RAG – Langchain workshop

BY SHAHZAIB HAMZA

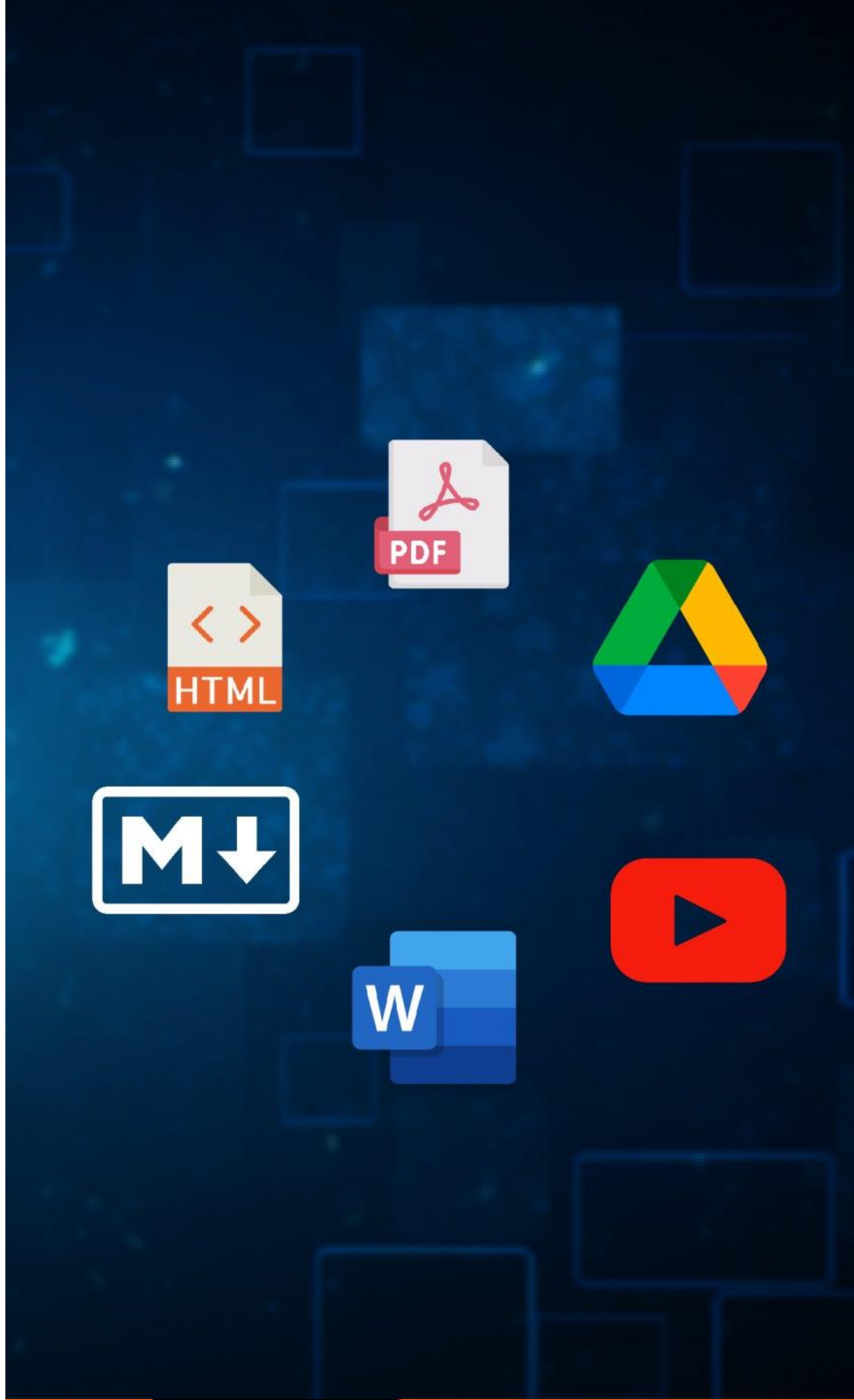
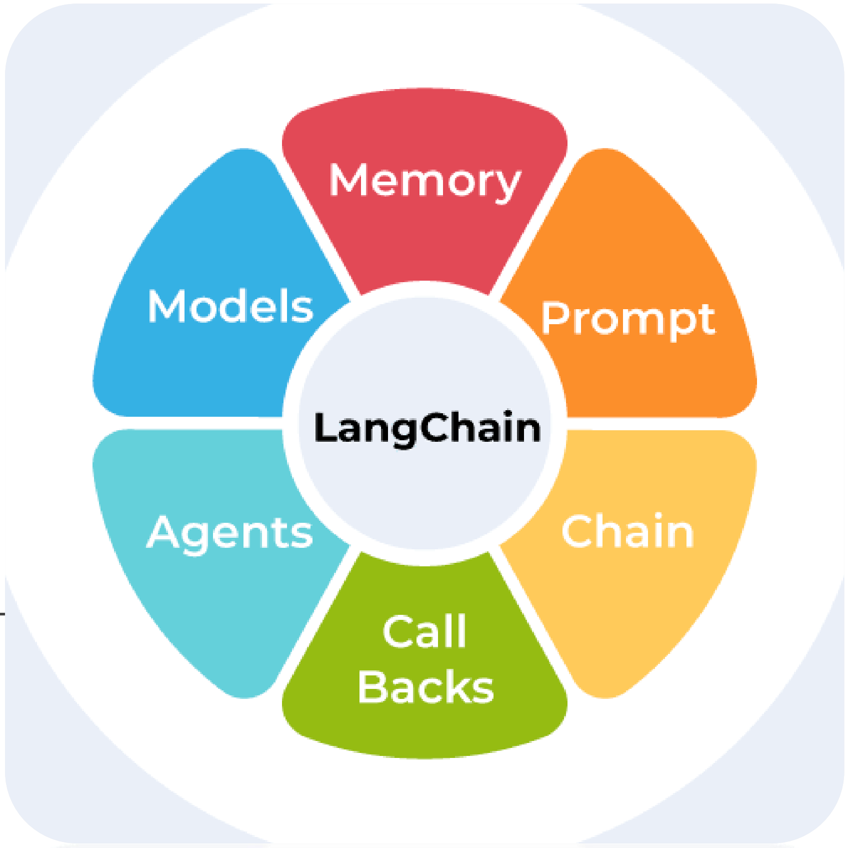


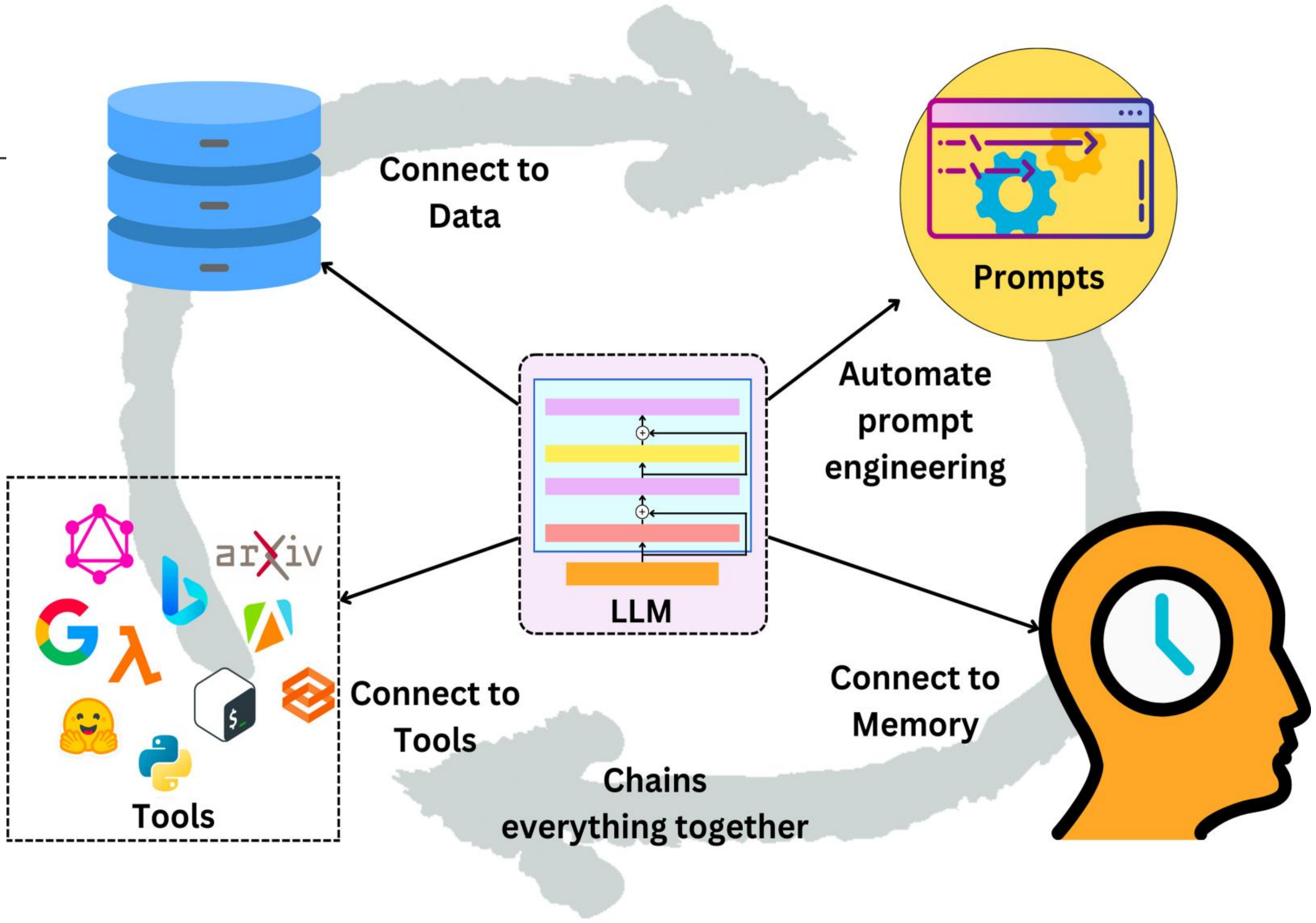
Table of Contents

LANGCHAIN



Langchain

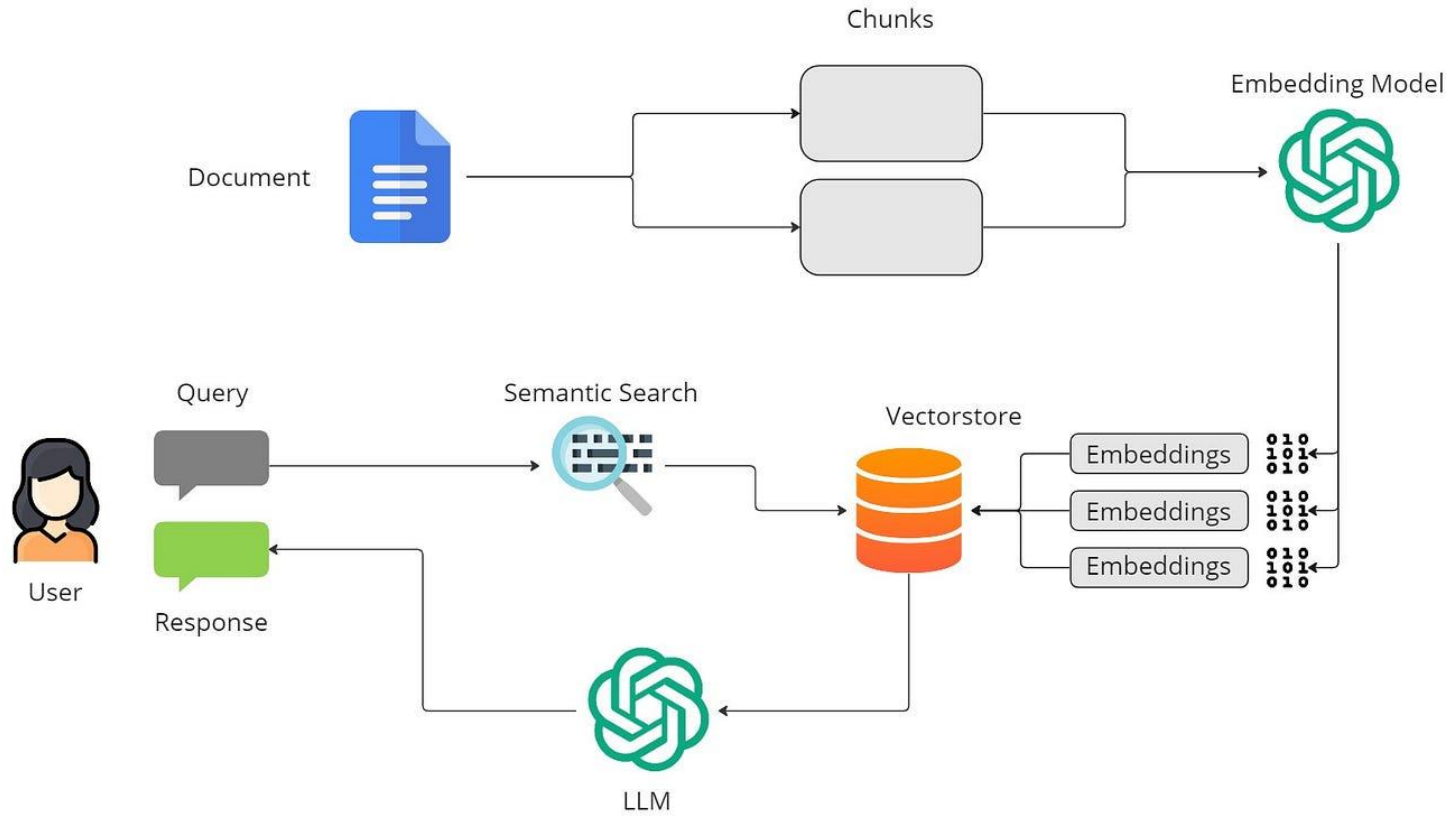
LangChain is an open-source framework that helps developers connect Large Language Models (LLMs) to external data and real-world actions. It works by standardizing and linking several core components:



Component	What it is	Role in Your System
LLMs/Chat Models 🧠	The brain (e.g., Gemini, GPT).	Generates the final, natural language response.
Prompt Templates 📜	Standardized inputs for the LLM.	Define the instructions and contain variables for dynamic content (like the user's question or the retrieved data).
Indexes/Retrievers 🔍	Tools to connect to and search external data.	Manages the preparation and searching of your company's documents.
Chains 🔗	A predefined sequence of steps or calls.	Defines the workflow: 1) Get data 2) Build Prompt 3) Call LLM.
Agents 🤖	LLMs that can use tools and reason about which step to take next.	Go beyond a fixed Chain by dynamically deciding if they need to search a database, call an API, or just answer directly.

RAG

RETRIEVAL-AUGMENTED GENERATION


Chunks
Embedding Model
Document
Query
Semantic Search
Vectorstore
Embeddings
Embeddings
Embeddings
User
Response
LLM

RAG

RAG is a technique that gives a Large Language Model (LLM) the ability to talk about **specific, external information,** like your company's product manuals or return policies, that it was **not** trained on. It essentially acts as a smart "fact-checker" before the LLM generates an answer.

RAG is the technique that uses LangChain components to make your LLM specific to your company's data, which solves the problem of hallucination and generic responses.

The process is split into two distinct phases: **Preparation (Indexing)** and **Execution (Retrieval & Generation).**

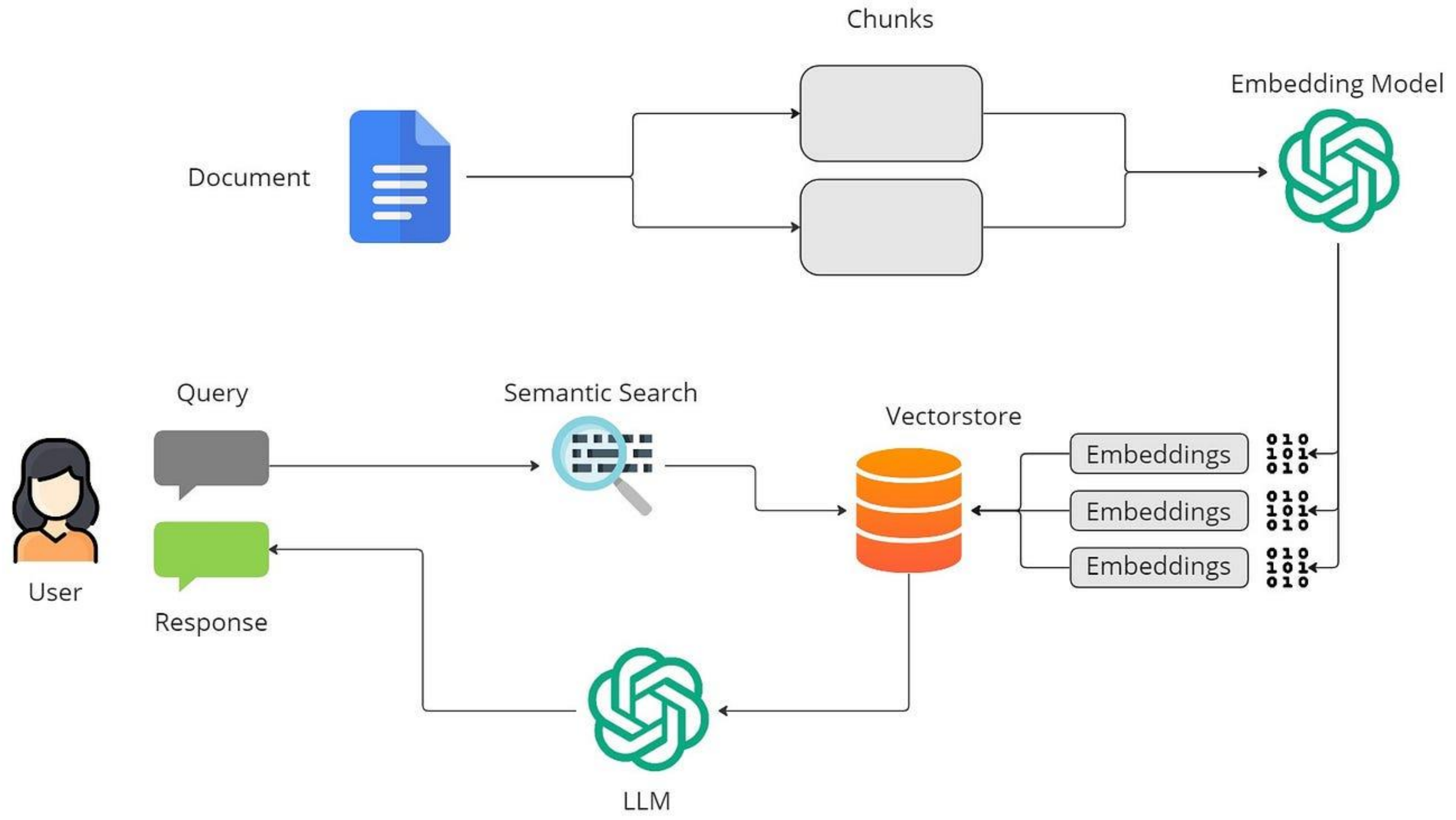
Indexing

This is done once, before any customer asks a question, to organize your private data for fast searching.

1. **Load Data:** Load all your source documents (PDFs, manuals, support pages).
2. **Chunking:** The critical step. Large documents are split into smaller, meaningful chunks (e.g., 500-1000 characters each). This is essential because embeddings work best on focused, small pieces of text.
3. **Embedding:** Each chunk is passed through an **Embedding Model** that converts the text into a long list of numbers called a **Vector**. Texts with similar meaning have numerically close vectors.
4. **Storage (Vector Store):** These vectors (and their corresponding original text chunks) are saved in a specialized database called a **Vector Store** (e.g., Chroma, Pinecone).

Execution

Step	Action	LangChain Component Used
1. User Query	A customer asks, "How do I return a product?"	User Input
2. Retrieval	The query is embedded (turned into a vector) and the Vector Store is searched for the closest matching vectors. The original, relevant text chunks from your documents are retrieved.	Retriever
3. Augmentation	The retrieved chunks are inserted into the Prompt Template as "Context."	Prompt Template
4. Generation	The final, powerful prompt (Context + Instructions + Query) is sent to the LLM. The LLM is now grounded and answers only based on the facts provided in the Context.	LLM/Chat Model
5. Output	The specific, accurate answer is returned to the customer.	Chain


Chunks
Embedding Model
Document
Query
Semantic Search
Vectorstore
Embeddings
Embeddings
Embeddings
User
Response
LLM

What are vector Embeddings & Database?

Embeddings are like *smart numbers* that represent meaning.

They turn text (like a sentence, paragraph, or document) into a list of numbers called a **vector.**

The idea is that **similar meanings** have **similar numbers.**
For example:

- “cat” → [0.1, 0.8, 0.3, ...]
- “dog” → [0.12, 0.75, 0.35, ...]
- “car” → [0.9, 0.2, 0.1, ...]
  Here, “cat” and “dog” vectors are *closer together* than “cat” and “car”.

So, embeddings help computers understand the *meaning* of text instead of just the exact words.

Vector Database

A **vector database** stores **embeddings** number lists that represent the *meaning* of text.

When you ask something:

Your question is turned into a vector (numbers).

The database finds other vectors that are **most similar in meaning.**

Those matching texts are sent to the AI to help answer your question.

“**LangChain:** A tool that helps connect LLMs (like ChatGPT) with data, APIs, and logic to build AI apps.”

“RAGChain: A framework that helps AI **find and use relevant information** before giving an answer making responses more accurate and factual.”

Implementation

RAG CHAIN

Implementation of RAG Pipeline

Summary:

1. User uploads files or pastes URL
2. Text is extracted and chunked
3. Chunks converted to embeddings (sentence-transformers)
4. Embeddings stored in FAISS vector DB
5. Query → embedding → nearest chunks retrieved
6. LLM generates answer using retrieved chunks
7. Streamlit provides the frontend

Architecture

1. **UI (Streamlit)** — simple two-column layout: left for upload/paste URL, right for chat/result.
2. **Ingestion** — user uploads PDF/DOCX/TXT or pastes URL. We extract text.
3. **Chunking** — split extracted text into overlapping chunks using a text splitter.
4. **Embedding** — create vector embeddings for chunks using a free sentence-transformer model.
5. **Vector store** — save embeddings to FAISS on disk.
6. **Retrieval** — when user asks a question, retrieve top-k relevant chunks.
7. **LLM** — construct a prompt with retrieved context and call Gemini API to generate the final answer.

Building a Retrieval-Augmented Generation app where users upload a document or paste a link in a simple Streamlit UI; documents are embedded, stored in FAISS, and queries are answered by a LangChain retrieval chain that calls the Gemini API for fluent answers.

Step 1: Creating project folder, setting environment and installing dependencies

```
# create a folder for your project

mkdir rag_langchain_project


# go inside the folder

cd rag_langchain_project


# Create virtual environment

python -m venv rag_env

source rag_env/Scripts/activate
```

```
#upgrade pip
pip install --upgrade pip

#install required libraries
pip install langchain sentence-transformers faiss-cpu streamlit
requests beautifulsoup4 PyPDF2 gpt4all
```

What each does:

- `langchain` → RAG tools & chain management
- `sentence-transformers` → embeddings model
- `faiss-cpu` → vector database for similarity search
- `streamlit` → web app UI
- `requests` & `beautifulsoup4` → fetch + clean text from websites
- `PyPDF2` → extract text from PDFs

File layout

```
rag-gemini/
├─ streamlit_app.py
├─ rag_utils.py
├─ gemini_client.py
├─ requirements.txt
├─ data/             # where uploaded files & faiss index will be stored
│   ├─ index.faiss
│   └─ docs/
└─ README.md
```

`requirements.txt`

```
streamlit>=1.20
langchain>=0.0.200
faiss-cpu
sentence-transformers>=2.2.2
python-multipart
requests
beautifulsoup4
readability-lxml
pdfplumber
python-docx
tqdm
pydantic
typing_extensions
pytest  # optional for tests
```

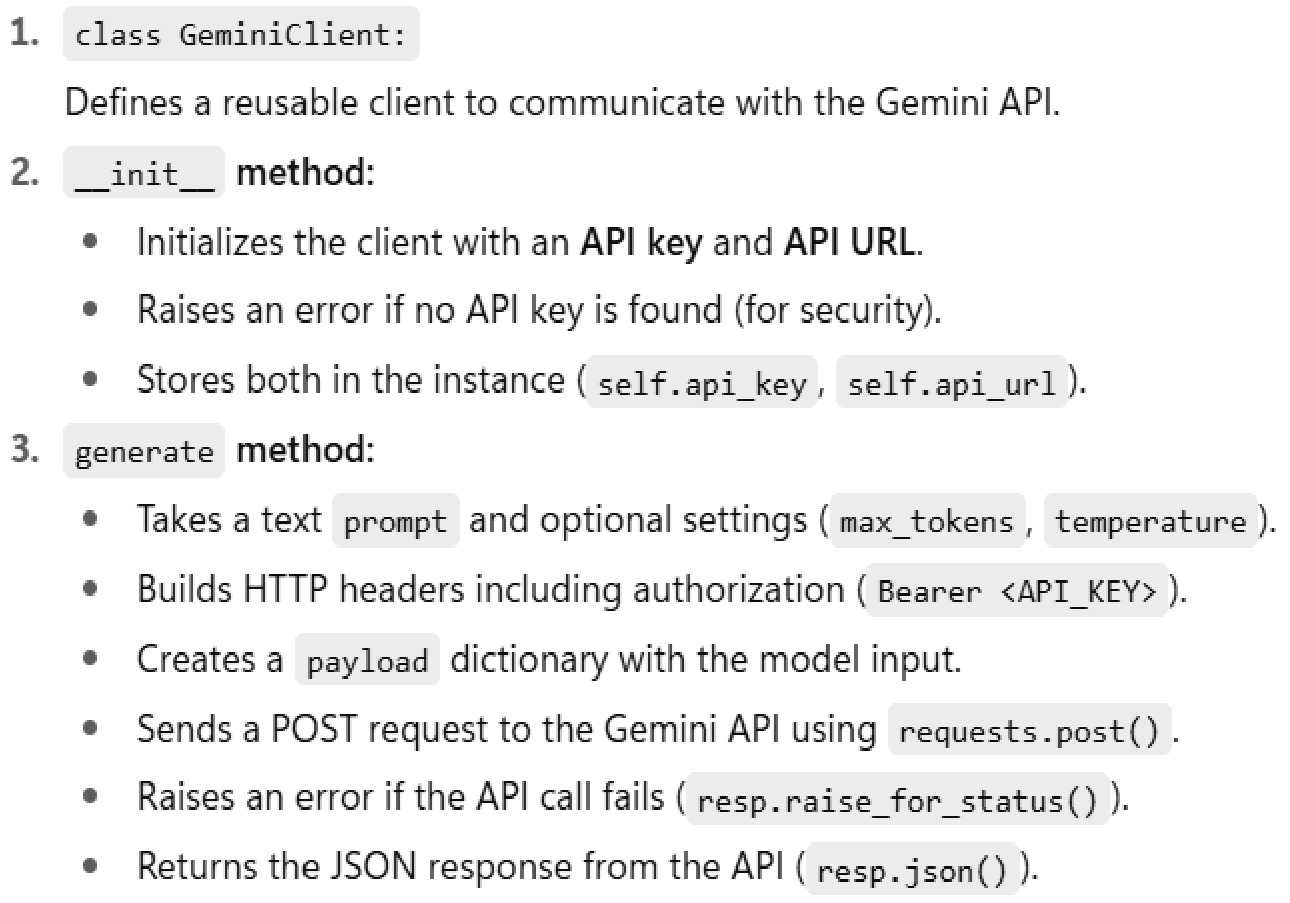
If `faiss-cpu` fails on some platforms, install `faiss-cpu` from conda or use `pip install faiss-cpu --no-binary :all:` depending on your system.

Gemini_client.py

```
import os

import requests

from typing import Dict, Any

GEMINI_API_KEY =
os.getenv("GEMINI_API_KEY")

GEMINI_API_URL =
os.getenv("GEMINI_API_URL",
"https://api.gemini.example/v1/generate")
```

- **import os** → allows access to environment variables and system-level operations.
- **import requests** → lets you make HTTP requests to APIs (like Gemini).
- **from typing import Dict, Any** → provides type hints; used later for defining data structures (e.g., request or response types).
- **GEMINI_API_KEY = os.getenv("GEMINI_API_KEY")** → reads the API key stored in your environment variables for security (so it's not hard-coded).
- **GEMINI_API_URL = os.getenv("GEMINI_API_URL", "https://api.gemini.example/v1/generate")** → reads the API URL from environment variables; if it's not set, it defaults to the given URL string.

```
class GeminiClient:
    def __init__(self, api_key: str = GEMINI_API_KEY, api_ur
str = GEMINI_API_URL):
        if not api_key:
            raise ValueError("Set GEMINI_API_KEY environmen
variable")
        self.api_key = api_key
        self.api_url = api_url

    def generate(self, prompt: str, max_tokens: int = 512,
temperature: float = 0.0) -> Dict[str, Any]:
        headers = {
            "Authorization": f"Bearer {self.api_key}",
            "Content-Type": "application/json",
        }
        payload = {
            "prompt": prompt,
            "max_tokens": max_tokens,
            "temperature": temperature,
        }
        resp = requests.post(self.api_url, json=payload,
headers=headers, timeout=30)
        resp.raise_for_status()
        return resp.json()
```



1. `class GeminiClient:`

   Defines a reusable client to communicate with the Gemini API.

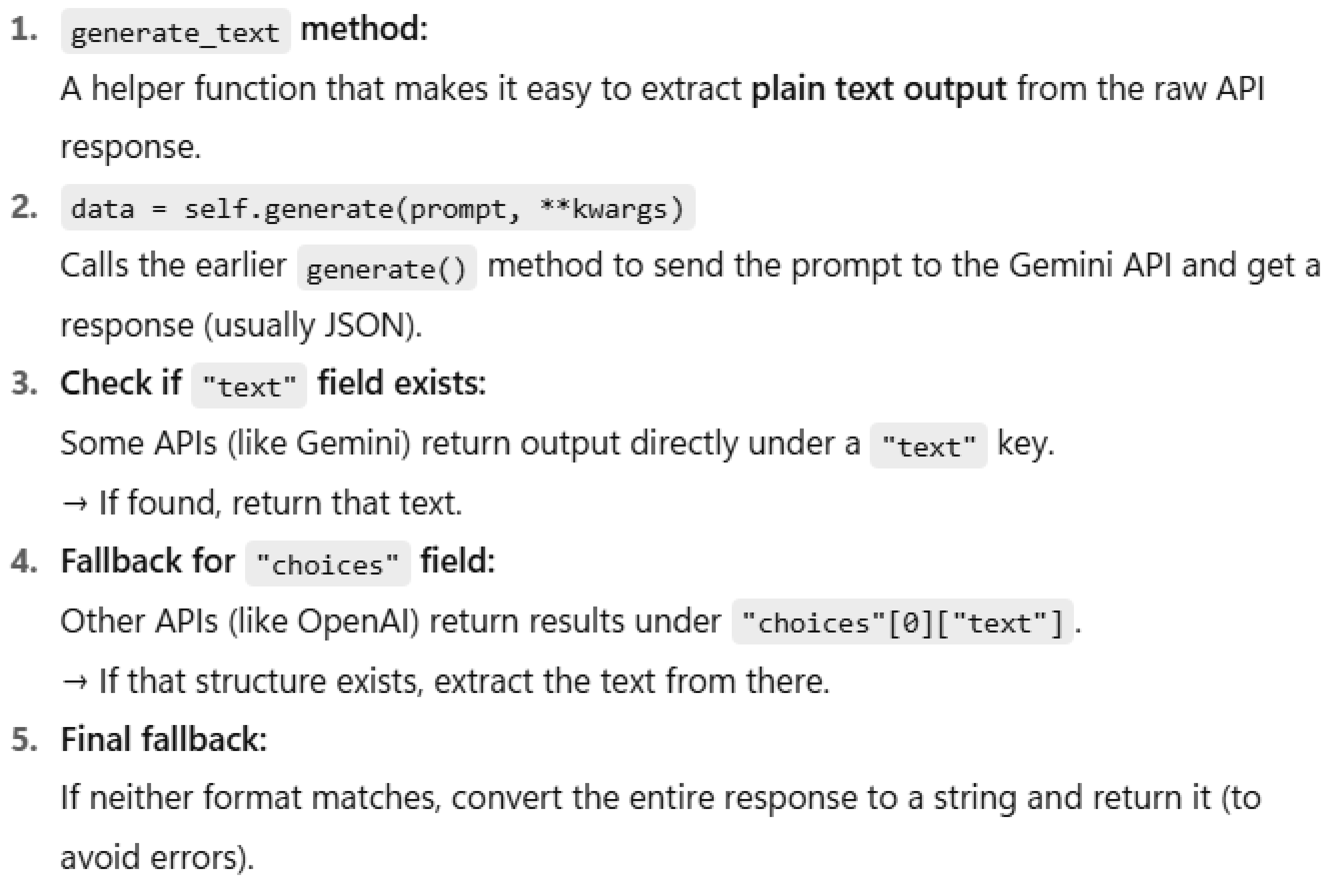
2. `__init__` **method:**

   - Initializes the client with an **API key** and **API URL**.
   - Raises an error if no API key is found (for security).
   - Stores both in the instance (`self.api_key`, `self.api_url`).

3. `generate` **method:**

   - Takes a text `prompt` and optional settings (`max_tokens`, `temperature`).
   - Builds HTTP headers including authorization (`Bearer <API_KEY>`).
   - Creates a `payload` dictionary with the model input.
   - Sends a POST request to the Gemini API using `requests.post()`.
   - Raises an error if the API call fails (`resp.raise_for_status()`).
   - Returns the JSON response from the API (`resp.json()`).

```
def generate_text(self, prompt: str, kwargs) -> str:
    data = self.generate(prompt, kwargs)
    # adjust this depending on the provider's response schema
    if isinstance(data, dict) and "text" in data:
        return data["text"]
    # fallback - try common fields
    if isinstance(data, dict) and "choices" in data and len(data["choices"]) > 0:
        return data["choices"][0].get("text", "")
    return str(data)
```



1. `generate_text` **method:**

   A helper function that makes it easy to extract **plain text output** from the raw API response.

2. `data = self.generate(prompt, **kwargs)`

   Calls the earlier `generate()` method to send the prompt to the Gemini API and get a response (usually JSON).

3. **Check if `"text"` field exists:**

   Some APIs (like Gemini) return output directly under a `"text"` key.

   → If found, return that text.

4. **Fallback for `"choices"` field:**

   Other APIs (like OpenAI) return results under `"choices"[0]["text"]`.

   → If that structure exists, extract the text from there.

5. **Final fallback:**

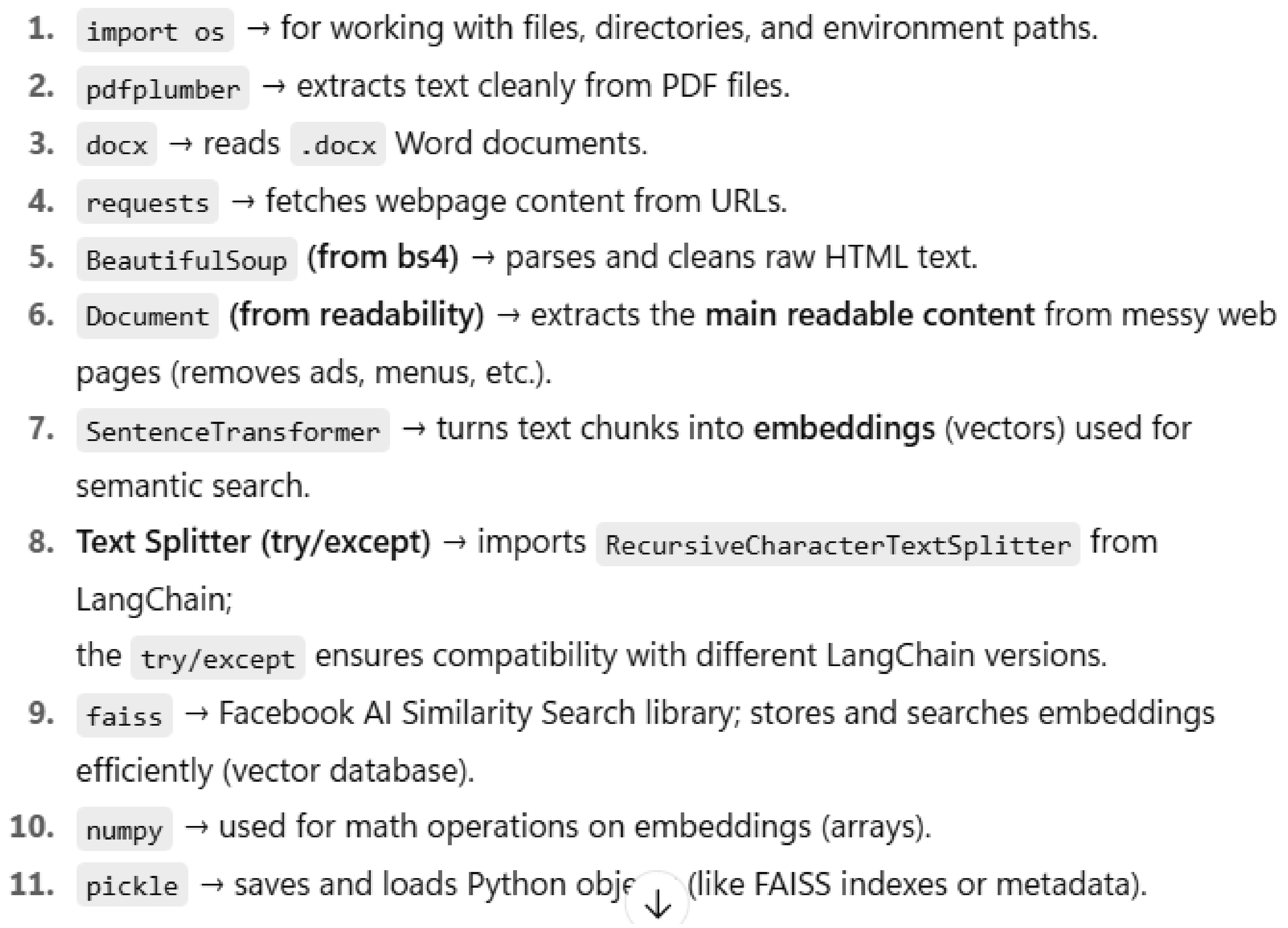
   If neither format matches, convert the entire response to a string and return it (to avoid errors).

Rag_utils.py

Link to the reference code:

https://github.com/shahzaibAmeerHamza/Langchain_ai_chatbot_rag

Explaination



1. `import os` → for working with files, directories, and environment paths.
2. `pdfplumber` → extracts text cleanly from PDF files.
3. `docx` → reads `.docx` Word documents.
4. `requests` → fetches webpage content from URLs.
5. `BeautifulSoup` **(from bs4)** → parses and cleans raw HTML text.
6. `Document` **(from readability)** → extracts the **main readable content** from messy web pages (removes ads, menus, etc.).
7. `SentenceTransformer` → turns text chunks into **embeddings** (vectors) used for semantic search.
8. **Text Splitter (try/except)** → imports `RecursiveCharacterTextSplitter` from LangChain;
   the `try/except` ensures compatibility with different LangChain versions.
9. `faiss` → Facebook AI Similarity Search library; stores and searches embeddings efficiently (vector database).
10. `numpy` → used for math operations on embeddings (arrays).
11. `pickle` → saves and loads Python obje (like FAISS indexes or metadata).

EMBEDDING_MODEL_NAME = "all-MiniLM-L6-v2"

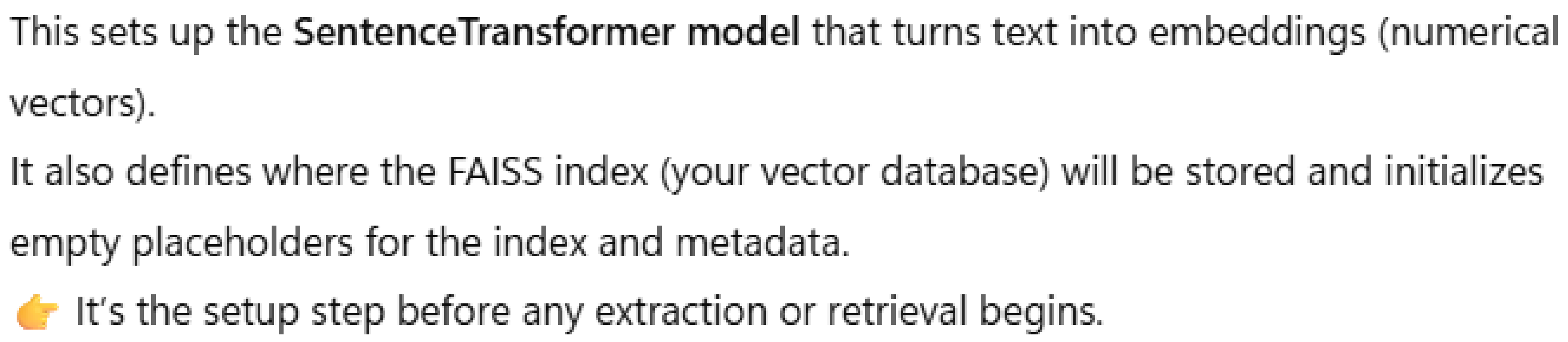
EMBEDDINGS_DIM = 384

Uses a **Sentence Transformer model** to convert text → embeddings (vectors of 384 dimensions).

This model is lightweight and works great for RAG tasks.

Init – initialize the ingestor

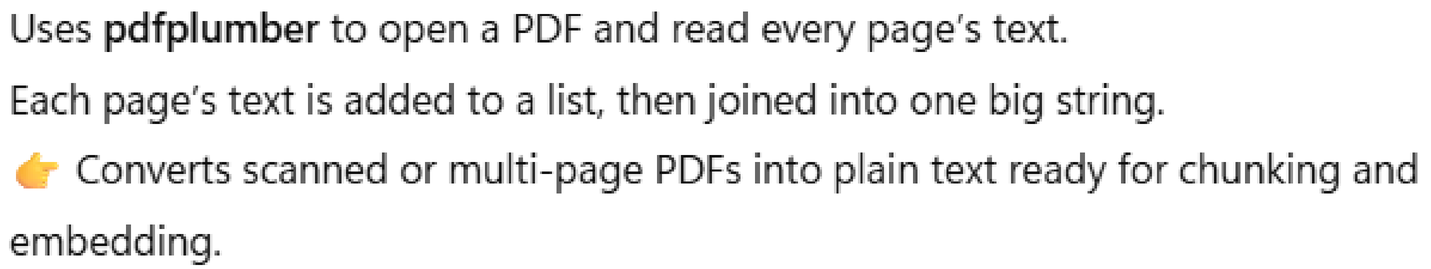
```
def __init__(self, model_name: str = EMBEDDING_MODEL_NAME, index_path: str = "data/index.faiss"):
    self.model = SentenceTransformer(model_name)
    self.index_path = index_path
    self.index = None
    self.metadata = []
```



This sets up the **SentenceTransformer model** that turns text into embeddings (numerical vectors).
It also defines where the FAISS index (your vector database) will be stored and initializes empty placeholders for the index and metadata.
👉 It's the setup step before any extraction or retrieval begins.

Extract text

```
def extract_text_from_pdf(self, path: str) -> str:
    texts = []
    with pdfplumber.open(path) as pdf:
        for page in pdf.pages:
            texts.append(page.extract_text() or "")
    return "\n".join(texts)
```



Uses **pdfplumber** to open a PDF and read every page's text.

Each page's text is added to a list, then joined into one big string.

👉 Converts scanned or multi-page PDFs into plain text ready for chunking and embedding.

Fetch URL

```
def fetch_url_text(self, url: str) -> str:
    r = requests.get(url, timeout=15)
    doc = Document(r.text)
    html = doc.summary()
    soup = BeautifulSoup(html, "html.parser")
    return soup.get_text(separator="\n")
```

Fetches a webpage with **requests**, uses **readability** to extract only the main article (not ads or menus), and cleans it with **BeautifulSoup** to get plain text.

👉 Ideal for pulling and cleaning text from blogs, news articles, or any website.

Chunk test

Uses **LangChain's text splitter** to break long documents into small, overlapping chunks.
Each chunk shares a bit of text with the next to maintain context.
This improves retrieval accuracy and prevents LLMs from losing meaning when answering.

```
def chunk_text(self, text: str, chunk_size: int = 800, chunk_overlap: int = 200):
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap,
        separators=["\n\n", "\n", ".", " "]
    )
```

Embed_text – Converts texts to embeddings

Uses the SentenceTransformer model to turn every text chunk into a numerical vector (embedding).
This is how the system understands and compares the *meaning* of text.

```
def embed_texts(self, texts):
    return self.model.encode(texts, show_progress_bar=True, convert_to_numpy=True)
```

Create_or_load_index (prepare FAISS Database)

If a FAISS index file already exists, it loads it along with its metadata (text info).
Otherwise, it creates a new, empty FAISS index.
This ensures your vector database is ready for adding or searching embeddings.

add_texts - Add New Texts to the Index

Creates embeddings for the new text chunks, adds them to the FAISS index, and saves everything (both index and metadata).
 This step grows your searchable database whenever you upload or ingest new documents.

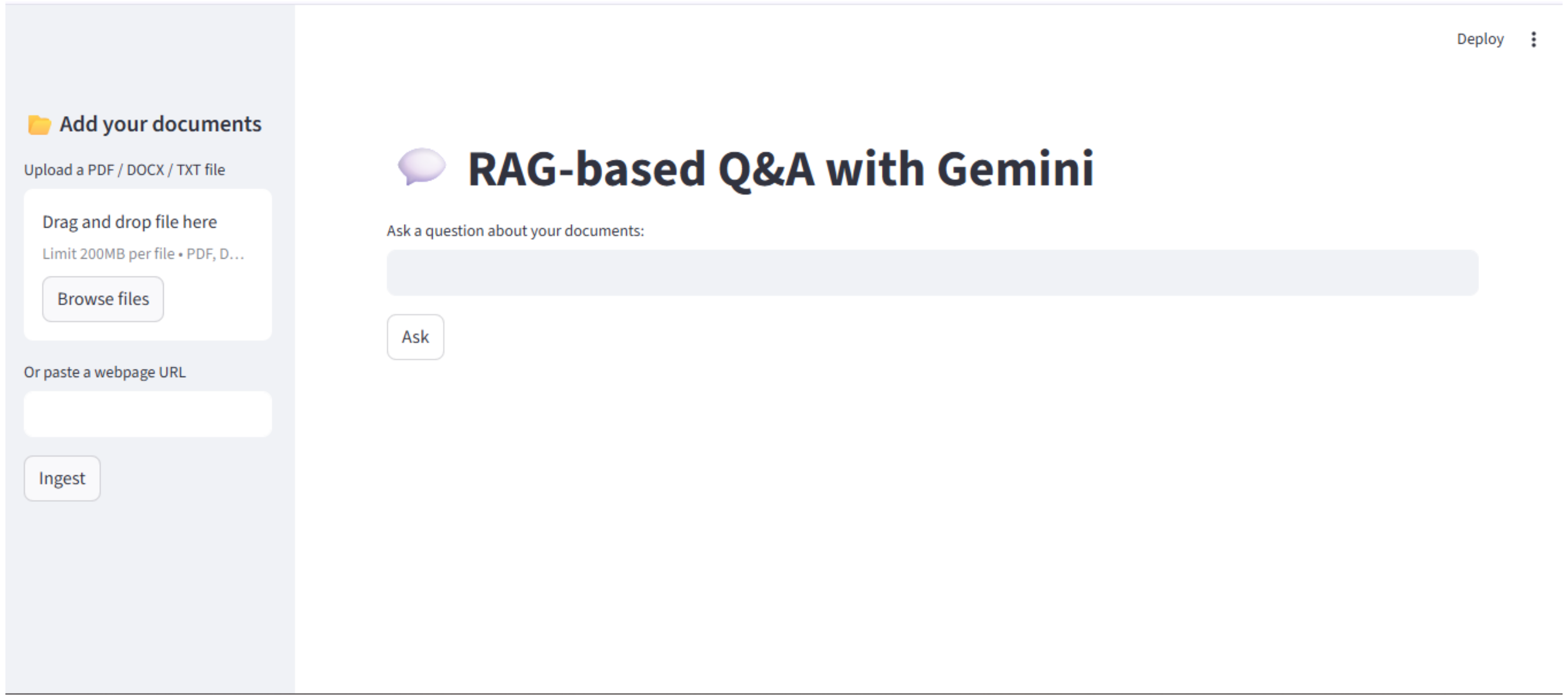
query - Search for Relevant Chunks

Converts the user’s query into an embedding and searches for the k most similar vectors in FAISS.
Then it returns the metadata (text chunks) for those results.
This is how your app *retrieves the most relevant information* before sending it to the LLM for generation.

```
def query(self, query_text, k: int = 5):
    q_emb = self.model.encode([query_text], convert_to_numpy=True)
    D, I = self.index.search(q_emb, k)
    results = [self.metadata[idx] for idx in I[0] if idx < len(self.metadata)]
    return results
```

Creating UI

https://github.com/shahzaibAmeerHamza/Langchain_ai_chatbot_rag


Deploy
Add your documents
Upload a PDF / DOCX / TXT file
Drag and drop file here
Limit 200MB per file • PDF, D...
Browse files
Or paste a webpage URL
Ingest
RAG-based Q&A with Gemini
Ask a question about your documents:
Ask

Environment variables & running locally

```
export GEMINI_API_KEY="your_key_here"
export GEMINI_API_URL="https://api.gemini.example/v1/generate"  # if needed

pip install -r requirements.txt
streamlit run streamlit_app.py
```

The app will run at `http://localhost:8501` by default.

THANK YOU