Table of Contents

[LANGCHAIN 2](#_Toc205543429)

[KEY FEATURES & MODULES 2](#_Toc205543430)

[CONCEPTS 2](#_Toc205543431)

[CHAINING 3](#_Toc205543432)

[THE BUILDING BLOCKS OF LANGCHAIN 3](#_Toc205543433)

[PROMPT TEMPLATE 4](#_Toc205543434)

[CHAT MODELS 4](#_Toc205543435)

# 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.

### THE BUILDING BLOCKS OF LANGCHAIN

**Chains:**Chains in LangChain serve as the foundational concept that unites various AI components, allowing for context-aware responses. A chain represents a sequence of automated actions that begins with the user’s query and ends in the model’s output. For example, developers can employ a chain for tasks such as establishing connections to diverse data sources, generating unique content, translating multiple languages, and responding to user inquiries.

**Links:**Within the LangChain framework, chains are comprised of links, with each connected action forming a link. These links enable developers to decompose complex tasks into smaller, manageable actions. Examples of links include formatting user input, submitting a query to an LLM, accessing data from cloud storage, and translating between languages.

In the LangChain framework, a link receives user input and forwards it to the LangChain libraries for processing. Additionally, LangChain supports the reordering of links to create diverse AI workflows, offering flexibility in constructing chains tailored to specific requirements.

### 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

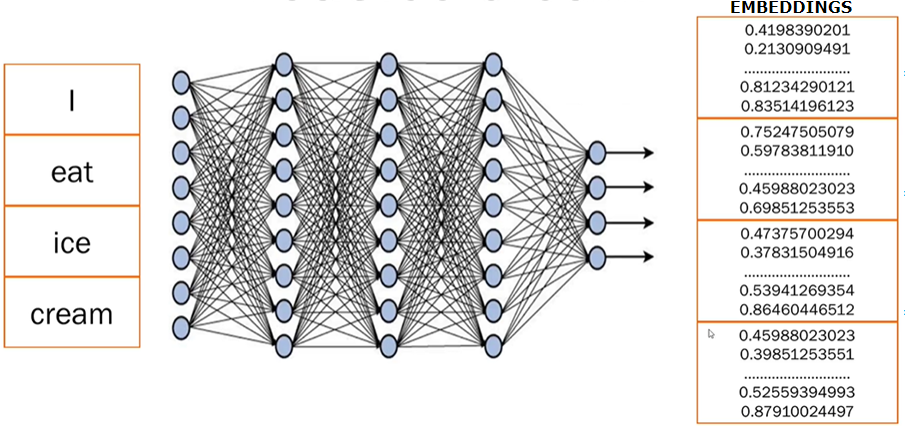
### EMBEDDINGS

* 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?

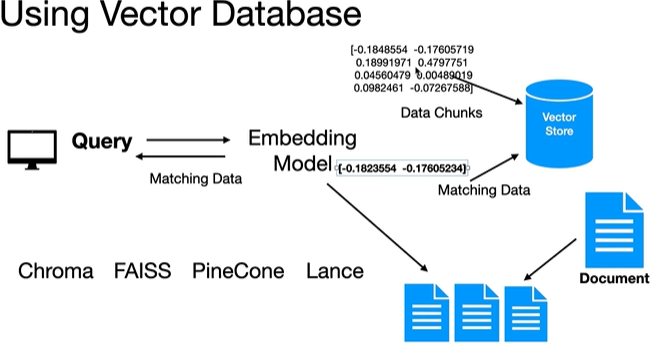
It helps to:

* **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

It is very easy to use all of these using Lang chain.

* Lang chain makes it super easy to run these within memory when you run your program, or you can have
* them on your machine using your file space as well.
* And it is very easy to switch from one vector store to another thanks to lang chain.
* So that process again we take all the documents we want the search to be based off within our organization
* or for a use case.
* We break all the data into small chunks.
* We calculate the embedding for each of those chunks.
* Take that embedding and that chunk.
* Hand it over to the vector store.
* It will store that chunk and the embedding.
* It will do some internal indexing for us.
* And then when the user has a query we take that query, calculate the embedding, give that embedding
* to the vector store.
* The vector store is intelligent enough to find all the similar data chunks for that embedding, which
* match that embedding, and then return all that matching data, which we can then send back to the user
* or the application.