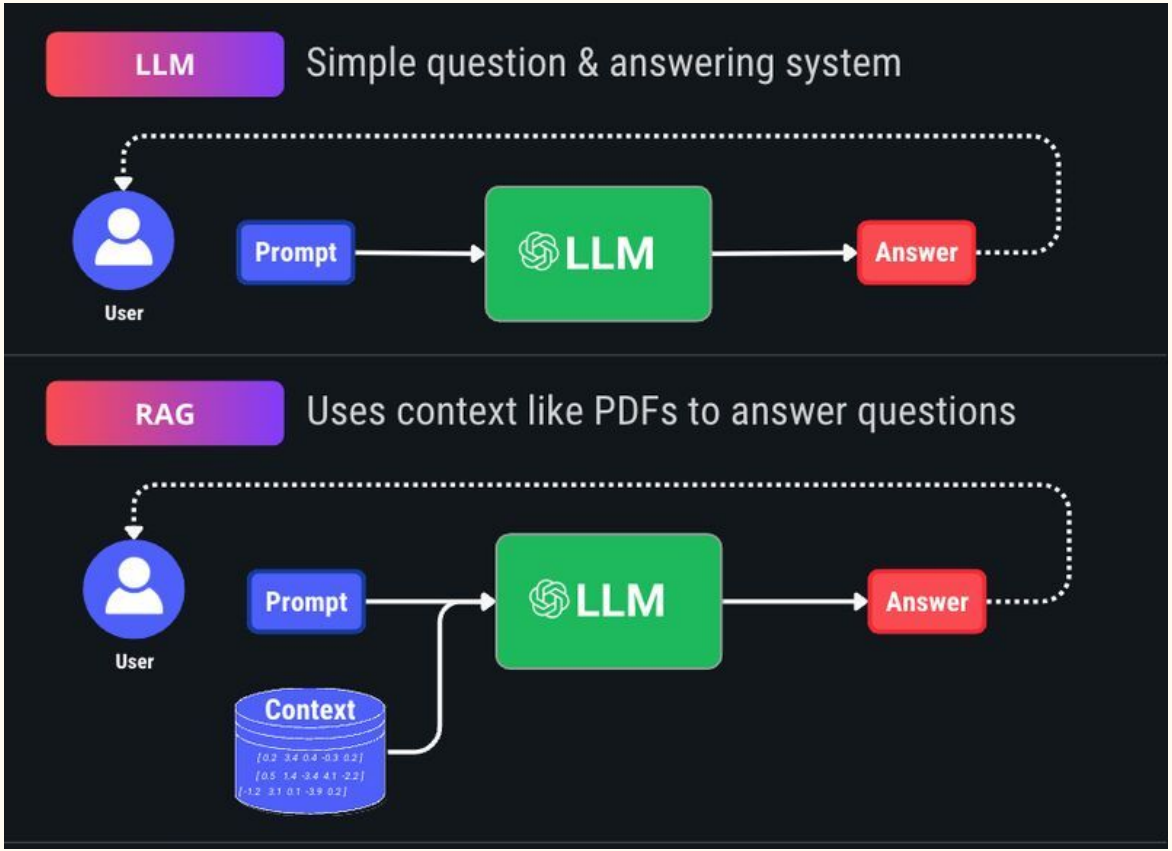
RAG PIPELINE

What is RAG?

- **RAG** can stand for **Retrieval-Augmented Generation**, an AI framework that improves large language models (LLMs) by allowing them to access and cite external knowledge sources.
- How it works: Instead of relying solely on its training data, the LLM first retrieves relevant information from a specified set of documents or databases and then uses that new information to create a more accurate and up-to-date answer.

Why RAG?


LLM
Simple question & answering system
User
Prompt
LLM
Answer
RAG
Uses context like PDFs to answer questions
User
Prompt
LLM
Answer
Context

RAG Pipeline

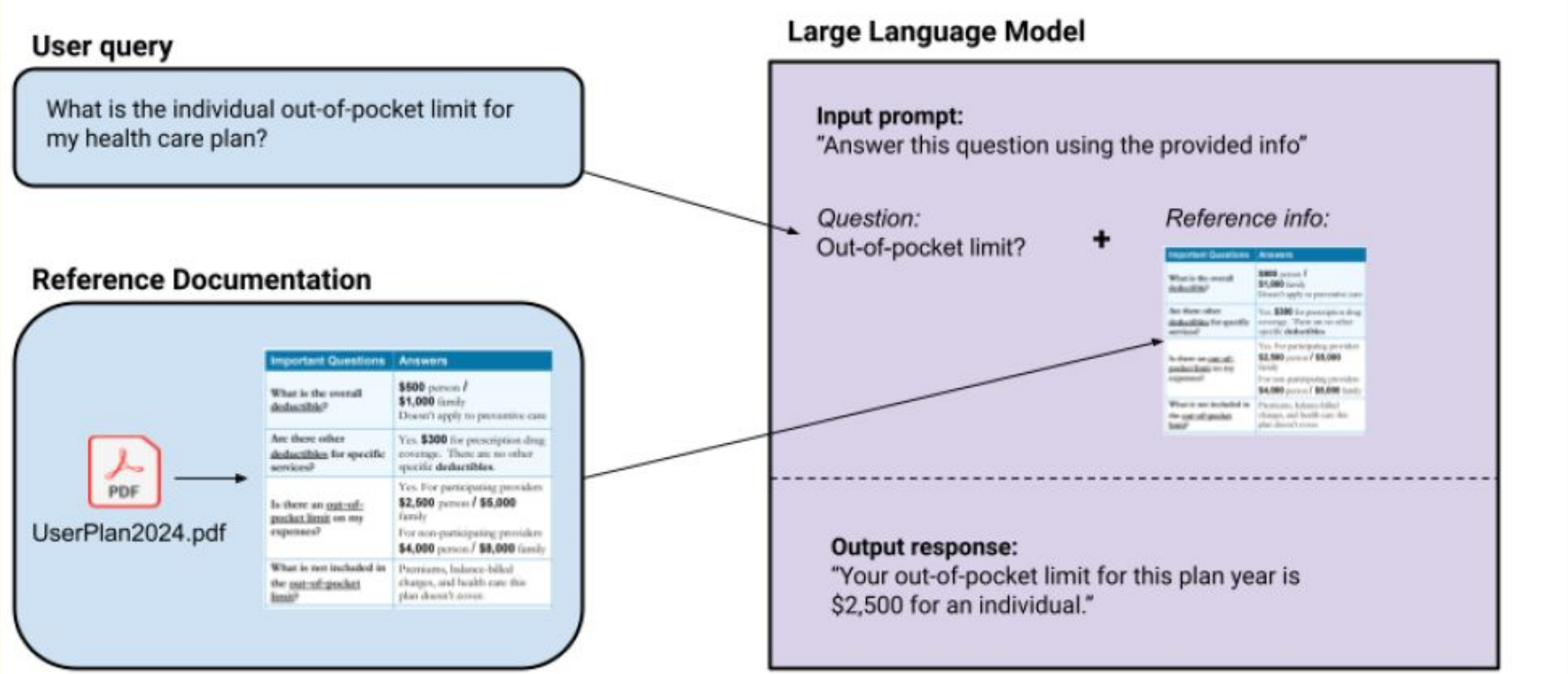
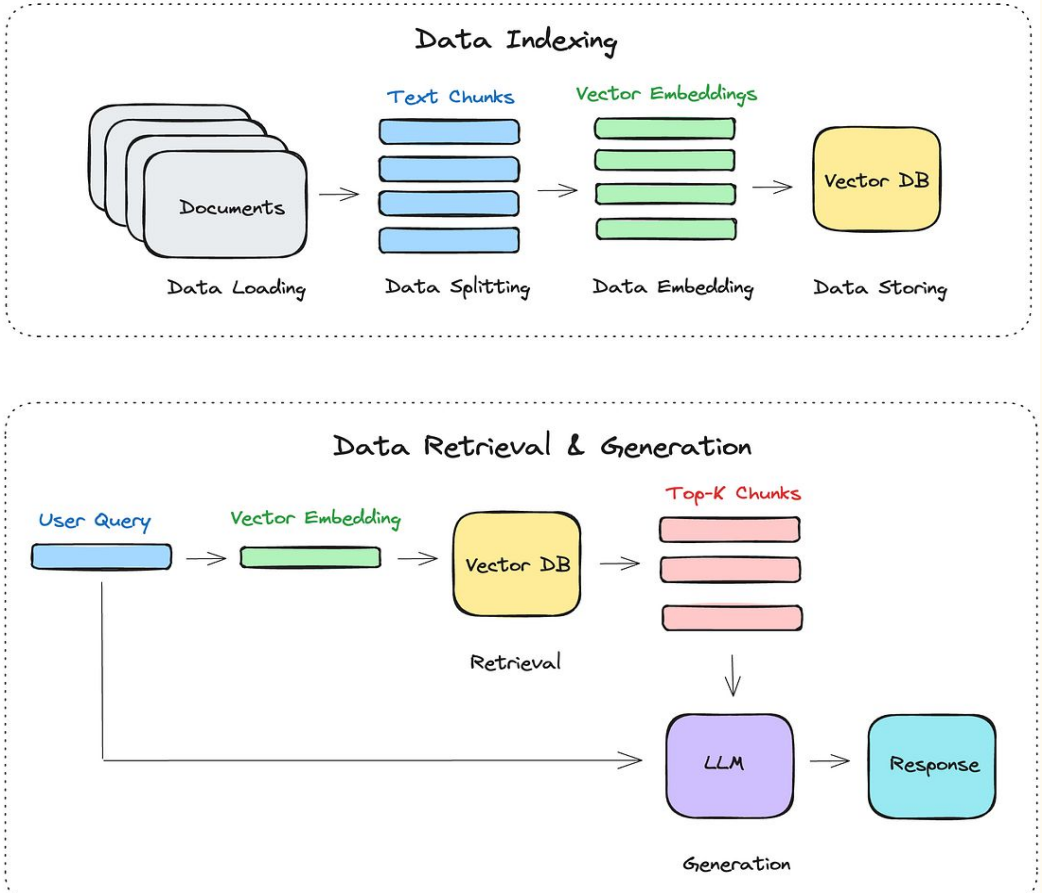


Figure 1. A basic RAG example where reference information is used to help a user's question.

RAG Pipeline



RAG Pipeline - Vector Embeddings

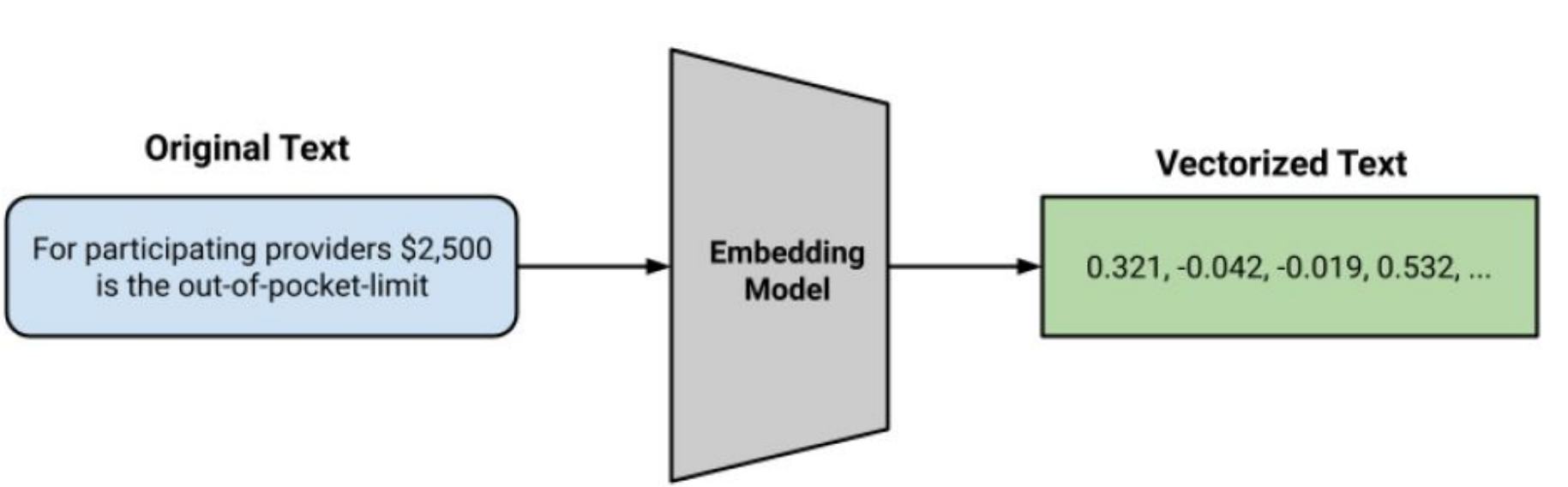


Figure 3. Embedding models convert text into high-dimensional numerical vectors.

RAG Pipeline - Top K chunks

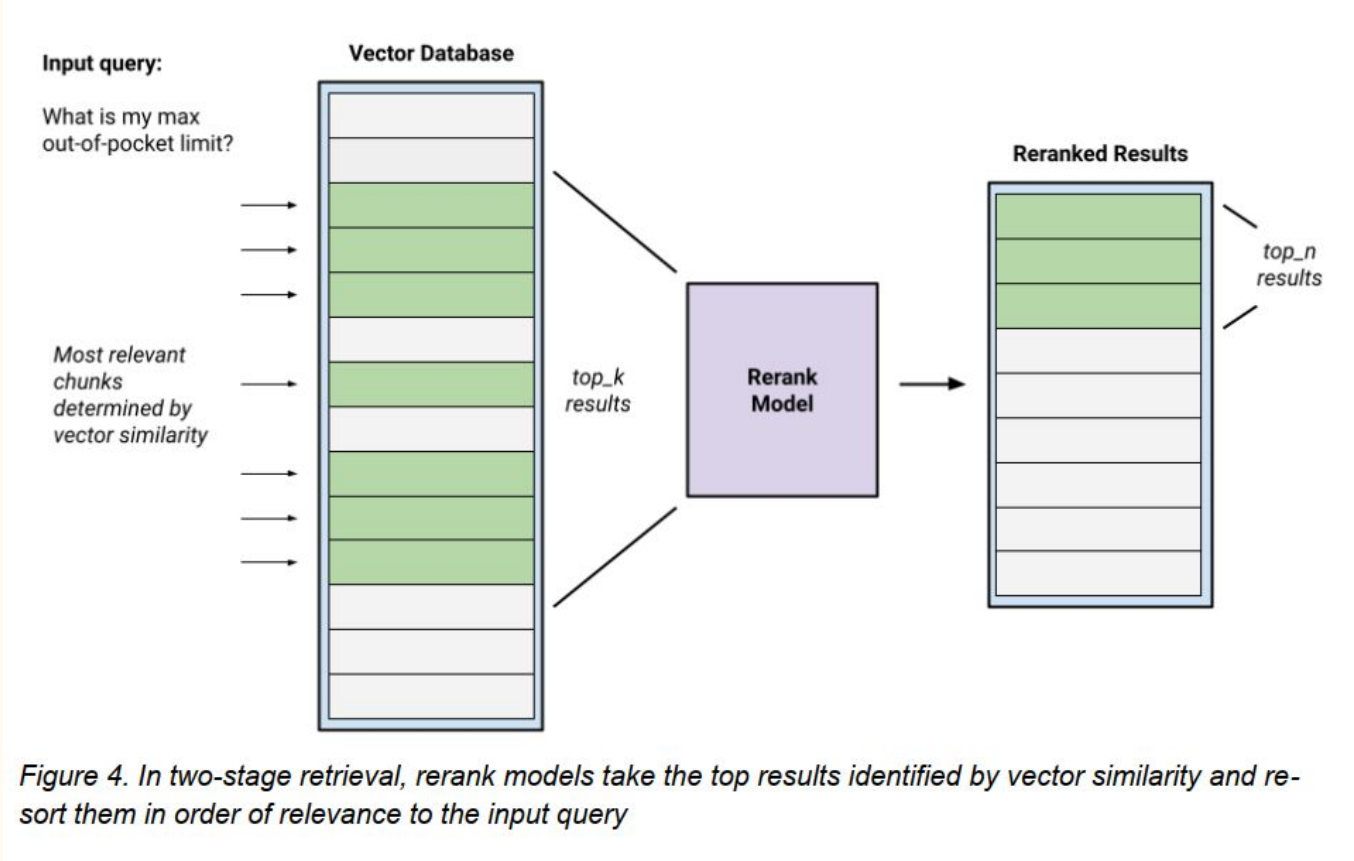


Figure 4. In two-stage retrieval, rerank models take the top results identified by vector similarity and re-sort them in order of relevance to the input query

Installations

- `langchain` – Helps your program *talk* to AI models like ChatGPT and connect them to other tools.
- `langchain-community` – Adds *extra helpers* made by the community.
- `langchain-openai` – Lets LangChain work with *OpenAI models* (like ChatGPT).
- `faiss-cpu` – Helps the computer *remember and find information fast*, like searching in a big pile of notes.
- `pypdf` – Lets your code *read PDFs* so it can understand documents.
- `gradio` – Helps you *make a cute little website or app* to show your AI to others.
- `python-dotenv` – Keeps your *secrets safe*, like passwords or API keys, in a secret `.env` file.
- `pandas` – Helps you *organize and analyze data*, like putting things neatly in a table.

*The version numbers ensure compatible library versions to prevent conflicts, while -q runs the installation quietly without extra messages.

Imports

General helpers

These are your program's **basic tools**:

- `os`, `shutil`, `pickle` – help your program **manage files and folders**.
- `numpy` and `pandas` – help your program **do math and handle tables of data**.
- `dotenv` – keeps **secret keys** safe (like your OpenAI API key).
- `tqdm` – shows a **progress bar** so you can see when your program is working.

Reading and splitting documents

- `Document` and `PyPDFLoader` – let your program **read PDF files** and turn them into text it can understand.
- `RecursiveCharacterTextSplitter` – **chops big chunks of text into smaller pieces** so it's easier for the AI to process.

Memory and retrieval

- `ParentDocumentRetriever`, `InMemoryStore`, and `InMemoryDocstore` – help your program **store and recall information quickly**.
- `faiss` – works like a **super-fast search engine** that helps the program find the most relevant information from memory.

Imports

Understanding and answering

- `FAISS` and `OpenAIEmbeddings` – turn text into **number patterns (embeddings)** so the program can understand meaning.
- `ChatOpenAI` and `RetrievalQA` – the **brain** of your program — these make it **think and answer questions** based on what it has read.

Checking performance

- `cosine_similarity` – helps measure **how similar two answers or ideas are**, to check how well your program understood things.

User Interface

- `gradio` – lets you build a **simple website or app** where people can **chat with your program.**

Colab helper

- `google.colab.files` – lets you **download files** from Colab to your own computer.

Environment Fix

Sometimes, Colab secretly puts up **walls called “proxies”** that make it hard for your program to talk to the internet properly — like when you’re trying to call OpenAI or download something.

```
print("Applying environment fix...")
```

This just **tells you what’s happening** — a message that says,

> “I’m fixing the environment!”

```
os.environ.pop('HTTP_PROXY', None)
```

```
os.environ.pop('HTTPS_PROXY', None)
```

These two lines mean:

> “Remove the invisible walls (proxy settings) if they exist.”

So your robot can **connect to the internet directly** — no more blocked doors!

```
print("Environment fix applied.")
```

This message means “All clear! The playground is open — your robot can now talk to the internet safely.”

Load API Key

Load the secret key

```
load_dotenv('template.env')
OPENAI_API_KEY = os.getenv('OPENAI_API_KEY')
```

- Your program needs a **special key** (API key) to talk to OpenAI (like a password to a secret clubhouse).
- This key is kept safe in a **`.env` file** called `template.env`.
- `load_dotenv()` opens that file and tells Python:

  “Hey, here are my secret keys!”
- `os.getenv('OPENAI_API_KEY')` grabs the OpenAI key from that file.

Load API Key

Check if the key exists

```
if not OPENAI_API_KEY:
    raise ValueError("OpenAI API key not found. Please set it in 'template.env'.")
else:
    print("OpenAI API Key loaded successfully.")
```

- If the key is missing, the program says:

  “I can’t work without my secret key!”

- If the key is there, the program is happy:

  “Yay! I’m ready to play!”

Load API Key

Set folder paths

```
DATA_PATH = "docs/"

VECTOR_STORE_PATH = "vector_store/"
```

- `DATA_PATH` → the **folder where your PDFs or documents live**.
- `VECTOR_STORE_PATH` → the **folder where your program stores its memory** (embeddings) so it can find answers faster later.

Load Documents (This is where you code!)

This part of your project is where your **robot learns to read both PDFs and spreadsheets (CSVs)** — and even *summarize the data smartly all by itself!*

What this section does

You're teaching your program to:

1. **Find** all the PDF and CSV files in your `docs/` folder.
2. **Read** them one by one.
3. **Summarize** what's inside (especially the CSVs).
4. **Turn** everything into mini "documents" that the AI can remember later.

Load Documents

Step 1: Read the CSV

```
df = pd.read_csv(file_path)

filename = os.path.basename(file_path)
```

- Opens the spreadsheet.
- Saves the name (like “sales_data.csv”) so the program remembers where it came from.

Step 2: Detect column types

```
numeric_cols = df.select_dtypes(include=np.number).columns.tolist()

categorical_cols = df.select_dtypes(include=['object', 'category']).columns.tolist()
```

- The program looks at each column and asks:
  - “Is this a **number column**?” (like price, age, or quantity)
  - “Is this a **word column**?” (like name, city, or color)

Step 3: Split the file into smaller pieces

```
for i in range(0, len(df), rows_per_chunk):
```

- The program doesn’t read the whole sheet at once — it **breaks it into chunks** (like 25 rows at a time).

Load Documents

Step 4: Create a smart summary

The robot writes a mini paragraph like:

> “This chunk from ‘sales.csv’ contains 25 records.
> ‘Price’ ranges from 10 to 100.
> Key ‘Category’ values include: Fruits, Vegetables.”

It picks only the **most useful columns** (up to 8) so the summary doesn’t get too long.

Step 5: Turn it into a pretty table

```
markdown_table = chunk_df.to_markdown(index=False)
```

The data chunk is turned into a **nice Markdown table** (clean and readable for AI).

Step 6: Store it as a “Document”

```
doc = Document(page_content=final_content, metadata={"source": file_path, "row_start": i})
```

Each chunk becomes a **mini document** that includes:

- The text summary
- The data table
- Some details about where it came from (metadata)

Blank Cell #1 (This is where you code!)

Step 1: Bring in the helpers

```
import os

from google.colab import userdata
```

- `os` helps your code work with files and your computer's system.
- `userdata` is like a **safe box in Google Colab** where you can keep secrets (like your OpenAI key) safely — so you don't have to type them every time.

Step 2: Try to get the key automatically

```
openai_api_key = userdata.get('OPENAI_API_KEY')

print("Using API key from Colab Secrets Manager.")
```

- The code asks Colab:

  "Do you already have my secret key stored safely?"

- If yes, it says:

  "Yay! Found it in the secret box!"

Blank Cell #1 (This is where you code!)

Step 3: If no key is found, ask the user

```
except Exception as e:

    print(f"Could not retrieve API key from Colab Secrets Manager: {e}")

    openai_api_key = input("Please enter your OpenAI API key: ")

    print("API key entered manually.")
```

- If the program can’t find the key, it politely asks you:
  “Can you please type your OpenAI key for me?”
- Then it stores what you typed in memory for now.

Blank Cell #1 (This is where you code!)

Step 4: Save the key into a `.env` file

- `with open('template.env', 'w') as f:`
- `    f.write(f'OPENAI_API_KEY={openai_api_key}\n')`
- Now your program **writes the key down** in a file called `template.env`.

- This way, it can **find the key easily** next time without asking again.

Step 5: Confirmation message

- `print("OpenAI API key has been written to template.env")`
- Just a cheerful message saying:

“Done! I’ve safely written your key to the file.”

In short:
This code safely finds (or asks for) your OpenAI API key and saves it inside a hidden `.env` file so your program can use it securely in future runs.

Blank Cell #2 (This is where you code!)

What happens here

```
with open('/content/template.env', 'r') as f:
    print(f.read())
```

1. **`with open('/content/template.env', 'r') as f:`**
    - This opens the file called `template.env` in **read mode** (`'r'` means “I just want to look, not change anything”).
    - Think of it like your program **opening a notebook** to read what’s written inside.
2. **`print(f.read())`**
    - This reads **everything inside** the file and prints it out.

So you’ll see something like:

```
OPENAI_API_KEY=sk-abc123xyz...
```

- That’s your **secret key** saved earlier.

Important reminder!!:

Be careful not to **share or show** the output (your API key) to anyone — it’s like a **password** that gives access to your OpenAI account.

2.2 Create Advanced Retriever and Vector Store (This is where you code!)

Starting setup

```
print("Setting up advanced retriever...")
```

Collect all documents

```
all_docs = pdf_documents + csv_derived_documents
```

- This combines everything your program read earlier — all the **PDF pages** and **CSV summaries** — into **one big list** called `all_docs`.
- Think of it as putting *all the notes* into one big notebook.

Split documents into smaller pieces

```
parent_splitter = RecursiveCharacterTextSplitter(chunk_size=1200)

child_splitter = RecursiveCharacterTextSplitter(chunk_size=300)
```

- The program uses these tools to **cut the text into smaller chunks**:
  - Big pieces (parents) — 1200 characters.
  - Tiny pieces (children) — 300 characters.
- Smaller pieces make it **easier to find the right answers later**.

2.2 Create Advanced Retriever and Vector Store (This is where you code!)

Create text embeddings (turn words into numbers)

```
embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY, chunk_size=900)

embedding_dimension = len(embeddings.embed_query("test"))
```

- The program uses **OpenAI's embeddings** to turn each sentence into **a list of numbers** — this helps it *understand meaning*.
- It also checks how long those number lists are (`embedding_dimension`).

2.2 Create Advanced Retriever and Vector Store (This is where you code!)

Build a memory index

```
index = faiss.IndexFlatL2(embedding_dimension)
```

- FAISS is like a **super-speedy search machine**.
- It helps the program **find which piece of text is most similar** to your question.
- Imagine asking a librarian for info — FAISS helps them find the *right page instantly*.

Connect memory systems together

```
faiss_docstore = InMemoryDocstore()

vectorstore = FAISS(

    embedding_function=embeddings,

    index=index,

    docstore=faiss_docstore,

    index_to_docstore_id={} )

store = InMemoryStore()
```

- These are like the program's **notebooks** where it keeps:
    - The embeddings (vectorstore), the actual text (docstore), and all connected nicely so it knows which memory belongs to which document.

2.2 Create Advanced Retriever and Vector Store (This is where you code!)

Create the retriever

```
retriever = ParentDocumentRetriever(
    vectorstore=vectorstore,
    docstore=store,
    child_splitter=child_splitter,
    parent_splitter=parent_splitter,)
```

- This retriever is the program's **smart librarian**.
- It remembers both **big ideas (parent chunks)** and **details (child chunks)** — so when you ask a question, it finds *just the right piece* of information!

Add documents to memory

```
retriever.add_documents(all_docs, ids=None)
```

- The program now **fills its memory** with all the text it's read.
- It's like saying: "Here's everything I've learned so far — remember it well!"

All done

```
print("Advanced retriever created successfully.")
```

- Your program proudly says: "My memory system is ready! I can now understand and recall things intelligently!"

2.3 Save Retriever Components (This is where you code!)

Step 1: Message

```
print("Saving retriever components...")
```

Just a friendly message to say: “I’m now saving my memory!”

Step 2: Save the vector store

```
retriever.vectorstore.save_local(VECTOR_STORE_PATH)
```

- The **vector store** is where the program keeps its **embeddings** (the number versions of your text).
- This line **saves that memory** into a folder called `vector_store/`.
- So next time, the program can just **load it instantly** instead of rebuilding it.

Step 3: Save the parent document store

```
with open(os.path.join(VECTOR_STORE_PATH, "parent_docstore.pkl"), "wb") as f:
    pickle.dump(retriever.docstore, f)
```

- This part saves another piece of memory — the **parent docstore** — which holds the **original text chunks** that the AI links to.
- It uses `pickle`, a tool that **turns Python objects into files**.
- The `"wb"` means “write in binary” — a fancy way of saying “save this safely, exactly as it is.”

Step 4: Confirmation

`print(f"Retriever components saved to '{VECTOR_STORE_PATH}'.")` Your program proudly says: “All done! My memories are now saved in the `vector_store` folder!”

2.3 Load Retriever Components (This is where you code!)

1. `OpenAIEmbeddings` — Recreates the same embedding model used during saving to ensure compatibility with the FAISS index.
2. `FAISS.load_local()` — Loads the vectorstore (FAISS index + metadata).
3. `pickle.load()` — Loads the saved document store that maps document IDs to their content.
4. **Recreates** `ParentDocumentRetriever` — Since text splitters aren't stored, they're reinitialized to maintain consistent document structure.
5. **Final print** — Confirms successful reloading so you can continue retrieval or RAG (Retrieval-Augmented Generation) tasks immediately.

If you run this cell after your save step, it should complete without errors.

Create the RetrievalQA Chain (This is where you code!)

`ChatOpenAI`

- Creates a chatbot powered by the `gpt-3.5-turbo` model.
- `temperature=0.1` makes it give focused, factual answers instead of creative ones.
- The `OPENAI_API_KEY` connects it securely to OpenAI's servers.

`RetrievalQA.from_chain_type()`

- Combines your **retriever** (which fetches the most relevant documents) with your **LLM** (which answers questions).
- `chain_type="stuff"` means the retriever's results are "stuffed" (combined) together and passed into the model as context.
- `return_source_documents=True` ensures it also tells you which document(s) it used to form the answer.

Result:

- The variable `qa_chain` is now your smart Q&A system.

You can now ask questions like:

```
result = qa_chain({"query": "What is the company's policy on data retention?"})
print(result["result"])
```

- and it will respond with the most relevant, fact-based answer using your uploaded documents.

Run Evaluation

1. `evaluation_set`
   It's a list of test questions and their *correct answers* (like an answer key).
   Example: "What are the three core clinical features of dementia with Lewy bodies?"
2. `for item in evaluation_set:`
   The code loops through each question one by one.

3. `qa_chain.invoke({"query": query})`
   Asks your RAG chatbot the question and gets its answer.

4. `embeddings.embed_query()`
   Turns both the chatbot's answer *and* the true answer into number patterns (embeddings) so they can be compared mathematically.

5. `cosine_similarity(...)`
   Checks how similar those two number patterns are —

   - 1.0 = very similar (almost the same meaning)
   - 0 = completely different

6. `results.append({...})`
   Saves the question, true answer, chatbot's answer, and its similarity score.

7. `average_similarity` **&** `accuracy`
   Calculates how well your RAG system performed overall.

   - **Average similarity** → how close all answers were to the truth on average
   - **Accuracy (>0.75)** → how many answers were "good enough"

8. `print()` **at the end**
   Shows a full report — each question, answer, score, and a summary.

Launch Gradio UI

Step-by-step breakdown

1. `def ask_question_for_chat_interface(query, history):`
   This function is your chatbot's **brain** —
   it listens to the user's question (`query`) and remembers past chats (`history`).

2. `qa_chain.invoke({"query": query})`
   The chatbot sends your question into the **RAG system** (retriever + LLM)
   to find and generate the best answer.

   - `sources_text` **part**
     The chatbot checks which documents it used to answer.
     It shows them nicely inside a *collapsible box* — so you can see *where* the answer came from.
     Example:

     ```
     Sources (2)
     - dementia_study.pdf (p. 3)
     - activity_data.csv
     ```

3. `sources_text` **part**

   The chatbot checks which documents it used to answer.

Launch Gradio UI

Step-by-step breakdown

`4. full_response = answer + sources_text`
Combines the answer + sources into one clean message.

`5. demo = gr.ChatInterface(...)`
This builds a **Gradio chat window** — like a mini app where you can:

- Type a question
- See your chatbot's answer
- Click to see document sources
- Try out example questions

`6. demo.launch(debug=True, share=True)`
Starts your chatbot!

- `debug=True` → helps show errors if something goes wrong
- `share=True` → gives you a **shareable link** so others can try it too