can I assist you today? |

It shows how to **add per‑session chat memory** to an Azure OpenAI chat model using LangChain’s **RunnableWithMessageHistory**, so the model can remember what a user said earlier in the same session and use it in follow‑ups.

#### EXPLANATION

Create an in‑memory session store

**store = {}**

* A plain Python dict used to **hold conversation histories** *per* session\_id.
* For production, we can use Redis/DB-backed history.

Session history factory

|  |
| --- |
| **def get\_session\_history(session\_id: str) 🡪 BaseChatMessageHistory:**  **if session\_id not in store:**  **store[session\_id] = ChatMessageHistory()**  **return store[session\_id]** |

* Given a session\_id, returns a ChatMessageHistory object (creating it if absent).
* ChatMessageHistory stores a sequence of HumanMessage/AIMessage pairs.

Wrap the model with message history

**with\_message\_history = RunnableWithMessageHistory(llm, get\_session\_history)**

* RunnableWithMessageHistory wraps llm so that **on every call** it:
  + **Loads the chat history** for the provided session\_id.
  + **Prepends** that history to the new input messages.
  + Invokes the underlying llm.
  + **Appends** the new human + AI messages back into the history for that session.

**Caller (.invoke with session\_id)**

**└─▶ RunnableWithMessageHistory**

**├─ get\_session\_history(session\_id) → ChatMessageHistory**

**├─ prepend past messages + new HumanMessage**

**├─ call AzureChatOpenAI**

**└─ append new Human + AI messages back to history**

Configure a session (chat1)

**config = {"configurable": {"session\_id": "chat1"}}**

* Every call to the runnable must pass a **config** telling it which session to read/write.
* Here, session\_id="chat1" identifies one user/session.

First turn in session chat1

|  |
| --- |
| **response = with\_message\_history.invoke(**  **[HumanMessage(content="Hi , My name is Krish and I am a Chief AI Engineer")],**  **config=config,**  **)**  **print(response.content)** |

* RunnableWithMessageHistory looks up chat1 → empty history initially.
* It sends **only this HumanMessage** to the model (since there’s no prior history).
* The model replies (e.g., “Nice to meet you, Krish!”).
* The wrapper **stores both** the human input and the AI reply in store["chat1"].
* **Result:** History for chat1 now contains:
  + HumanMessage("Hi, my name is Krish…")
  + AIMessage("Nice to meet you…")

Second turn in session chat1

|  |
| --- |
| **response = with\_message\_history.invoke(**  **[HumanMessage(content="Can you please tell me my name?")],**  **config=config**  **)**  **print(response.content)** |

* The wrapper loads history for chat1 (which now has 2 messages).
* It **prepends** that history to the new message and calls the model with the **full** conversation.
* The model can now infer: *“Your name is Krish.”*
* Wrapper appends the new question and AI’s answer to the history.
* **Expected output:** Something like *“Your name is Krish.”*

New, independent session (chat2)

|  |
| --- |
| **config2 = {"configurable": {"session\_id": "chat2"}}**  **response2 = with\_message\_history.invoke(**  **[HumanMessage(content="Can you please tell me my name?")], config=config2**  **)**  **print(response2.content)** |

* New session ID means **no prior history**.
* The model only sees the single question: “Can you please tell me my name?”
* Without context, a helpful model usually says it doesn’t know (or asks for your name).
* **Expected output:** Something like *“I don’t know your name yet.”*

## EMBEDDING MODELS

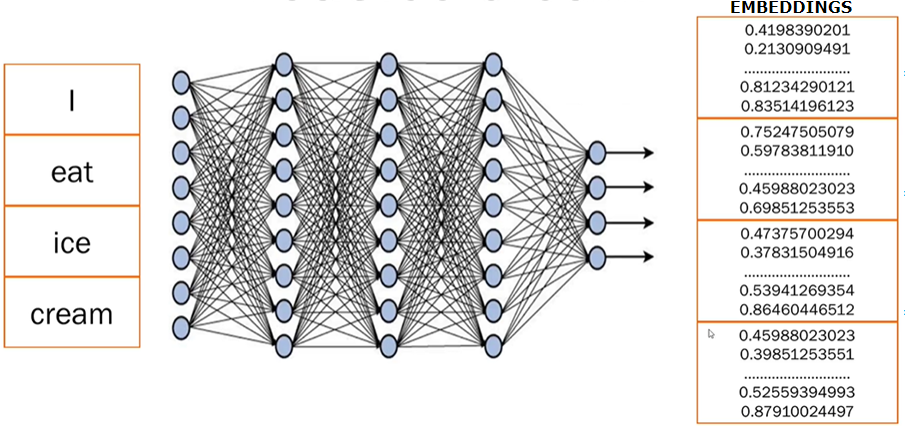
### EMBEDDINGS

* Embeddings are numerical representations of text(called vectors)—a way to convert words into numbers so machines can understand and process them.
* These vectors **capture meaning and relationships** between words or sentences.

How Embeddings Work?

* Embeddings encode:
  + Frequency of word usage in different contexts.
  + Relationships between words and sentences.
  + These vectors allow machines to compare meanings effectively.

### HOW EMBEDDINGS ARE GENERATED



1. **STEP 1:** **BREAKING TEXT INTO TOKENS**

* Sentences are broken into smaller pieces or tokens (e.g., splitting "I eat ice cream" into 4 tokens—"I", "eat", "ice", "cream").

2. **STEP 2: NEURAL NETWORK PROCESSING**:

* Trained **transformer models** analyze the text, generate embeddings, and capture meaning, context, and relations between tokens.

1. **STEP 3: NUMERICAL EMBEDDINGS:**

* Each token is converted into numerical data (random numbers).
* These numbers represent embeddings, storing all learned information about the word or sentence.
* Only the transformer model understands what these embeddings mean based on its training.

Understanding Semantic Relationships

|  |  |
| --- | --- |
|  | Considering a simple example   * Here is an example of three words **happy, joyful, and glad**, which are usually used in a similar context. * If we assign numeric codes (like 1, 2, 3) to these words, any algorithm or machine, by looking at these numeric codes, can easily figure out the relationship between these words. * 1 and 2 are much closer than 1 and 3, so it will know that happy and joyful are much more frequently used, are closely related than happy and glad. * *But in real time, these numeric codes or embeddings are much more complex.* |

Example of Sentence Embeddings

Sentences like:

***"The sun is shining brightly in clear blue sky."***

***"It's a beautiful day with clear sunny sky."***

Have similar meanings, and their embeddings will be close together in vector space.

Applications of Embeddings

* 1. Document similarity: Find texts with similar meanings.
  2. Search engines: Enable semantic search beyond keyword matching.
  3. Recommendation systems: Suggest similar items based on meaning.
  4. Language translation: Match meaning across languages.

### GENERATING EMBEDDINGS USING LANGCHAIN

|  |
| --- |
| **from langchain\_openai import AzureOpenAIEmbeddings**  **import os**  **from dotenv import load\_dotenv**  **load\_dotenv()**  **# Set your Azure OpenAI credentials**  **subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]**  **endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]**  **api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]**  **deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]**  **# Initialize the Azure OpenAI Embeddings**  **embeddings = AzureOpenAIEmbeddings(**  **azure\_deployment=deployment,**  **api\_version=api\_version,**  **azure\_endpoint=endpoint,**  **api\_key=subscription\_key,**  **)**  **# Example usage**  **try:**  **text = input("Enter the text to embed: ")**  **response = embeddings.embed\_query(text)**  **print(f"Embedding vector length: {len(response)}")**  **print(f"First 5 dimensions: {response[:5]}")**  **except Exception as e:**  **print(f"Error generating embeddings: {e}")** |
| **.env**  **AZURE\_OPENAI\_API\_KEY=2H8z7i6cZxgDx8X7WksANiaTZQveZBoqYOsHBjzsXB1ObMv7MyqKJQQJ99BHACYeBjFXJ3w3AAABACOGs6Zg**  **AZURE\_OPENAI\_ENDPOINT=https://ai-dojo-open-ai.openai.azure.com/**  **AZURE\_OPENAI\_API\_VERSION=2024-12-01-preview**  **AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING=text-embedding-ada-002** |
| **OUTPUT** |

### SIMILARITY FINDER

* A similarity finder is a tool that compares two pieces of text—words, phrases, or sentences—and tells us how similar they are in meaning.
* It uses embeddings, which are numerical representations of text, to do this.

### HOW IT WORKS?

1. Convert Text to Embeddings

* + Each input text is passed through an embedding model (like OpenAI’s text-embedding-3-large).
  + The model returns a vector (a list of numbers) that captures the semantic meaning of the text.

2. Compare the Vectors

* The two vectors are compared using cosine similarity, which measures the angle between them.
* This is done using a mathematical function like np.dot() from the NumPy library.

3. Interpret the Similarity Score

The result is a number between 0 and 1:

* 1 means the texts are very similar.
* 0 means they are completely different.

Example

|  |  |  |
| --- | --- | --- |
| Text 1 | Text 2 | Similarity Score |
| "dog" | "cat" | 0.76 (76%) |
| "cat" | "kitten" | 0.88 (88%) |
| "You are great" | "You are bad" | 0.87 (87%) |
| "You are awesome" | "You are good" | 0.91 (91%) |

Why It’s Useful

* Semantic Search: Find documents or answers that match a query in meaning.
* Recommendation Systems: Suggest similar items based on user input.
* Plagiarism Detection: Identify texts that are similar meanings.
* Chatbots & Assistants: Retrieve relevant responses or context.

#### EXAMPLE

|  |  |
| --- | --- |
| INSTALL NUMPY | **pip install numpy** |

* Similarity finder that will figure out how similar given two words or sentences are using their embedding values.

|  |  |  |  |
| --- | --- | --- | --- |
| from langchain\_openai import AzureOpenAIEmbeddings  import os  from dotenv import load\_dotenv  import numpy as np  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  # Initialize the Azure OpenAI Embeddings  embeddings = AzureOpenAIEmbeddings(  azure\_deployment=deployment,  api\_version=api\_version,  azure\_endpoint=endpoint,  api\_key=subscription\_key,  )  # Example usage  try:  **text1 = input("Enter the text1: ")**  **text2 = input("Enter the text2: ")**  **# Get the embedding of the text**  **response1 = embeddings.embed\_query(text1)**  **response2 = embeddings.embed\_query(text2)**  **# Calculate the cosine similarity**  **similarity = np.dot(response1, response2)**  **print(similarity)**  except Exception as e:  print(f"Error generating embeddings: {e}") | | | |
| **OUTPUT: The output will be the similarity score in between 0 and 1. The higher the score, the closer those texts are.** | | | |
|  |  |  |  |

## VECTORS

A **vector** is a list of numbers that represents something—like a word, an image, or even a sentence—in a way that a computer can understand and work with.

|  |  |
| --- | --- |
| Simple Analogy | |
| Imagine we want to describe a fruit (say, an **apple**) using numbers:   * Sweetness: 8 * Crunchiness: 7 * Juiciness: 6   We could represent the apple as a vector: [8, 7, 6]  Now, a **banana** might be:[9, 3, 7]  These vectors help a computer compare fruits based on their features. | A diagram of a diagram  AI-generated content may be incorrect. |

In Language Models

When we talk about **words**, we turn them into vectors using **embeddings**. For example:

* “cat” → [0.12, -0.45, 0.88, ..., 0.03]
* “dog” → [0.10, -0.40, 0.85, ..., 0.05]

These vectors are **high-dimensional** (often 300 to 1,000+ numbers long) and capture the **meaning** of the word based on how it’s used in language.

### WHY ARE VECTORS USEFUL?

They allow computers to:

* **Compare** things (e.g., how similar two words or images are)
* **Search** by meaning (semantic search)
* **Cluster** similar items together
* **Feed data into machine learning models**

### VECTOR DATABASE

A diagram of a data processing process

AI-generated content may be incorrect.

* A **vector database** is a special kind of database designed to store and search **vectors (***which are just lists of numbers that represent things like text, images, or audio in a way that computers can understand*.)
* Primarily used for storing embeddings that represent complex data like images, text and audio in a form that machine can understand and process

Why Vectors?

* When you use **embeddings** (like we discussed earlier), we turn data (like the word *“cat”*) into a vector, such as:

**[0.12, -0.45, 0.88, ..., 0.03]**

* These vectors capture **meaning** and **context**. But once we have millions of them, we need a smart way to **store** and **search for** them efficiently. That’s where vector databases come in.

What Does a Vector Database Do?

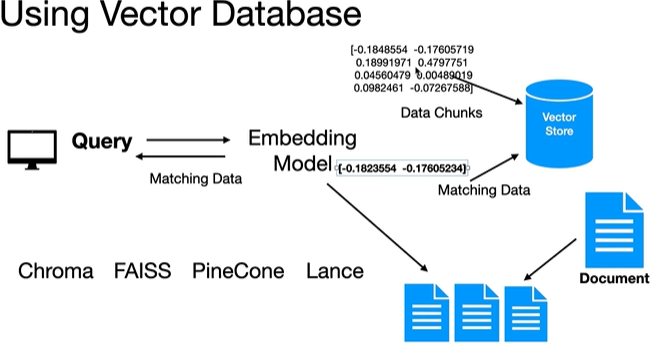
It helps to:

* **Store** millions or billions of embeddings
* **Search** for the most similar vectors (e.g., “find texts similar to this one”)
* **Rank** results by similarity (using distance metrics like cosine similarity)

Real-World Example

Let’s say we run a **document search engine**:

1. We convert all the documents into vectors using an LLM.
2. We store those vectors in a vector database (like **Pinecone**, **Weaviate**, **FAISS**, or **Milvus**).
3. When a user asks a question, we:
   * Convert the question into a vector
   * Search the database for the **most similar document vectors**
   * Return the most relevant documents



* A vector store allows us to store and retrieve data using high dimensional vectors.
* This is very useful to build similarity search applications.

#### STEPS

* Step 1: Chunking
  + The process to store and retrieve data in a vector store involves taking that data and breaking it down into small chunks.
* Step 2: Calculate Embeddings
  + Once we have the chunks, we'll calculate the embeddings for each of these chunks using an embedding large language model.
* Step 3: Store in Vector Database
  + Once we have the embedding, **we'll use this embedding as metadata will take the chunk of the data and then send it to the vector store.**
  + The vector store will create an index internally using these embeddings, and it will store the data chunk under that index.(So the vector store is aware of the high dimensional vector for that data chunk.)

When Queried By User

* + When the user later comes up with a query or a search will take that search query, calculate the embedding for it, and will hand over that embedding to the vector store.
  + The vector store will use that embedding, find all the similar embeddings, and return back the data chunks for those embeddings, so all the matching data will be returned back by the vector store, which we can then send back to the user.

#### TYPES OF VECTOR STORE

* **FAISS**
* **Chroma**
* **Pinecone**
* **Weaviate**
* **Milvus**
* **Qdrant**
* **Redis**
* **ElasticSearch**

#### EXAMPLE - LANGCHAIN

**DATA- joblist.txt**

|  |
| --- |
| 1. **Software Engineer at TechCorp - Responsibilities include developing and maintaining software applications, collaborating with cross-functional teams, and ensuring code quality. Requires proficiency in Java, Python, and SQL.**  **2. Data Scientist at DataMinds - Duties involve analyzing large datasets, building predictive models, and presenting insights to stakeholders. Requires expertise in Python, R, and machine learning.**  **3. Digital Marketing Specialist at MarketGurus - Role includes creating and managing online marketing campaigns, analyzing web traffic, and optimizing SEO. Requires experience with Google Analytics, SEM, and content creation.**  **4. Project Manager at BuildIt - Responsibilities include overseeing construction projects, managing budgets, and coordinating with contractors. Requires strong leadership skills and knowledge of project management software.**  **5. Graphic Designer at CreativeWorks - Role involves designing marketing materials, collaborating with the creative team, and adhering to brand guidelines. Requires proficiency in Adobe Creative Suite and a strong portfolio.**  **6. Financial Analyst at FinExperts - Duties include analyzing financial data, preparing reports, and advising on investment decisions. Requires strong analytical skills and experience with financial modeling.**  **7. Human Resources Manager at PeopleFirst - Responsibilities include recruiting, onboarding, and managing employee relations. Requires excellent communication skills and knowledge of HR software.**  **8. Cybersecurity Specialist at SecureNet - Role involves protecting the company's IT infrastructure, monitoring for security breaches, and implementing security protocols. Requires expertise in network security and experience with security tools.**  **9. Sales Manager at RetailStars - Duties include managing the sales team, developing sales strategies, and achieving sales targets. Requires strong leadership skills and experience in retail sales.**  **10. Content Writer at WordSmiths - Responsibilities include creating engaging content for blogs, social media, and websites. Requires excellent writing skills and a creative mindset.** |

In this example

* We will use job\_listings.txt file which has several job positions like software engineer, data scientist, digital marketing project, manager, graphic designer, cyber security specialist, content writer and more.
* We will use this data(**job\_listing.txt**)
  1. Break in into chunks
  2. Calculate the embeddings
  3. Load it up into the vector store

|  |  |
| --- | --- |
| **INSTALL LANGCHAIN CHROMA(Vector Store)** | **pip install langchain\_chroma** |

|  |
| --- |
| from langchain\_openai import AzureOpenAIEmbeddings  **from langchain\_community.document\_loaders import TextLoader**  **from langchain\_text\_splitters import RecursiveCharacterTextSplitter**  **from langchain\_chroma import Chroma**  import os  from dotenv import load\_dotenv  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  # Initialize the Azure OpenAI Embeddings  llm = AzureOpenAIEmbeddings(  azure\_deployment=deployment,  api\_version=api\_version,  azure\_endpoint=endpoint,  api\_key=subscription\_key,  )  **# Load the job listings document from a text file**  **document = TextLoader("job\_listings.txt").load()**  **# Create a text splitter to break the document into smaller chunks**  **text\_spiltter = RecursiveCharacterTextSplitter(chunk\_size=200, chunk\_overlap=10)**  **# Split the loaded document into manageable chunks for processing**  **chunks = text\_spiltter.split\_documents(document)**  **# Create a Chroma vector database from the document chunks using embeddings**  **vector\_db = Chroma.from\_documents(chunks, llm)**  **# Get user input for the search query**  **query = input("Enter the query: ")**  **# Generate embedding vector for the user's query**  **embedding\_of\_query = llm.embed\_query(query)**  **# Search the vector database for documents similar to the query embedding**  **docs = vector\_db.similarity\_search\_by\_vector(embedding\_of\_query)**  **# Iterate through the retrieved similar documents**  **for doc in docs:**  **# Print the content of each similar document**  **print(doc.page\_content)** |
| **OUTPUT** |
|  |

#### QUERY THE VECTOR DB USING RETRIEVER

* Another easy way to query the vector database is to use a retriever instance.

**retriever = vector\_db.as\_retriever()**

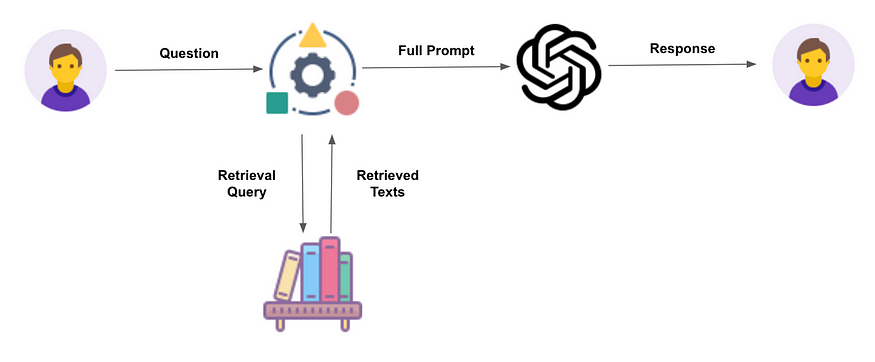
**The retriever instance is capable of taking plain text, calculate the embedding for it, then query the vector database with that embedded text and retrieve the data as well.**

|  |
| --- |
| from langchain\_openai import AzureOpenAIEmbeddings  **from langchain\_community.document\_loaders import TextLoader**  **from langchain\_text\_splitters import RecursiveCharacterTextSplitter**  **from langchain\_chroma import Chroma**  import os  from dotenv import load\_dotenv  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  # Initialize the Azure OpenAI Embeddings  llm = AzureOpenAIEmbeddings(  azure\_deployment=deployment,  api\_version=api\_version,  azure\_endpoint=endpoint,  api\_key=subscription\_key,  )  **# Load the job listings document from a text file**  **document = TextLoader("job\_listings.txt").load()**  **# Create a text splitter to break the document into smaller chunks**  **text\_spiltter = RecursiveCharacterTextSplitter(chunk\_size=200, chunk\_overlap=10)**  **# Split the loaded document into manageable chunks for processing**  **chunks = text\_spiltter.split\_documents(document)**  **# Create a Chroma vector database from the document chunks using embeddings**  **vector\_db = Chroma.from\_documents(chunks, llm)**  **# Convert the vector database to a retriever interface for easier querying**  **retriever = vector\_db.as\_retriever()**  **# Get user input for the search query**  **query = input("Enter the query: ")**  **# Use the retriever to find relevant documents based on the query**  **docs = retriever.invoke(query)**  **# Iterate through the retrieved relevant documents**  **for doc in docs:**  **# Print the content of each relevant document**  **print(doc.page\_content)** |

# RAG (RETRIEVER AUGUMENTED GENERATION)

A white background with blue text

AI-generated content may be incorrect.



* **Retriever**: Fetches relevant information from a knowledge base.
* **Augmented**: Adds value by combining retrieved data with generative capabilities.
* **Generation**: Produces a response using a language model (e.g., ChatGPT, Gemini).

## RAG WORKFLOW

|  |
| --- |
| Step 1: Ask a Question –   * When we ask a question. The question goes into a knowledge base (like a smart library). * The knowledge source can be Databases, Articles or Websites * This knowledge base is often a vector database, which stores information in a format that helps find similar meanings.   Step 2: It Finds Relevant Text   * The system retrieves the most relevant documents or pieces of information that match the question. In the vector Database. This is called the retrieval step.   Step 3: It Builds a Full Input   * The retrieved information is combined with the original question. * This combination becomes a full prompt or input for the next step.   Step 4: AI Generates the Answer   * **Full prompt is sent to a Language Model (like ChatGPT or Gemini).** * **Note**: Retrieved data + original query = **prompt** for the **Language Model (LM)**. * The model reads both questions and the retrieved info and then generates a smart answer. * The response is based on internal data (from the knowledge base) and the intelligence of the AI model. |

## WHY RAG?

**RAG is important due to the following limitations of LLMs**

1. KNOWLEDGE CUT-OFF DATE
   1. The LLM model will have information up to their training cut-off date and lack of information beyond that point.
   2. Hence LLMs **can’t access real-time updates** or dynamic data, and they may miss recent changes in the organization or product.
2. LACK OF ACCESS OF ENTERPRISE DATA
   1. They lack access of enterprise specific data unless they are fine-tuned for customized for that enterprise

|  |
| --- |
| * Let’s say in banks or an enterprise company, the data will be sitting privately within companies private infrastructure or within companies’ network. * In this use-case - We don't want ChatGPT or any other LLMs to have information about private data. So that is where vector databases come into picture where the questions retrievals will happen from the company’s internal source. * Then – we are just using the capabilities of LLM to prepare a nice prompt to give a proper reply. |

Enterprise Use Cases

* **Customer support**: Pulls product details from internal DBs.
* **Educational tools**: Provides precise, sourced answers.
* **Banking/Enterprise**: Keeps sensitive data internal while leveraging LMs.

## BENEFITS

Access to Private/Internal Data

* RAG allows LLMs to use your organization’s internal documents (e.g., product manuals, policies, reports) that are not part of public training data.
* This makes responses more relevant and accurate for enterprise use.

Reduces Hallucinations

* LLMs sometimes make up answers when they don’t have enough information.
* RAG reduces this by grounding responses in real, retrieved documents.

Improves Contextual Accuracy

* RAG retrieves context-specific information before generating a response.
* This ensures the answer is tailored to the user’s query and environment.

Keeps Data Secure

* Sensitive data stays within private infrastructure.
* The LLM only sees the retrieved content, not the entire database—helping with data privacy and compliance.

Dynamic and Up-to-Date Responses

* Instead of relying on static training data, RAG can pull real-time or recently updated documents.
* This makes it ideal for fast-changing domains like tech support or policy updates.

## RAG ARCHITECTURE

A diagram of a computer system

AI-generated content may be incorrect.Step 1: Prepare Your Data

* Collect documents, images, videos, etc.
* Send them to an **Embedder**, which converts them into a format (vectors) that computers can search easily.

Step 2: Store the Data

* The Embedder sends these vectors to a **Vector Storage and Retrieval Engine** (like a smart library).

Step 3: User Asks a Question

* A person types a question into a **chat interface**.

Step 4: Process the Question

* The question goes to a **User Query module**, then to the **Embedder** to be converted into a vector (just like the data was).

Step 5: Search for Matching Info

* The query vector is sent to the **Vector Storage and Retrieval Engine**.
* It searches for the most relevant information based on meaning (semantic search).

Step 6: Generate the Answer

* The retrieved information is sent to a **Large Language Model** (like ChatGPT).
* The model uses both the question and the retrieved info to create a smart, accurate response.

Step 7: Show the Answer

* The response is sent back to the **chat interface**, where the user sees the final answer.

## RAG PROCESS- BUILDING RAG PIPELINE

A diagram of a step 3

AI-generated content may be incorrect.

A diagram of a computer

AI-generated content may be incorrect.

### STEP 1: DATA INGESTION (USING DOCUMENT LOADERS)

#### TEXT LOADER

|  |
| --- |
| **from langchain\_community.document\_loaders import TextLoader**  **import os**  **# Get the directory of the current script**  **current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))**  **speech\_file\_path = os.path.join(current\_dir, "speech.txt")**  **text\_document = TextLoader(speech\_file\_path).load()**  **print(text\_document)** |

#### PDF LOADER

|  |  |
| --- | --- |
| INSTALL pypdf | **pip install pypdf** |

|  |
| --- |
| **from langchain\_community.document\_loaders import PyPDFLoader**  **import os**  **# Get the directory of the current script**  **current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))**  **pdf\_file\_path = os.path.join(current\_dir, "attention.pdf")**  **pdf\_document = PyPDFLoader(pdf\_file\_path).load()**  **print(pdf\_document) 🡨 List of document with some meta data** |

#### WEB BASED LOADER

* Read the content of the web pages(Web Scraping)
* The “web\_paths” parameter will allow as to configure multiple URLs
* We can use beautiful Soup package to parse the document further

|  |
| --- |
| **web\_loader = WebBaseLoader(**  **web\_paths=("https://www.aarpmedicareplans.com/",),**  **header\_template={**  **"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36"**  **},**  **)**  **web\_document = web\_loader.load()**  **print(web\_document)** |

#### EXAMPLE -LOAD DATA – PDF

|  |
| --- |
| **from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI**  **from langchain\_community.document\_loaders import PyMuPDFLoader**  **import os**  **from dotenv import load\_dotenv**  **load\_dotenv()**  **# Set your Azure OpenAI credentials (single resource for both models)**  **subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]**  **endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]**  **api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]**  **embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]**  **gpt\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Your GPT model deployment**  **# Initialize the Azure OpenAI Chat model for text generation**  **llm = AzureChatOpenAI(**  **azure\_deployment=gpt\_deployment,**  **api\_version=api\_version,**  **azure\_endpoint=endpoint,**  **api\_key=subscription\_key,**  **)**  **# Initialize the Azure OpenAI Embeddings for vector creation**  **embeddings = AzureOpenAIEmbeddings(**  **azure\_deployment=embedding\_deployment,**  **api\_version=api\_version,**  **azure\_endpoint=endpoint,**  **api\_key=subscription\_key,**  **)**  **def load\_pdf\_with\_langchain(pdf\_path):**  **loader = PyMuPDFLoader(pdf\_path)**  **document = loader.load()**  **return document**  **docs = load\_pdf\_with\_langchain("rag/academic\_research\_data.pdf")**  **print("\n Sample Extracted Content:")**  **for i, doc in enumerate(docs[:2]):**  **print(f"\n--- Chunk {i + 1} ---")**  **print(doc.page\_content[:500])  # Show first 500 characters**  **print("Metadata:", doc.metadata)** |
| OUTPUT: The Pdf data is read by the loaded page by page |
| * The loader extracted the text **page by page**, and wrapped each into a Document object with metadata (like page number and file name). * This gives us a **basic form of chunking** — each page is its own chunk . For better performance in retrieval(from Vector DB), we’ll need to apply **smarter chunking strategies** next — like:   + Splitting by characters or sentences   + Ensuring chunks have overlap   + Preserving context between chunks |

### STEP 2- DATA CHUNKING

* Once we extract text from a document like a PDF, we usually don’t feed that entire text to a language model at once. It’s too large, and most of it might not be relevant to a specific question. That’s where **data chunking** comes in.
* Breaking documents (from step 1) into smaller, meaningful pieces – a process called **chunking**.

|  |
| --- |
| **Chunking** is the process of breaking down a large block of text into smaller, manageable parts (called "chunks") — so they:   * **Fit within a model’s token limits** * **Make sense on their own** * **Can be searched and retrieved efficiently** |

* Data chunking is important when dealing with huge data like large clinical trial PDFs, which can often exceed 100 pages. Large Language Models (LLMs) are not able to process such extensive content in one go, and even if they could, retrieving relevant information would be inefficient.
* By splitting the documents into chunks, such as paragraphs or sections, we can retain sufficient context without overwhelming the system.
* In LangChain, utilities like `**RecursiveCharacterTextSplitter**` or `**MarkdownHeaderTextSplitter**` can be used depending on the structure of the data.
* These tools allow us to split text based on logical separators like headings or paragraphs, define chunk sizes (for example, 500 characters), and maintain overlaps (such as 100 characters) to ensure context is preserved across chunk boundaries.

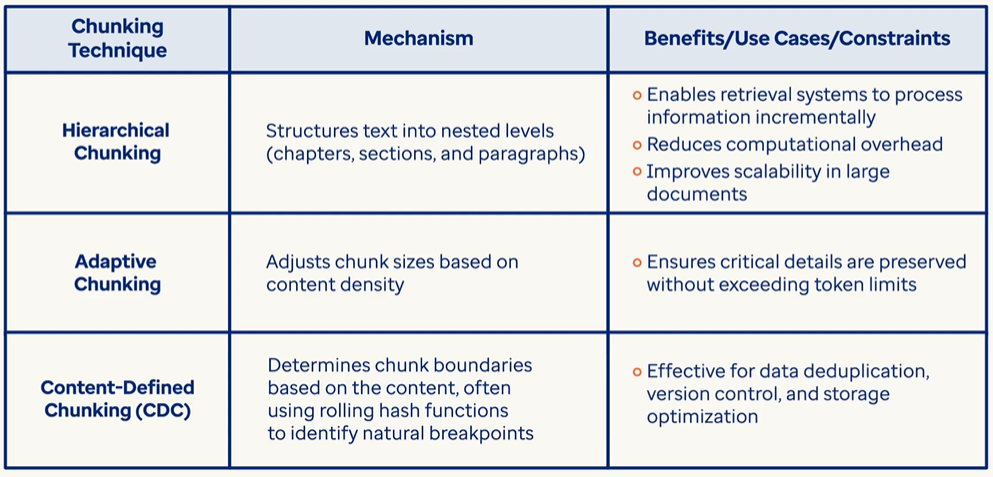
|  |
| --- |
| **Example** |
| **INSTALLLING TEXT SPLITTER: pip install langchain-text-splitters** |
| **document = TextLoader("rag/product-data.txt").load()**  **text\_spiltter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)**  **chunks = text\_spiltter.split\_documents(document)** |

#### DIFFERENT CHUNKING METHODS

A diagram of a mechanism

AI-generated content may be incorrect.

CHUNKING TECHNIQUES FOR COMPLEX AND VARAIBLE DATA STRUCTURES



##### FIXED-SIZE CHUNKING

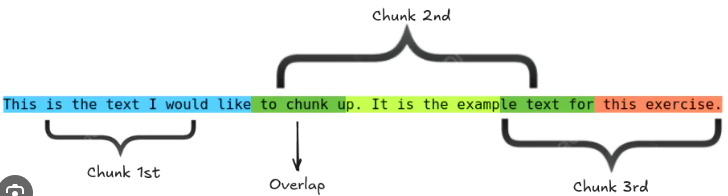
* Splits text into chunks of a fixed number of characters or tokens (e.g., 500 characters).
* It doesn’t care about sentence or paragraph boundaries.
* Though it is Simple and fast but may break sentences or lose context.

**Example**

A paragraph of 1,200 characters might be split into 3 chunks of 500, 500, and 200 characters—even if that cuts a sentence in half.

What is chunk\_size and chunk\_overlap?

* **chunk\_size**:
  + The number of characters in each chunk.
  + Example: chunk\_size=500 creates chunks of 500 characters.
* **chunk\_overlap**:
  + The number of characters repeated between chunks to keep context.
  + Example: chunk\_overlap=50 means the last 50 characters of one chunk appear at the start of the next.
  + Use overlap to avoid cutting off sentences or breaking flow across chunks.



###### TYPES OF FIXED CHUNKING

There are two types of **fixed chunking** in LangChain:

1. **CharacterTextSplitter**
2. **RecursiveCharacterTextSplitter**

Both are used to split long documents into smaller parts (chunks), but they work differently.

RecursiveCharacterTextSplitter

* Recommended for Generic Text . By default it tries to split the text by **[“\n\n”,”]n”,””,” “]** i.e. it tries to split text at **natural boundaries**: paragraphs → sentences → words → characters.
* Uses a **list of separators**, and recursively falls back if cleaner splits aren't possible.

|  |
| --- |
| **# Splits documents using RecursiveCharacterTextSplitter which preserves context better**  **from langchain\_text\_splitters import RecursiveCharacterTextSplitter**  **def recursive\_chunking(docs, chunk\_size=500, chunk\_overlap=50):**  **splitter = RecursiveCharacterTextSplitter(**  **chunk\_size=chunk\_size,**  **chunk\_overlap=chunk\_overlap**  **)**  **return splitter.split\_documents(docs)**  **recursive\_chunks = recursive\_chunking(docs)**  **print(f" Total recursive chunks: {len(recursive\_chunks)}\n")**  **print(f" Example: First Chunk \n{recursive\_chunks[0].page\_content[:]}")** |

Example

|  |
| --- |
| Let’s say we have this text:  ***"Healthcare personalization is key. It improves outcomes.\n\nAI tools help tailor experiences."***  And we want to split it into chunks of max 40 characters.  **Step-by-step breakdown:**  Try splitting by paragraphs:   * "Healthcare personalization is key. It improves outcomes." * "AI tools help tailor experiences."   **Each paragraph is ~50 characters, so still too long.**  Try splitting by sentences:   * "Healthcare personalization is key." * "It improves outcomes." * "AI tools help tailor experiences."   **These are ~30–35 characters each — perfect!**  So the final chunks would be:  **[**  **"Healthcare personalization is key.",**  **"It improves outcomes.",**  **"AI tools help tailor experiences."**  **]** |

CODE : [Langchain/buildiing\_rag\_pipeline/ai\_dojo\_fixed\_size\_data\_chunking\_RecursiveCharacterTextSplitter.py at main · avishekhsinhaRepo/Langchain](https://github.com/avishekhsinhaRepo/Langchain/blob/main/buildiing_rag_pipeline/ai_dojo_fixed_size_data_chunking_RecursiveCharacterTextSplitter.py)

CharacterTextSplitter

* Splits text into chunks based on a **fixed number of characters**.
* Uses a **single separator** (e.g. space " " or newline "\n").
* If a sentence or paragraph is too long, it may split mid-sentence.
* Simple and fast, but may not always preserve context cleanly.

|  |
| --- |
| **# Define a function to split text into fixed-size character chunks using LangChain's CharacterTextSplitter.**  **from langchain\_text\_splitters import CharacterTextSplitter**  **def fixed\_size\_chunking(docs, chunk\_size=500, chunk\_overlap=50):**    **splitter = CharacterTextSplitter(**  **separator=" ",**  **chunk\_size=chunk\_size,**  **chunk\_overlap=chunk\_overlap**  **)**  **return splitter.split\_documents(docs)**  **fixed\_chunks = fixed\_size\_chunking(docs)**  **print(f" Total fixed-size chunks: {len(fixed\_chunks)}\n")**  **print(f" Example:First Chunk \n{fixed\_chunks[0].page\_content[:]}")** |

CODE : [Langchain/buildiing\_rag\_pipeline/ai\_dojo\_fixed\_size\_data\_chunking\_CharacterTextSplitter.py at main · avishekhsinhaRepo/Langchain](https://github.com/avishekhsinhaRepo/Langchain/blob/main/buildiing_rag_pipeline/ai_dojo_fixed_size_data_chunking_CharacterTextSplitter.py)

HTMLTextSplitter

* Designed to handle HTML content.
* It strips tags and splits the text into chunks of fixed size.
* Useful when are working with web pages or scraped HTML data.

RecursiveJSONTextSplitter

* **Traversal**: It goes **depth-first** through the JSON structure.
* **Chunking**: It tries to keep nested objects **intact**, but will split them if they exceed the max\_chunk\_size.
* **Chunk Size**: Measured in **number of characters**.
* **String Handling**: Large strings are **not split** unless WE use a secondary text splitter.
* **List Handling**: Optionally, lists can be **converted to dicts** and split accordingly.

|  |
| --- |
| **INPUT JSON**  {  "title": "Demo Document",  "metadata": {  "author": "Avishekh Sinha",  "date": "2025-09-08"  },  "sections": [  {  "heading": "Introduction",  "content": "This is a long introduction text that might exceed the chunk size limit depending on configuration..."  },  {  "heading": "Details",  "content": "This section contains detailed information that is also quite lengthy and might need splitting."  }  ]  } |
| from langchain.text\_splitter import RecursiveJsonSplitter  splitter = RecursiveJsonSplitter(  chunk\_size=200, # max characters per chunk  min\_chunk\_size=100, # minimum characters per chunk  strip\_whitespace=True  )  chunks = splitter.split\_json(json\_data) |
| OUTPUT  **Chunk 1**:  {  "title": "Demo Document",  "metadata": {  "author": "Avishekh Sinha",  "date": "2025-09-08"  }  }  **Chunk 2**:  {  "sections": [  {  "heading": "Introduction",  "content": "This is a long introduction text that might exceed the chunk size limit..."  }  ]  }  **Chunk 3**:  {  "sections": [  {  "heading": "Details",  "content": "This section contains detailed information that is also quite lengthy..."  }  ]  } |

JSON CHUNKING – split\_json

|  |
| --- |
| import json  import requests  from langchain\_text\_splitters import **RecursiveJsonSplitter**  json\_data = requests.get('https://api.smith.langchain.com/openapi.json').json()  json\_splitter = RecursiveJsonSplitter(max\_chunk\_size=300)  json\_chunks = json\_splitter.split\_json(json\_data)  print(json\_chunks) |

JSON CHUNKING – create\_document()

|  |
| --- |
| import requests  from langchain\_text\_splitters import RecursiveJsonSplitter  json\_data = requests.get("https://api.smith.langchain.com/openapi.json").json()  json\_splitter = RecursiveJsonSplitter(max\_chunk\_size=300)  docs = json\_splitter.create\_documents(texts=[json\_data])  for doc in docs[:3]:      print(doc) |

JSON CHUNKING – split\_text()

|  |
| --- |
| import requests  from langchain\_text\_splitters import RecursiveJsonSplitter  json\_data = requests.get("https://api.smith.langchain.com/openapi.json").json()  json\_splitter = RecursiveJsonSplitter(max\_chunk\_size=300)  texts = json\_splitter.split\_text(json\_data)  print(texts[0])  print(texts[1]) |

##### SENTENCE BASED CHUNKING

* **What it does**: Splits text based on complete sentences.
* **How it works**: Groups sentences together until a size limit is reached (e.g., 3–5 sentences per chunk).
* **Pros**: Preserves natural language flow.
* **Cons**: Chunk sizes can vary, and some chunks may be too short or too long.

**What is sentences\_per\_chunk?**

* **sentences\_per\_chunk** defines **how many sentences** to include in each chunk.
* Example: sentences\_per\_chunk=3 will group 3 sentences together as one chunk.
* It keeps the chunks **meaningful and readable** by not cutting through sentences.
* Great for preserving logical flow and making sure each chunk conveys a complete thought or mini-topic.

|  |  |
| --- | --- |
| INSTALL **Natural Language Toolkit (NLTK)** | pip install **nltk** |

|  |
| --- |
| **from nltk.tokenize import sent\_tokenize**  **import nltk**  **# Download required NLTK data**  **try:**  **nltk.data.find("tokenizers/punkt\_tab")**  **except LookupError:**  **print("Downloading NLTK punkt tokenizer...")**  **nltk.download("punkt\_tab")**  **# Define a function to split each page into chunks of N sentences.**  **def sentence\_based\_chunking(docs, sentences\_per\_chunk=3):**  **chunks = []**  **for doc in docs:**  **sentences = sent\_tokenize(doc.page\_content)**  **for i in range(0, len(sentences), sentences\_per\_chunk):**  **chunk\_text = " ".join(sentences[i:i + sentences\_per\_chunk])**  **chunks.append(chunk\_text)**  **return chunks**  **sentence\_chunks = sentence\_based\_chunking(docs)**  **print(f" Total sentence-based chunks: {len(sentence\_chunks)}\n")**  **print(f" Example:\n{sentence\_chunks[0][:]}")** |
| * The sentence\_based\_chunking() function breaks the text into chunks of a fixed number of sentences (e.g. 3). * This method **respects natural language flow**, making each chunk easier for the model to understand. * It’s especially helpful when documents have clean sentence structures. |
| NOTE ON **Natural Language Toolkit (NLTK)**   * **Natural Language Toolkit (NLTK)** is Python library for working with human language data. * sent\_tokenize() is used to **split a paragraph or block of text into individual sentences**   **Example Code**  **from nltk.tokenize import sent\_tokenize**  **text = "Healthcare personalization is key. It improves outcomes. AI tools help tailor experiences."**  **sentences = sent\_tokenize(text)**  **print(sentences)**  **Output:**  **[**  **"Healthcare personalization is key.",**  **"It improves outcomes.",**  **"AI tools help tailor experiences."**  **]**  Each sentence is cleanly separated, even though the original text was a single string. |

CODE : [Langchain/buildiing\_rag\_pipeline/ai-dojo-sentence-based-data-chunking.py at main · avishekhsinhaRepo/Langchain](https://github.com/avishekhsinhaRepo/Langchain/blob/main/buildiing_rag_pipeline/ai-dojo-sentence-based-data-chunking.py)

##### SEMANTIC CHUNKING

* **Semantic chunking splits content based on paragraph breaks or logical boundaries. This helps preserve the meaning of each idea.**

|  |
| --- |
| **# Define a function to split based on paragraph breaks (using two newlines)**  **def semantic\_chunking(docs):**  **chunks = []**  **for doc in docs:**  **paragraphs = doc.page\_content.split("\n\n")**  **for para in paragraphs:**  **cleaned = para.strip()**  **if cleaned:**  **chunks.append(cleaned)**  **return chunks**  **semantic\_chunks = semantic\_chunking(docs)**  **print(f" Total semantic chunks: {len(semantic\_chunks)}")**  **print(f" Example:\n{semantic\_chunks[0][:]}")** |
| * **The semantic\_chunking() function splits text by paragraph breaks, using \n\n as the divider.** * **This keeps ideas grouped by their topic or theme, preserving meaning.** * **It's great when the original text is well-formatted with clear paragraph structure.** * **Best for: structured PDFs or reports with proper formatting (like research papers).** |

CODE: [Langchain/buildiing\_rag\_pipeline/ai-dojo-semantic-data-chunking.py at main · avishekhsinhaRepo/Langchain](https://github.com/avishekhsinhaRepo/Langchain/blob/main/buildiing_rag_pipeline/ai-dojo-semantic-data-chunking.py)

### STEP 3: CREATING EMBEDDINGS

#### COMMON EMBEDDING TECHNIQUES

|  |  |  |
| --- | --- | --- |
| **TECHNIQUE** | **DESCRIPTION** | **COMMON USE CASES** |
| **One-Hot Encoding** | Represents words or categories as binary vectors with a single high bit | Simple categorical data, small vocabularies |
| **TF-IDF** | Weighs word frequency by inverse document frequency | Text classification, information retrieval |
| **Word2Vec** | Learns word embeddings using skip-gram or CBOW models | Semantic similarity, NLP tasks |
| **GloVe** | Global Vectors for word representation using co-occurrence statistics | NLP tasks, semantic analysis |
| **FastText** | Extension of Word2Vec that includes subword information | Morphologically rich languages |
| **ELMo** | Contextual embeddings using deep bi-directional LSTMs | Named entity recognition, sentiment analysis |
| **BERT Embeddings** | Contextual embeddings from transformer-based BERT model | Question answering, text classification |
| **Sentence Embeddings** | Embeddings for entire sentences using models like Sentence-BERT | Semantic search, clustering |
| **Doc2Vec** | Embeddings for entire documents | Document classification, retrieval |
| **Autoencoders** | Neural networks that learn compressed representations | Dimensionality reduction, anomaly detection |
| **PCA (Principal Component Analysis)** | Linear dimensionality reduction technique | Preprocessing, visualization |
| **t-SNE / UMAP** | Non-linear dimensionality reduction for visualization | Embedding visualization |
| **Graph Embeddings** | Represent nodes/edges in graphs as vectors | Social networks, recommendation systems |
| **Image Embeddings** | Feature vectors from CNNs or vision transformers | Image classification, similarity search |

#### EMBEDDING PROVIDERS/PLATFORMS

**These embedding techniques are provided by different platforms or providers. For example,**

1. **OpenAI**
2. **Hugging Face**
3. **Ollama**

These are **platforms or providers** that offer access to models that **implement embedding techniques**. The below tab;e explains how these platform r**elate to Embedding Techniques**

|  |  |  |
| --- | --- | --- |
| **Platform** | **Uses Which Techniques?** | **How You Use It** |
| **OpenAI** | Transformer-based embeddings (e.g., GPT) | Call API → Get embeddings |
| **Hugging Face** | Word2Vec, BERT, RoBERTa, etc. | Download model → Run locally |
| **Ollama** | Transformer-based (e.g., LLaMA2, Mistral) | Install model → Run locally via CLI |

##### CREATING EMBEDDING USING OpenAI PLATFORM

* We can think OpenAI as a **cloud-based service**, where we send our text to their API, and they use a **pretrained model** (like text-embedding-3-small) to generate embeddings.
* The technique behind it is based on **transformer models**, similar to BERT or GPT.

|  |  |
| --- | --- |
| **INSTALL LANGCHAIN OPEN AI LIBRARY** | **pip install langchain-openai** |

|  |
| --- |
| **from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI**  **import os**  **from dotenv import load\_dotenv**  **load\_dotenv()**  **# Set your Azure OpenAI credentials**  **subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]**  **endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]**  **api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]**  **deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]**  **embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]**  **# Model used is text-embedding-ada-002**  **# Initialize the Azure OpenAI Embeddings**  **embeddings\_client = AzureOpenAIEmbeddings(**  **azure\_deployment=embedding\_deployment,**  **api\_version=api\_version,**  **azure\_endpoint=endpoint,**  **api\_key=subscription\_key,**  **)**  **query = "This is open AI embeddings"**  **query\_embedding = embeddings\_client.embed\_query(query)**  **print(f"Embedding for query: {query\_embedding}")**  **print(f"Length of the embedding: {len(query\_embedding)}") 🡪 *Length of the embedding: 1536*** |

* In the above example – it has created vectors (embeddings) of dimension 1536.
* We can customize the dimension of embeddings too (supported by few embedding models)

|  |
| --- |
| embeddings\_client = AzureOpenAIEmbeddings(      azure\_deployment=embedding\_deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,      dimensions=1024 # Specify the dimensions for the embedding (default is 1536, but can be set to 1024 for text-embedding-ada-002  } |

Example

In the below example- we will use text document

1. To create chunks
2. Create vectors of the chunks using embedding model
3. Store in Vector DB (Chroma)
4. Retrieve the data using query from vector DB

|  |
| --- |
| **Chroma.from\_documents()** **method** :   1. Takes a list of documents (usually text chunks). 2. Convert them into **embeddings** using a specified embedding model. 3. Store those embeddings in a Chroma vector store. |
| **INSTALL CHROMA DB : pip install chromadb** |

|  |
| --- |
| from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI  import os  from langchain\_community.document\_loaders import TextLoader  from dotenv import load\_dotenv  from langchain\_text\_splitters import RecursiveCharacterTextSplitter  from langchain\_community.vectorstores import Chroma  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  # Model used is text-embedding-ada-002  # Initialize the Azure OpenAI Embeddings  embeddings\_client = AzureOpenAIEmbeddings(  azure\_deployment=embedding\_deployment,  api\_version=api\_version,  azure\_endpoint=endpoint,  api\_key=subscription\_key,    )  # Get the directory of the current script  current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  speech\_file\_path = os.path.join(current\_dir, "speech.txt")  text\_document = TextLoader(speech\_file\_path).load()  text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=50, chunk\_overlap=10)  chunks = text\_splitter.split\_documents(text\_document)  **vector\_db = Chroma.from\_documents(chunks, embeddings\_client)**  query = "The world must be made safe for democracy."  result = vector\_db.similarity\_search(query, k=2)  print(f"Query: {query}")  print(f"Result: {result}") |

##### CREATING EMBEDDING USING HUGGINGFACE PLATFORM

* Hugging Face is like a **library of models**.
* You can download models like BERT, RoBERTa, or Sentence Transformers and run them locally.
* These models use techniques like:
  + **Word embeddings** (e.g., Word2Vec, GloVe)
  + **Contextual embeddings** (e.g., BERT, DistilBERT)
* You have full control—you can fine-tune, customize, or even train your own embedding models.

##### CREATING EMBEDDING USING OLLAMA PLATFORM

* Ollama is a **local-first platform**.
* It lets you run large language models (like LLaMA2 or Mistral) on your own machine.
* Some of these models support generating embeddings using transformer-based techniques.
* It’s great for privacy and offline use, but you’re limited to the models Ollama supports.

#### EMBEDDING TECHNIQUES

There are several ways to generate embeddings. Two widely used methods:

**a) Word2Vec**

* A traditional method using shallow neural networks.
* Captures the relationship between words based on **how often they appear together**.
* Learns embeddings like: “king - man + woman = queen”.

**b) AzureOpenAI using Langchain Embeddings**

OpenAI provides powerful pre-trained models for creating embeddings using their API.

**Most Common Model:**

* **text-embedding-ada-002** — small, fast, and very good.
* It produces a **1536-dimensional vector** for each chunk.
* Ideal for search, retrieval, and similarity tasks.

**How It Works:**

* You send a chunk of text to AzureOpenAI Endpoint.
* The model returns a fixed-length embedding (vector) that captures the **semantic meaning** of the text.
* AzureOpenAI using Langchain embeddings are **widely used in production RAG pipelines**, especially when quality and ease-of-use matter.

Embedding Dimensions

|  |  |  |
| --- | --- | --- |
| Method | Dimension Size | Notes |
| **Word2Vec** | Custom (e.g., 100) | Based on training setup; basic context |
| **OpenAI** | 1536 | Very high quality, best for production use |

* All methods turn text into **vectors**, but the **quality and context-awareness** improve as you go from Word2Vec → OpenAI.

#### CREATING WORD EMBEDDING USING WORD2VEC

#### AZUREOPENAI USING LANGCHAIN EMBEDDINGS

### STEP 4: STORING AND QUERYING VECTOR DATA

#### VECTOR DB – FAISS

* Facebook AI Similarity Search (**Faiss**) is a library for efficient similarity search and clustering of dense vectors.
* It contains algorithms that search in sets of vectors of any size, up to ones that possibly do not fit in RAM.
* It also contains supporting code for evaluation and parameter tuning.

|  |  |
| --- | --- |
| **INSTALL FAISS** | **pip install faiss-cpu** |

|  |
| --- |
| from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI  import os  from langchain\_community.document\_loaders import TextLoader  from dotenv import load\_dotenv  from langchain\_text\_splitters import RecursiveCharacterTextSplitter  from langchain\_community.vectorstores import FAISS  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  # Model used is text-embedding-ada-002  # Initialize the Azure OpenAI Embeddings  embeddings\_client = AzureOpenAIEmbeddings(      azure\_deployment=embedding\_deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,)  # Get the directory of the current script  current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  speech\_file\_path = os.path.join(current\_dir, "speech.txt")  text\_document = TextLoader(speech\_file\_path).load()  text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=50, chunk\_overlap=10)  chunks = text\_splitter.split\_documents(text\_document)  **vector\_db = FAISS.from\_documents(chunks, embeddings\_client)**  query="How does the speaker describe the desired outcome of the war?"  result = vector\_db.similarity\_search(query)  print(f"Query: {query}")  print(f"Result: {result}") |

##### FAISS- SAVE VECTOR INDEX IN LOCAL DISK

* In LangChain, **save\_local()** is a method available on certain vector store implementations (like FAISS) that allows us to persist the vector index to disk.
* **This is useful when we want to avoid recomputing embeddings every time the app starts.**

|  |
| --- |
| What is vector Index?   * A vector index is a data structure that stores embeddings and allows for efficient similarity search. * It enables fast retrieval of documents or chunks that are semantically similar to a query. * FAISS, Chroma, and Annoy are popular vector index libraries used in LangChain. * Process   + Generate embeddings from your documents.   + We store those embeddings in a vector index.   + Then we query the index with a new embedding (from a user query) to find similar documents. |

|  |  |
| --- | --- |
| from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI  import os  from langchain\_community.document\_loaders import TextLoader  from dotenv import load\_dotenv  from langchain\_text\_splitters import RecursiveCharacterTextSplitter  from langchain\_community.vectorstores import FAISS  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  # Model used is text-embedding-ada-002  # Initialize the Azure OpenAI Embeddings  embeddings\_client = AzureOpenAIEmbeddings(      azure\_deployment=embedding\_deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key)  # Get the directory of the current script  current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  speech\_file\_path = os.path.join(current\_dir, "speech.txt")  text\_document = TextLoader(speech\_file\_path).load()  text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=50, chunk\_overlap=10)  chunks = text\_splitter.split\_documents(text\_document)   * **Converts the chunks into embeddings and stores them in a FAISS index.** * **Saves the index locally for reuse.**   vector\_db = FAISS.from\_documents(chunks, embeddings\_client)  vector\_db.save\_local("faiss\_index")   * **Loads the saved FAISS index.** * **Performs a similarity search using the query to find the most relevant chunk(s) from the speech.**   new\_vector\_db = vector\_db.load\_local("faiss\_index",embeddings\_client,allow\_dangerous\_deserialization=True)  query="How does the speaker describe the desired outcome of the war?"  result = new\_vector\_db.similarity\_search(query)  print(f"Query: {query}")  print(f"Result: {result}") | |
| Note   * FAISS will create an adjacent folder with the name of “**index name**” * It create .pkl and .faiss file to store the index |  |

#### VECTOR DB - CHROMA

|  |  |
| --- | --- |
| **INSTALL Chroma (deprecated)** | pip install Chroma |
| **INSTALL LANGCHAIN CHROMA** | pip install **langchain\_chroma** |

* In the below example – we are using 2 -ways to retrieve from vector store
  + **Using Similarity Seach**
  + **Using Retriever**

|  |
| --- |
| from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI  import os  from langchain\_community.document\_loaders import TextLoader  from dotenv import load\_dotenv  from langchain\_text\_splitters import RecursiveCharacterTextSplitter  **from langchain\_chroma import Chroma**  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  # Model used is text-embedding-ada-002  # Initialize the Azure OpenAI Embeddings  embeddings\_client = AzureOpenAIEmbeddings(      azure\_deployment=embedding\_deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,)  # Get the directory of the current script  current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  speech\_file\_path = os.path.join(current\_dir, "speech.txt")  text\_document = TextLoader(speech\_file\_path).load()  text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=50, chunk\_overlap=10)  chunks = text\_splitter.split\_documents(text\_document)  **vector\_db = Chroma.from\_documents(chunks, embeddings\_client)**  query="How does the speaker describe the desired outcome of the war?"  print(f"Query: {query}")  ## USING SIMILARITY SEARCH  result\_ss = new\_vector\_db.similarity\_search(query)  print(f"Result Similarity Search: {result\_ss}")  ## USING RETRIEVER  retriever = new\_vector\_db.as\_retriever()  result = retriever.invoke(query)  print(f"Result : {result}") |

##### CHROMA- SAVE VECTOR INDEX IN LOCAL DISK

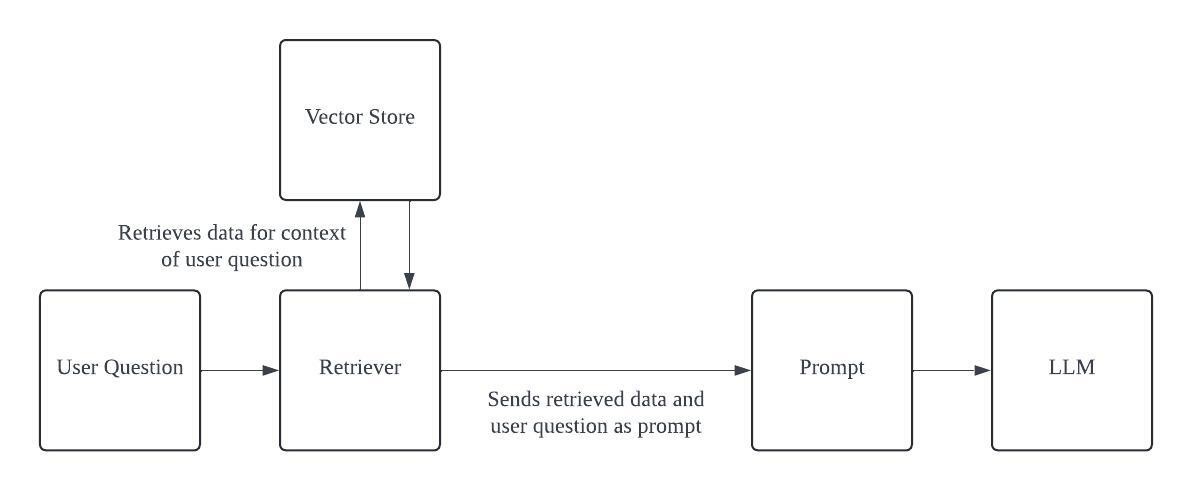
|  |
| --- |
| from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI  import os  from langchain\_community.document\_loaders import TextLoader  from dotenv import load\_dotenv  from langchain\_text\_splitters import RecursiveCharacterTextSplitter  **from langchain\_chroma import Chroma**  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  # Model used is text-embedding-ada-002  # Initialize the Azure OpenAI Embeddings  embeddings\_client = AzureOpenAIEmbeddings(      azure\_deployment=embedding\_deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,    )  # Get the directory of the current script  current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  speech\_file\_path = os.path.join(current\_dir, "speech.txt")  text\_document = TextLoader(speech\_file\_path).load()  text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=50, chunk\_overlap=10)  chunks = text\_splitter.split\_documents(text\_document)  **vector\_db = Chroma.from\_documents(chunks, embeddings\_client, persist\_directory="./chroma\_db")**  # To load the persisted Chroma vector store  **new\_vector\_db = Chroma(persist\_directory="./chroma\_db", embedding\_function=embeddings\_client)**  query = "How does the speaker describe the desired outcome of the war?"  result = new\_vector\_db.similarity\_search(query)  print(f"Query: {query}")  print(f"Result: {result}") |

### STEP 5: DATA RETRIEVAL (QUERING VECTOR DB)

**If we already have similarity search why we need Retrieval Chain?**

When we talk about **RAG (Retrieval-Augmented Generation)**, two main parts are involved:

1. **Similarity Search** (from a vector DB)
   * This step finds the most relevant chunks/documents based on a query embedding.
   * Example: You embed *“What are the harmful chemicals in this face wash?”* and search vector DB to get the top 3 similar chunks.
   * Output will be Raw documents/snippets.
2. **Retrieval Chain**



* + This is the *process layer* that connects similarity search with the LLM.
  + It doesn’t just *retrieve*, it orchestrates **how the retrieved docs are combined with the query** before passing to the LLM.
  + Responsibilities include:
    - **Query transformation** → refine, rewrite, or expand the query before searching.
    - **Filtering/ranking** → pick best results, re-rank, or apply metadata filters (time, user context, permissions).
    - **Context packaging** → format retrieved docs into a structured prompt (e.g., “Here are relevant documents, answer using only them.”).
    - **Memory handling** → include conversation history (for conversational RAG).
    - **Fallback strategies** → if retrieval fails, either expand search, call an API, or return a default response.

👉 So, **similarity search = the raw retrieval**  
👉 **retrieval chain = the orchestrator** that makes RAG robust, accurate, and LLM-friendly.

Without a retrieval chain:

* You’d just get top-k docs and dump them into the LLM. This often leads to hallucinations or irrelevant context.

With a retrieval chain:

* You get a controlled pipeline — where queries are refined, irrelevant docs filtered, and responses grounded in the right context.

Analogy:

* **Similarity search** is like asking a librarian for “books about AI.” They hand you 10 books.
* **Retrieval chain** is like the librarian *also reading the index, filtering for “RAG topics only,” giving you 2 best chapters, and attaching sticky notes saying “Focus on pages 15–20.”*

#### RETRIEVAL CHAIN

* Retrieval Chain is a powerful pattern used to retrieve relevant information from a knowledge base (like a vector store) and then use a Language Model (LM) to generate a response based on that information.
* A Retrieval Chain combines two main components:
  + **Retriever**: Finds relevant documents or chunks based on a query.
  + **LLM** (Language Model): Uses those documents to answer the query, summarize, or perform other tasks.

**NOTE : This is commonly used in RAG (Retrieval-Augmented Generation) systems.**

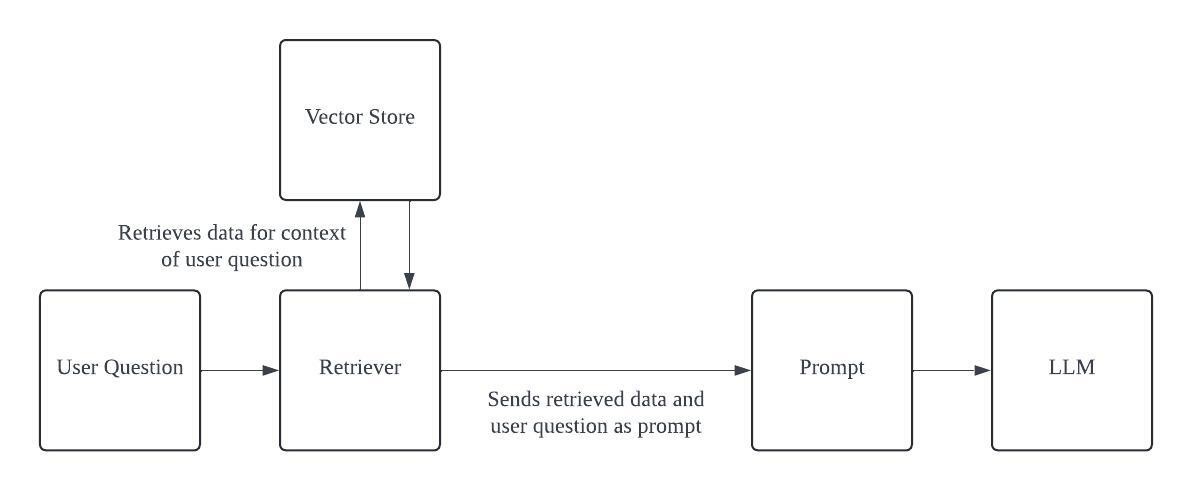
##### CORE COMPONENTS

|  |  |
| --- | --- |
| **Component** | **Role** |
| **Vector Store** | Stores embedded chunks of text (e.g., FAISS, Chroma, Pinecone). |
| **Retriever** | Queries the vector store to find relevant chunks. |
| **Prompt Template** | Defines how the retrieved content and user query are passed to the LLM. |
| **LLM Chain** | Uses the prompt and retrieved documents to generate a response. |
| **Output Parser *(optional)*** | Structures the output from the LLM. |

* In the above steps – so far, we have the data in the vector store, let’s create a retrieval chain.
* For the retrieval chain, we need a prompt. The prompt will have the retrieved data and the user question. To do this, we will use **ChatPromptTemplate**.

|  |  |
| --- | --- |
| **prompt = ChatPromptTemplate.from\_template("""Answer the following question based only on the provided context:**  **<context>**  **{context}**  **</context>**  **Question: {input}""")** | |
| **ChatPromptTemplate.from\_template(...)** | This creates a prompt that can be reused with different inputs. |
| **{context}** | This is where LangChain will insert the retrieved documents or chunks (your background info). |
| **{input}** | This is where LangChain will insert the user's question |
| **<context> ... </context>** | These tags are just for formatting — they help the model clearly see what part is the context. |
| **Example**  If the context is:  "LangChain helps build apps with language models."  And the question is: "What is LangChain?" | Answer the following question based only on the provided context:   <context>  LangChain helps build apps with language models.  </context>  Question: What is LangChain? |
| Why This Is Useful | This prompt ensures the model:   * Focuses only on the provided context * Doesn’t guess or hallucinate * Gives accurate answers based on your data |

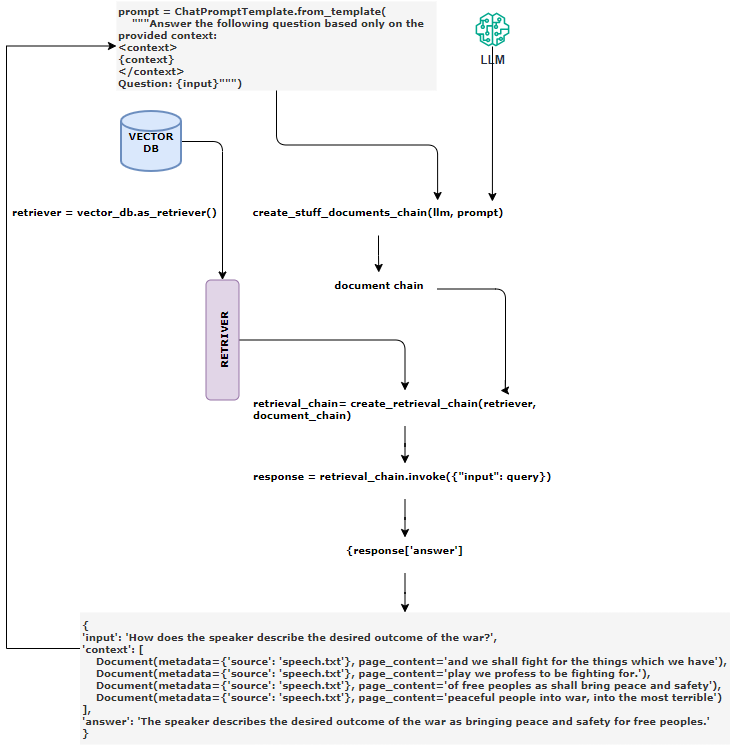
**The next step is to retrieve the data that is closely related to the user question from the vector store and provide it to the LLM as part of the prompt**



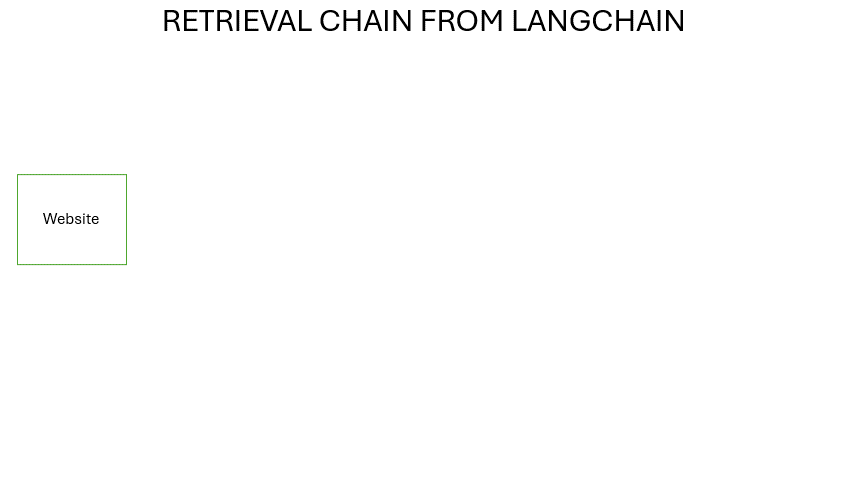
This is done using a retriever and a retrieval chain.

A diagram of a process flow

AI-generated content may be incorrect.



##### SUMMARY OF STEPS

****

##### EXAMPLE

|  |
| --- |
| from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI  import os  from langchain\_community.document\_loaders import TextLoader  from dotenv import load\_dotenv  from langchain\_text\_splitters import RecursiveCharacterTextSplitter  from langchain\_community.vectorstores import Chroma  from langchain\_core.prompts import ChatPromptTemplate  from langchain.chains.combine\_documents import create\_stuff\_documents\_chain  from langchain.chains import create\_retrieval\_chain  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  # Define the LLM (Language Model) instance  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  # Model used is text-embedding-ada-002  # Initialize the Azure OpenAI Embeddings  embeddings\_client = AzureOpenAIEmbeddings(      azure\_deployment=embedding\_deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key)  # Get the directory of the current script  current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  speech\_file\_path = os.path.join(current\_dir, "speech.txt")  text\_document = TextLoader(speech\_file\_path).load()  text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=50, chunk\_overlap=10)  chunks = text\_splitter.split\_documents(text\_document)  vector\_db = Chroma.from\_documents(chunks, embeddings\_client)  query = "How does the speaker describe the desired outcome of the war?"  result = vector\_db.similarity\_search(query)  prompt = ChatPromptTemplate.from\_template(      """Answer the following question based only on the provided context:  <context>  {context}  </context>  Question: {input}"""  )  document\_chain = create\_stuff\_documents\_chain(llm, prompt=prompt)  retriever = vector\_db.as\_retriever()  retrieval\_chain = create\_retrieval\_chain(retriever, document\_chain)  response = retrieval\_chain.invoke({"input": query})  print(response) |

## EXAMPLES – RAG PIPELINES

### EXAMPLE 1 – TEXT DOCUMENT DATA SOURCE

* Data source: [Docs/Machine Learning/resources/product-data.txt at master · avishekhsinhaRepo/Docs](https://github.com/avishekhsinhaRepo/Docs/blob/master/Machine%20Learning/resources/product-data.txt)

|  |
| --- |
| from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI  from langchain\_community.document\_loaders import TextLoader  from langchain\_text\_splitters import RecursiveCharacterTextSplitter  from langchain\_chroma import Chroma  from langchain.prompts import ChatPromptTemplate  import os  from dotenv import load\_dotenv  from langchain.chains import create\_retrieval\_chain  from langchain.chains.combine\_documents import create\_stuff\_documents\_chain  load\_dotenv()  # Set your Azure OpenAI credentials (single resource for both models)  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  gpt\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"] # Your GPT model deployment  # Initialize the Azure OpenAI Chat model for text generation  llm = AzureChatOpenAI(  azure\_deployment=gpt\_deployment,  api\_version=api\_version,  azure\_endpoint=endpoint,  api\_key=subscription\_key,  )  # Initialize the Azure OpenAI Embeddings for vector creation  embeddings = AzureOpenAIEmbeddings(  azure\_deployment=embedding\_deployment,  api\_version=api\_version,  azure\_endpoint=endpoint,  api\_key=subscription\_key,)  # Initialize the Azure OpenAI LLM  prompt\_template = ChatPromptTemplate.from\_messages(  [  (  "system",  """You are an assistant for answering questions.  Use the provided context to respond.If the answer  isn't clear, acknowledge that you don't know.  Limit your response to three concise sentences.  {context}  """,  ),  ("human", "{input}"),  ]  )  document = TextLoader("rag/product-data.txt").load()  text\_spiltter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)  chunks = text\_spiltter.split\_documents(document)  vector\_store = Chroma.from\_documents(chunks, embeddings)  retriever = vector\_store.as\_retriever()  qa\_chain = create\_stuff\_documents\_chain(llm, prompt\_template)  rag\_chain = create\_retrieval\_chain(retriever, qa\_chain)  question = input("Enter Your Question")  if question:  response = rag\_chain.invoke({"input": question})  print(response["answer"]) |

### EXAMPLE-2 – PDF DOCUMENT DATASOURCE

|  |  |
| --- | --- |
| INSTALL pypdf | pip install pypdf |

|  |
| --- |
| from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI  from langchain\_community.document\_loaders import PyPDFLoader  from langchain\_text\_splitters import RecursiveCharacterTextSplitter  from langchain\_chroma import Chroma  from langchain.prompts import ChatPromptTemplate  import os  from dotenv import load\_dotenv  from langchain.chains import create\_retrieval\_chain  from langchain.chains.combine\_documents import create\_stuff\_documents\_chain  load\_dotenv()  # Set your Azure OpenAI credentials (single resource for both models)  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  gpt\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Your GPT model deployment  # Initialize the Azure OpenAI Chat model for text generation  llm = AzureChatOpenAI(      azure\_deployment=gpt\_deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,)  # Initialize the Azure OpenAI Embeddings for vector creation  embeddings = AzureOpenAIEmbeddings(      azure\_deployment=embedding\_deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  # Initialize the Azure OpenAI LLM  prompt\_template = ChatPromptTemplate.from\_messages(      [          (              "system",              """You are an assistant for answering questions.                  Use the provided context to respond.If the answer                  isn't clear, acknowledge that you don't know.                  Limit your response to three concise sentences.                  {context}        """,          ),          ("human", "{input}"),      ]  )  document = PyPDFLoader("rag/academic\_research\_data.pdf").load()  text\_spiltter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)  chunks = text\_spiltter.split\_documents(document)  vector\_store = Chroma.from\_documents(chunks, embeddings)  retriever = vector\_store.as\_retriever()  qa\_chain = create\_stuff\_documents\_chain(llm, prompt\_template)  rag\_chain = create\_retrieval\_chain(retriever, qa\_chain)  question = input("Enter Your Question=")  if question:      response = rag\_chain.invoke({"input": question})      print(response["answer"]) |

### EXAMPLE- 3 – WORD DOCUMENT DATASOURCE

|  |  |
| --- | --- |
| INSTALL docx2txt | pip install docx2txt |

|  |
| --- |
| from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI  from langchain\_community.document\_loaders import Docx2txtLoader  from langchain\_text\_splitters import RecursiveCharacterTextSplitter  from langchain\_chroma import Chroma  from langchain.prompts import ChatPromptTemplate  import os  from dotenv import load\_dotenv  from langchain.chains import create\_retrieval\_chain  from langchain.chains.combine\_documents import create\_stuff\_documents\_chain  load\_dotenv()  # Set your Azure OpenAI credentials (single resource for both models)  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  gpt\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Your GPT model deployment  # Initialize the Azure OpenAI Chat model for text generation  llm = AzureChatOpenAI(      azure\_deployment=gpt\_deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  # Initialize the Azure OpenAI Embeddings for vector creation  embeddings = AzureOpenAIEmbeddings(      azure\_deployment=embedding\_deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,)  # Initialize the Azure OpenAI LLM  prompt\_template = ChatPromptTemplate.from\_messages(      [          (              "system",              """You are an assistant for answering questions.                  Use the provided context to respond.If the answer                  isn't clear, acknowledge that you don't know.                  Limit your response to three concise sentences.                  {context}        """,          ),          ("human", "{input}"),      ]  )  document = Docx2txtLoader("rag/Expenses\_policy.docx").load()  text\_spiltter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)  chunks = text\_spiltter.split\_documents(document)  vector\_store = Chroma.from\_documents(chunks, embeddings)  retriever = vector\_store.as\_retriever()  qa\_chain = create\_stuff\_documents\_chain(llm, prompt\_template)  rag\_chain = create\_retrieval\_chain(retriever, qa\_chain)  question = input("Enter Your Question=")  if question:      response = rag\_chain.invoke({"input": question})      print(response["answer"]) |

# LANGSMITH

**LangSmith** is a platform designed to help developers build, debug, and evaluate applications powered by large language models (LLMs). It provides tools for:

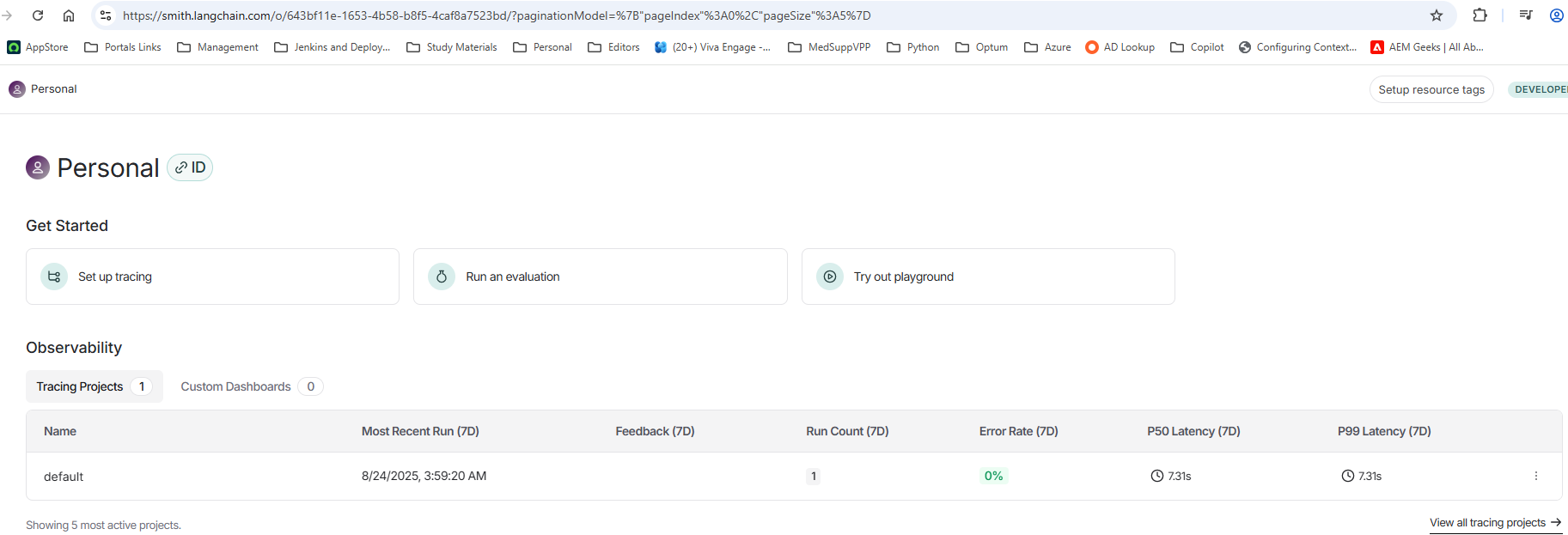
* + **Observability**: Tracking and analyzing how LLMs respond to prompts.
  + **Evaluation**: Measuring the quality and reliability of model outputs.
  + **Prompt Engineering**: Iterating and refining prompts to improve performance.

It works independently or alongside frameworks like LangChain and LangGraph, making it useful for both prototyping and production-grade AI systems.

## STEP TO SET UP LANGSMITH

* Step 1: Login /Signup to <https://smith.langchain.com/>

LangSmith Dashboard



### CREATE API KEY

1. Go to Settings 🡪 Create API Key

A screenshot of a computer

AI-generated content may be incorrect.

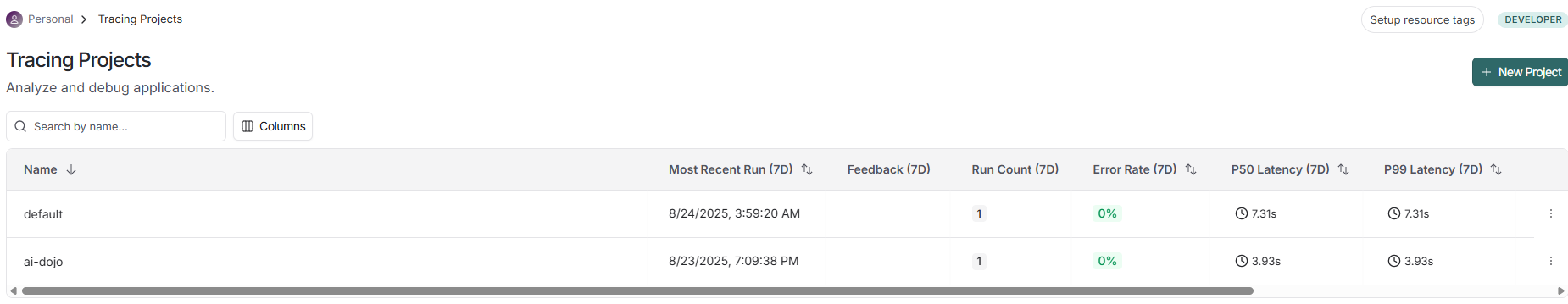
### CODE

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  import os  from dotenv import load\_dotenv  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  os.environ["LANGSMITH\_TRACING"] = os.getenv("LANGSMITH\_TRACING")  os.environ["LANGSMITH\_API\_KEY"] = os.getenv("LANGSMITH\_API\_KEY")  os.environ["LANGSMITH\_PROJECT"] = os.getenv("LANGSMITH\_PROJECT")  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,      temperature=0)  response = llm.invoke("What is Generative AI?")  print(response) |
| **LANGSMITH\_TRACING=true**  **LANGSMITH\_API\_KEY=<lang\_chain\_api\_key>**  **LANGSMITH\_PROJECT=ai-dojo** |

### TRACING PROJECT

A screenshot of a computer

AI-generated content may be incorrect.



* New Tracing project will be created “ai-dojo”
* It will show the trace of all the inputs & outputs.

A screenshot of a computer

AI-generated content may be incorrect.