Table of Contents

[LANGCHAIN 3](#_Toc205821188)

[KEY FEATURES & MODULES 3](#_Toc205821189)

[CONCEPTS 3](#_Toc205821190)

[CHAINING 4](#_Toc205821191)

[COMPONENTS OF LANGCHAIN 5](#_Toc205821192)

[CHAT MODELS 5](#_Toc205821193)

[LANGCHAIN IN ACTION 5](#_Toc205821194)

[PROMPT TEMPLATE 6](#_Toc205821195)

[EXAMPLE 7](#_Toc205821196)

[CHAINS 7](#_Toc205821197)

[LANGCHAIN EXPRESSION LANGUAGE (LCEL) – SIMPLE DEFINITION 7](#_Toc205821198)

[EXAMPLE 8](#_Toc205821199)

[SEQUENTIAL PROMPTS 8](#_Toc205821200)

[SIMPLE SEQUENTIAL CHAIN 9](#_Toc205821201)

[REGULAR SEQUENTIAL CHAIN 9](#_Toc205821202)

[CHAT HISTORY 9](#_Toc205821203)

[CHAT PROMPT TEMPLATE 10](#_Toc205821204)

[MAINTAINING HISTORY 10](#_Toc205821205)

[EMBEDDING MODELS 10](#_Toc205821206)

[EMBEDDINGS 10](#_Toc205821207)

[HOW EMBEDDINGS ARE GENERATED 11](#_Toc205821208)

[GENERATING EMBEDDINGS USING LANGCHAIN 12](#_Toc205821209)

[SIMILARITY FINDER 12](#_Toc205821210)

[HOW IT WORKS? 13](#_Toc205821211)

[VECTORS 14](#_Toc205821212)

[WHY ARE VECTORS USEFUL? 14](#_Toc205821213)

[VECTOR DATABASE 15](#_Toc205821214)

[RAG (RETRIEVER AUGUMENTED GENERATION) 19](#_Toc205821215)

[RAG WORKFLOW 20](#_Toc205821216)

[WHY RAG? 20](#_Toc205821217)

[BENEFITS 20](#_Toc205821218)

[RAG ARCHITECTURE 21](#_Toc205821219)

[RAG PROCESS- BUILDING RAG PIPELINE 22](#_Toc205821220)

[STEP BY STEP BUILDING RAG PIPELINE 23](#_Toc205821221)

[LOAD DATA – PDF 23](#_Toc205821222)

[DATA CHUNKING 25](#_Toc205821223)

[EXAMPLE 1– TEXT DOCUMENT 27](#_Toc205821224)

[EXAMPLE-2 – PDF DOCUMENT 29](#_Toc205821225)

[EXAMPLE- 3 – WORD DOCUMENT 30](#_Toc205821226)

[AGENTS 32](#_Toc205821227)

[HOW AGENTS WORK 32](#_Toc205821228)

[SETTING UP 32](#_Toc205821229)

# 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.
* Enables building apps like **agents** and **RAG (Retrieval-Augmented Generation)** systems.

A diagram of a diagram

AI-generated content may be incorrect.

## KEY FEATURES & MODULES

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 not just another AI tool, it is a comprehensive 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=2H8z7i6cZxgDx8X7WksANiaTZQveZBoqYOsHBjzsXB1ObMv7MyqKJQQJ99BHACYeBjFXJ3w3AAABACOGs6Zg**  **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.
* Instead of writing a full prompt every time, you create a **template** with placeholders, and then fill in those placeholders dynamically based on user input or other data.

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)** |

## CHAINS

### 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

## CHAT HISTORY

### CHAT PROMPT TEMPLATE

* ChatPromptTemplate is a specialized prompt class in LangChain designed for **chat-based models** like OpenAI's GPT-4 or Anthropic's Claude.
* It helps us structure prompts in a way that mimics a **conversation**, using roles like system, user, and assistant.

Example

|  |
| --- |
| **from langchain\_openai import AzureChatOpenAI**  **from langchain\_core.prompts import ChatPromptTemplate**  **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,**  **)**  **prompt = ChatPromptTemplate.from\_messages(**  **[("system", "You are a Sales Man."),**  **("human", "{input}")]**  **)**  **chain = prompt | llm**  **response = chain.invoke({"input": "Sell me the pen."})**  **print(response.content)** |

### MAINTAINING HISTORY

## EMBEDDING MODELS

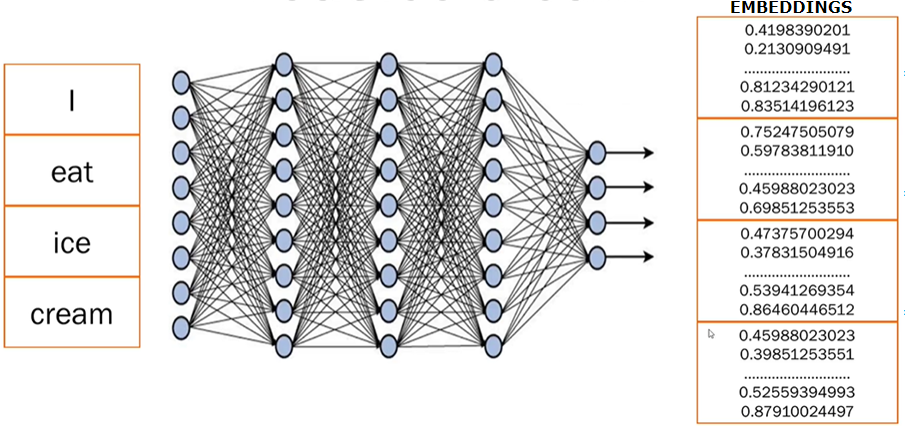
### EMBEDDINGS

* 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 in meaning.
* 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?

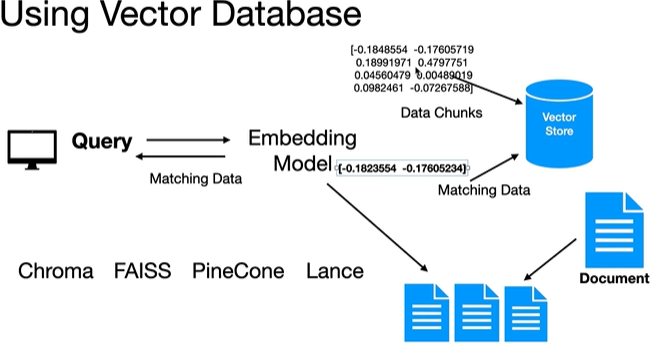
It helps to:

* **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()**

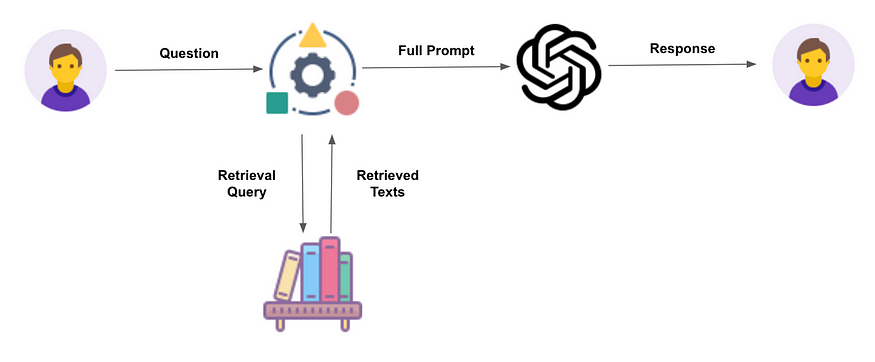
**The retriever instance is capable of taking plain text, calculate the embedding for it, then query the vector database with that embedded text and retrieve the data as well.**

|  |
| --- |
| 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.

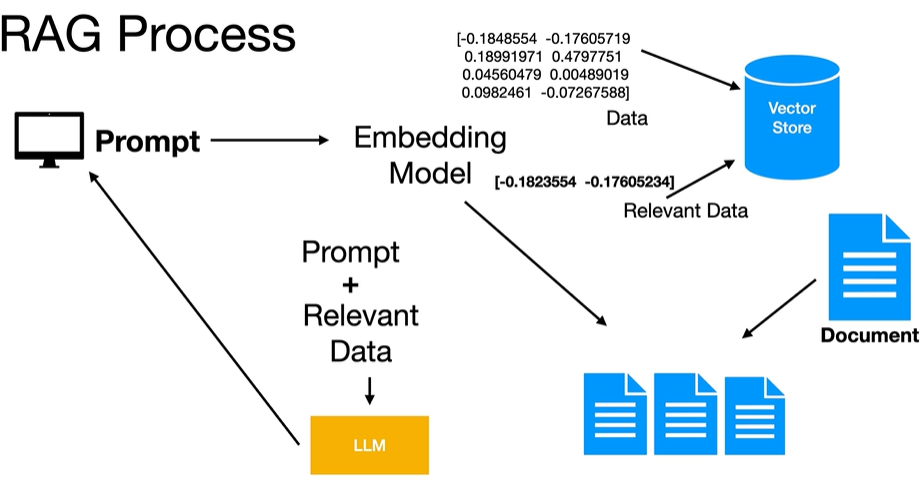
Step 7: Show the Answer

* 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.



1. **Step 1: Data Chunking**
   1. The first step is to break down the documents into smaller, meaningful pieces – a process called **chunking**.
   2. Data chunking 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.
   3. By splitting the documents into bite-sized chunks, such as paragraphs or sections, we can retain sufficient context without overwhelming the system.
   4. In LangChain, utilities like `**RecursiveCharacterTextSplitter**` or `**MarkdownHeaderTextSplitter**` can be used depending on the structure of the data.
   5. 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** |
| **document = TextLoader("rag/product-data.txt").load()**  **text\_spiltter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)**  **chunks = text\_spiltter.split\_documents(document)** |

1. **Step 2: Data Retrieval – Embedding And Vector Store**

* Once the documents are chunked, the next step is **embedding** and storing them in a **vector database**. Embedding transforms text into high-dimensional vectors that capture meaning and context. Each vector serves as a **numerical fingerprint** of its corresponding chunk.
* These embeddings are then stored in a vector database, which enables **fast similarity search** when a user submits a query. (Note : The vector store will have the embeddings as the metadata and the chunk itself as the data in them)
* This process ensures that relevant information can be retrieved efficiently, even if the query doesn’t use the exact wording found in the original text.

1. **Step 3: Reranking**
   1. Sometimes, the system retrieves **multiple chunks of text**, but the **best one isn’t at the top**. That’s where **reranking** helps:
      1. It takes the top few chunks (say, top 10).
      2. It uses a smarter model (called a **cross-encoder**) to **re-evaluate** how well each chunk matches the question.
      3. Then it **reorders** them so the most relevant chunk is first.
   2. Benefits
      1. It improves **accuracy**.
      2. It helps in **complex topics** like medical or legal documents.
      3. It ensures the AI gets the **best possible context** before answering.
2. **Step 4: Response Generation**
   1. When we receive a prompt from the user, we will use the embedding model to calculate the embedding for that prompt.
   2. We then use that embedding to retrieve the relevant data(chunk – it can be reranked chunks) from the vector store
   3. The prompt + the data retrieved from Vector database is send it to the LLM so that it can respond back correctly.
3. **Step 5: Evaluation**
   1. RAG pipelineis important to check whether the answers are:
      1. **Accurate**
      2. **Based on the right context**
      3. **Helpful to clinical teams**
   2. LangChain doesn’t force you to use a specific evaluation tool, but it supports:
      1. **Manual reviews**
      2. **Automated metrics**
   3. Key areas to evaluate include:
      1. **Precision and recall** of retrieved chunks
      2. **Faithfulness and factuality** of LLM-generated responses
      3. **Human feedback**
      4. **Groundedness scores** to ensure the model uses only retrieved data

## STEP BY STEP BUILDING RAG PIPELINE

### 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 |

### 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.
* **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**

#### DIFFERENT CHUNKING METHODS

##### 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.

**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.**  **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)**  **# Apply it**  **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[:]}")** |

**RecursiveCharacterTextSplitter**

* 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.
* Produces better structured chunks with less context loss.
* More advanced and preserves semantic meaning better.

Use RecursiveCharacterTextSplitter when you want **smart chunking**, especially for documents with mixed formatting or longer sentences.

|  |
| --- |
| **# Splits documents using RecursiveCharacterTextSplitter which preserves context better**  **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)**  **# Apply it**  **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[:]}")** |

##### 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.

|  |
| --- |
| **# 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. |

##### SEMANTIC CHUNKING

* **What it does**: Splits text based on **meaning**, not just size or sentence count.
* **How it works**: Uses AI models to detect topic shifts or semantic boundaries.
* **Pros**: Most intelligent and context-aware.
* **Cons**: Requires more computation and setup.

|  |
| --- |
| **# 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.** |

## EXAMPLE 1– TEXT DOCUMENT

* 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

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

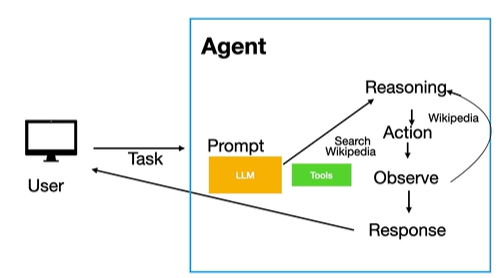
|  |
| --- |
| from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI  from langchain\_community.document\_loaders import PyPDFLoader  from langchain\_text\_splitters import RecursiveCharacterTextSplitter  from langchain\_chroma import Chroma  from langchain.prompts import ChatPromptTemplate  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

|  |  |
| --- | --- |
| 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"]) |

## AGENTS



### HOW AGENTS WORK

1. Task Assignment

* The user gives a task to the agent.
* The agent uses the **LLM as a reasoning engine** to understand the task.

2. Tool Selection

* The agent evaluates which tools it has available.
* It decides which tool is best suited to accomplish the task.

3. Action Execution

* The agent uses the selected tool to perform an action.
* It receives a **response** from the tool.

4. Response Evaluation

* The agent **does not immediately use or return** the response.
* It **observes and evaluates** the response.

5. Reasoning Loop

* If the response is **satisfactory**, it proceeds to generate the final output.
* If **not**, it continues reasoning:
  + It may reuse the same tool.
  + Or it may choose a **different tool**.

6. Final Response

* Once the agent is confident in the result, it uses the best response to generate a final answer.
* The final answer is sent back to the user.

LLMs That Support Agents

* **GPT** by OpenAI
* **Gemini** by Google
* **Mistral** from the open-source community

### SETTING UP

|  |  |
| --- | --- |
| Install The Tools | * **Wikipedia: pip install Wikipedia** * **Duckduckgo Search: pip install duckduckgo-search** * ***N****ote :* ***DuckDuckGo is a privacy-focused search engine that helps you find information online—just like Google or Bing*** |

|  |
| --- |
| **import os**  **from langchain\_openai import AzureChatOpenAI**  **# Import hub to pull pre-built prompts from LangChain Hub**  **from langchain import hub**  **# Import agent creation and execution classes for ReAct pattern**  **from langchain.agents import create\_react\_agent, AgentExecutor**  **# Import function to load predefined tools (Wikipedia, DuckDuckGo search)**  **from langchain\_community.agent\_toolkits.load\_tools import load\_tools**  **from dotenv import load\_dotenv**  **load\_dotenv()**  **subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]**  **endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]**  **api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]**  **gpt\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"] # Your GPT model deployment**  **llm = AzureChatOpenAI(**  **azure\_deployment=gpt\_deployment, # Specify which deployment to use**  **api\_version=api\_version, # Set the API version**  **azure\_endpoint=endpoint, # Set the Azure endpoint URL**  **api\_key=subscription\_key, # Provide authentication key**  **)**  **# Pull a pre-built ReAct (Reasoning + Acting) prompt template from LangChain Hub**  **# ReAct prompts help the agent reason through problems step-by-step and take actions**  **prompt = hub.pull("hwchase17/react")**  **# Load predefined tools for the agent to use**  **# "wikipedia": Allows searching Wikipedia for factual information**  **# "ddg-search": Allows searching DuckDuckGo for current web information**  **tools = load\_tools(["wikipedia", "ddg-search"])**  **# Create a ReAct agent that can reason and act using the provided tools**  **agent = create\_react\_agent(llm, tools, prompt)**  **# Create an executor to run the agent with verbose output to see reasoning steps**  **agent\_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)**  **# Get user input for the task they want the agent to perform**  **task = input("Assign me Task")**  **# Check if the user provided a task**  **if task:**  **# Execute the agent with the given task**  **# The agent will reason through the problem and use tools as needed**  **response = agent\_executor.invoke({"input": task})**  **# Print the final output/answer from the agent**  **print(response["output"])** |