**Main Technical Part :-How it works**

* The **RecursiveCharacterTextSplitter** is first set up to split large text into smaller chunks, but it doesn’t perform any action yet. Only when we call text\_splitter.split\_text(joined\_text) does it actually break the long text into smaller parts. We split text because many models and algorithms, especially embedding models, have **token limits**. Shorter, coherent chunks help maintain context, improve model accuracy, and make processing faster for tasks like search and analysis.
* To convert text into vectors, we use an embedding model. Here, we are using **OpenAI's ada embedding**, commonly accessed via a library like LangChain. This is different from older embeddings like Word2Vec and GloVe from Google. The embedding step starts with tokenization, breaking text like "The cats are running quickly" into smaller pieces (tokens). Transformers use subword tokenization, so "quickly" might become ["quick", "##ly"]. Special tokens like [CLS] and [SEP] are also added for classification or separation.
* ## means "this is a subword continuation."
* [CLS] marks the start of input for classification tasks.
* [SEP] separates sentences or marks the end of input
* In classical **NLP**, steps like **stemming, lemmatization, and stopword** removal were common, but they are skipped in **Transformers** because the models learn to handle language patterns directly. After tokenization, each token is mapped to a unique ID, turning the sentence into a list of integers. These IDs are then passed through an embedding layer, where each **token** gets transformed into a **dense vector** representing its meaning.
* Since **Transformers** don’t inherently understand word order, **positional encoding** is added to these embeddings, allowing the model to know the sequence of words. The vectors with positional information are then processed through Transformer layers using **self-attention mechanism**, capturing complex word relationships like the different meanings of "bank" in "river bank" versus "money bank."
* At the output stage, we get contextual embeddings for each token, which can be used for tasks like **text classification, question answering, or text generation.**
* This whole code setup is part of a **RAG (Retrieval-Augmented Generation)** pipeline. We first split text into overlapping chunks, convert each chunk into OpenAI embeddings, and store them in a **FAISS(Facebook AI Similarity Search)** index for fast similarity search. After generating an embedding vector from a sample text and converting it to a NumPy array, we determine its dimension (like **1536** for OpenAI embeddings). This dimension is critical because FAISS needs to know the size when initializing an **IndexFlatL2** object; otherwise, it throws errors if the vectors don't match the expected size.
* When setting up the retrieval, embedding\_function=embeddings.embed\_query defines how both stored texts and incoming queries are embedded into the same vector space. It’s crucial to use the same embedding model for indexing and searching; otherwise, similarity results will be meaningless.
* The index=index refers to the **FAISS IndexFlatL2** where all vectors are stored. It uses **Euclidean distance (L2)** to compute how close two vectors are. Although FAISS supports more complex index types, IndexFlatL2 is simple and fast, ideal for brute-force similarity search.
* For retrieving the original text after a similarity search, we use **docstore=InMemoryDocstore()**, a simple in-memory storage mapping document IDs to text chunks. This can be replaced with scalable solutions like Redis or cloud databases if needed. Finally, **index\_to\_docstore\_id={}** is a mapping that links FAISS internal vector indices back to their corresponding document IDs, ensuring we can recover the actual content after finding similar vectors.
* This code builds a **Retrieval-Augmented Generation (RAG)** system. Instead of expecting GPT-4 to "know everything," we store external knowledge in a **vector database**. When a user asks a question, we **retrieve** the top 2 most relevant documents (based on vector similarity) and feed them into GPT-4, making answers **accurate**, **context-aware**, and **scalable** for large datasets.
* The **language model** is GPT-4 (via ChatOpenAI) with **low temperature (0.2)** for focused, less random outputs, and **max\_tokens=150** to keep responses sharp and cost-effective. The **retriever** is built from the vectorstore.as\_retriever(), set to pull **k=2** relevant chunks for each query. This retrieval uses **vector distance** metrics (like cosine similarity).
* We connect everything using RetrievalQA.from\_chain\_type().
* chain\_type="stuff" simply **stuffs** retrieved documents and the query together into the prompt.
* **Memory** is used so the conversation feels **continuous**, remembering previous interactions.
* return\_source\_documents=False keeps answers clean (no raw docs shown).
* Finally, qa\_chain.run(query) ties it all together: retrieves docs, merges memory + query + docs, sends them to GPT-4, and returns a **grounded** response.
* **Note:**
* "Stuff" is simple but best when the total input is **small** enough for the model’s context window.
* For bigger datasets, consider smarter chain types like "map\_reduce" or "refine".
* This setup gives you a system that answers questions **smartly**, **accurately**, and **based on real data** — not just model memory.

**Question comes up in your mind!**

**✅ What does the embedding model do?**

The embedding model (like OpenAI's text-embedding-ada-002) is responsible for **turning raw text into vectors**. But **it doesn't track which exact document or chunk the vector came from** — it just creates the vector representation.

For example:

* The model takes in text like "FAISS is a vector search library used for similarity search."
* It outputs a vector (say, a 1536-dimensional array) that captures the **semantic meaning** of that text.

But here's the key point: **The model doesn’t store any metadata about the text** (e.g., document IDs or original content). It only provides the vector representation for that text.

**🧠 Why the embedding model doesn't handle this:**

Embedding models are **concerned only with converting text to vectors**. They don't need to track where the text came from or manage your document storage. That’s a separate concern handled by FAISS and your docstore system.

Think of it like:

* The embedding model is **the brain** (creating vectors).
* FAISS and the docstore are the **filing cabinet** (storing and retrieving documents).
* index\_to\_docstore\_id is like the **index card** that connects the brain's output to the physical document in the cabinet

**🧩 Where index\_to\_docstore\_id comes in**

Now, you need a **system** to:

1. **Store** that vector (which FAISS does).
2. **Connect it back to the original document**.

This is where **you** (or the higher-level framework) need the index\_to\_docstore\_id. FAISS doesn't automatically know where that vector came from — it only stores the vector. You need to manage the **relationship between vectors and documents**.

**🛠️ How it fits together:**

* **Embedding Model**: Converts text to vectors.
  + **Doesn't know** where the text came from or how to reference it later.
* **FAISS**: Stores the vectors and allows for efficient similarity search.
  + **Doesn't know** what text is behind each vector either.
* **index\_to\_docstore\_id**: Maps each vector’s index (e.g., 0, 1, 2, ...) to a **unique ID** for the document (e.g., "doc\_abc123").
  + **You manage this mapping** to tie vectors to actual text in a docstore.

**✅ TL;DR: (Too Long; Didn't Read)**

* The **embedding model** does **not** track where the text came from.
* **You** need to map **vectors to original documents** using index\_to\_docstore\_id.
* **FAISS** stores vectors for fast search, but doesn’t have metadata — you manage that separately.