Konecta Internship Task 6 (RAG System for PDF Manuals)

Name: Ahmed Ayman Ahmed Alhofy
Track: Artificial Intelligence & Machine Learning
Repository Link: https://github.com/AhmedAyman4/konecta-internship/tree/main/Task-6

In [ ]:
```
!pip install -U langchain-community langchain-google-genai pypdf langchain_huggingface chromadb
```

What is Retrieval-Augmented Generation (RAG)?

RAG (Retrieval-Augmented Generation) is a method that improves large language models by adding external knowledge. Instead of guessing or using built-in knowledge, RAG first **retrieves** relevant information from a document (like a manual or PDF), then uses that info to **generate** accurate answers.

This helps avoid fake or incorrect answers (hallucinations) and is perfect for tasks like answering questions about technical guides, company docs, or any specific dataset.

How RAG Works – Simple Steps

1. **Load Documents**
   Read files (like PDFs) using tools like `PyPDFLoader`.
2. **Split Text**
   Break text into small chunks so the model can process them easily.
3. **Create Embeddings**
   Convert each chunk into numbers (vectors) that represent its meaning.
4. **Store in Vector Database**
   Save these vectors in a search-friendly database like Chroma or FAISS.
5. **Retrieve on Query**
   When you ask a question, find the most relevant text chunks.
6. **Add Context to Prompt**
   Give the LLM both your question and the retrieved text.
7. **Generate Answer**
   The model answers based only on the provided context — no guessing.
8. **(Optional) Support Chat History**
   Handle follow-up questions by remembering past messages.

Why Use RAG?

- ✅ Reduces hallucinations
- ✅ Shows sources for answers
- ✅ Works with your own documents
- ✅ No need to train or fine-tune the model

In this task, RAG is used to answer questions about device manuals — giving accurate, cited responses from real product guides.

Importing Required Libraries

This cell sets up the environment and imports necessary libraries for document loading, text processing, embedding, vector storage, and conversational retrieval using LangChain and Google's Generative AI. It also configures warnings to be suppressed for cleaner output.

In [2]:
```
import os
import time
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from pathlib import Path
from langchain.chains import create_history_aware_retriever, create_retrieval_chain
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
from langchain_core.runnables.history import RunnableWithMessageHistory
from langchain_core.chat_history import InMemoryChatMessageHistory
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain.memory import ConversationBufferMemory
from langchain_google_genai import ChatGoogleGenerativeAI
from google.colab import userdata
from langchain_huggingface import HuggingFaceEmbeddings
from langchain.vectorstores import Chroma, FAISS

import warnings
warnings.filterwarnings("ignore")
```

Load PDF Documents

This cell loads multiple PDF manuals from specified file paths using `PyPDFLoader` and aggregates them into a single document list. Each loaded document retains its source metadata for reference.

In [3]:
```
# Create a list of paths to your PDF manuals
pdf_paths = [
    "/content/Data/legion_5.pdf",
    "/content/Data/loq_series.pdf",
    "/content/Data/zbook_14u_g6.pdf",
    "/content/Data/samsung_odyssey_g9_series.pdf"
]

docs = []
for path in pdf_paths:
    loader = PyPDFLoader(path)
    # The loader automatically adds document source metadata
    docs.extend(loader.load())
```

```
WARNING:pypdf.generic._data_structures:Multiple definitions in dictionary at byte 0x292363 for key /Im30
WARNING:pypdf.generic._data_structures:Multiple definitions in dictionary at byte 0x292c33 for key /Im220
WARNING:pypdf.generic._data_structures:Multiple definitions in dictionary at byte 0x292e4f for key /Im13
WARNING:pypdf.generic._data_structures:Multiple definitions in dictionary at byte 0x292f6c for key /Im275
WARNING:pypdf.generic._data_structures:Multiple definitions in dictionary at byte 0x2d683b for key /Im250
WARNING:pypdf.generic._data_structures:Multiple definitions in dictionary at byte 0x2d684a for key /Im251
WARNING:pypdf.generic._data_structures:Multiple definitions in dictionary at byte 0x2d6868 for key /Im250
WARNING:pypdf.generic._data_structures:Multiple definitions in dictionary at byte 0x2d6877 for key /Im251
WARNING:pypdf.generic._data_structures:Multiple definitions in dictionary at byte 0x2d6976 for key /Im252
WARNING:pypdf.generic._data_structures:Multiple definitions in dictionary at byte 0x2d6985 for key /Im251
WARNING:pypdf.generic._data_structures:Multiple definitions in dictionary at byte 0x2d6994 for key /Im253
WARNING:pypdf.generic._data_structures:Multiple definitions in dictionary at byte 0x2d6ac1 for key /Im250
```

Prepare Model and Split Documents

Initializes the Google Generative AI model (`gemini-1.5-flash`) for fast inference and sets up a text splitter to break documents into manageable chunks. Metadata such as document name and page number are preserved for context tracking.

In [4]:
```
google_llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0,
    max_tokens=None, # Max output tokens
    timeout=None,
    max_retries=2,
    convert_system_message_to_human=True,
    google_api_key=userdata.get('GOOGLE_API_KEY')
)
```

Split Documents into Chunks

Splits loaded PDFs into smaller chunks (1000 characters each, with 100-character overlap) for better retrieval. Metadata like document name and page number is preserved for source tracking.

In [5]:
```
splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=100,
    separators=["\n\n", "\n", ".", " "]
)

chunks = splitter.split_documents(docs)

# Attach extra metadata manually (doc name, page number)
for c in chunks:
    c.metadata["doc_name"] = c.metadata.get("source", "Unknown")
    c.metadata["page_number"] = c.metadata.get("page", "N/A")
```

In [6]:
```
chunks[20]
```

Out[6]:
```
Document(metadata={'producer': 'PDFlib+PDI 9.0.7p3 (C++/Win64)', 'creator': 'PTC Arbortext Publishing Engine', 'creationdate': '2021-06-15T10:52:52+08:00', 'configfile': 'E:\\program files\\ptc\\arbortext pe\\custom\\app\\standard.appcf', 'stylesheet': 'E:\\prog
ram files\\ptc\\arbortext pe\\custom\\doctypes\\ditabase\\8.5x11_book_hel.style', 'ptcarbortextbuild': 'R70M060-65', 'title': 'User Guide', 'epsprocessor': 'PStill version 1.84.42', 'printengine': 'PTC Arbortext Advanced Print Publisher 11.1.4334/W Library-x64',
'moddate': '2021-06-15T10:52:52+08:00', 'source': '/content/Data/legion_5.pdf', 'total_pages': 34, 'page': 14, 'page_label': '15', 'doc_name': '/content/Data/legion_5.pdf', 'page_number': 14}, page_content='5 15ACH6A and Lenovo Legion 5 15ACH6A \n- Input: 100 V
ac-240 V ac, 50 Hz-60 Hz\n- Output: 20 V dc, 11.5 A/20 V dc, 15 A\n- Power: 230 W/300 W\nBattery pack\n• 15-inch models \n- Capacity: 60 Wh/80 Wh\n- Number of cells: 4\n• 17-inch models \n- Capacity: 80 Wh\n- Number of cells: 4\nNote: The battery capacity is the
typical or average capacity as measured in a specific test \nenvironment. Capacities measured in other environments may differ but are no lower than the \nrated capacity (see product label).\nMicroprocessor To view the microprocessor information of your compute
r, type system information in the \nWindows search box and then press Enter.\nMemory\n• Type: Double data rate 4 (DDR4) small outline dual in-line memory module (SODIMM)\n• Number of slots: 2\nChapter 1 . Meet your computer 9')
```

In [7]:
```
len(chunks)
```

Out[7]:
```
411
```

Embedding Model and Vector Store Selection

We use the `all-MiniLM-L6-v2` sentence transformer model from Hugging Face, which efficiently maps text to a 384-dimensional dense vector space, making it ideal for semantic search and retrieval. It offers a strong balance between performance and speed for our use case.

Chroma is chosen as the vector store over **FAISS** for several reasons:

- **Ease of Use**: Chroma is designed to be developer-friendly, with straightforward APIs for adding, querying, and persisting data.
- **Native Persistence**: Chroma supports built-in persistence with minimal configuration (via `persist_directory`), whereas FAISS requires manual serialization (e.g., saving/loading indexes and embedding matrices separately).
- **LangChain Integration**: Chroma has seamless, first-party integration with LangChain, simplifying retrieval chain setup.
- **Metadata Support**: Chroma natively supports metadata filtering, which is useful for source tracking (e.g., filtering by document or page).

While **FAISS** (by Facebook) is highly optimized for performance and scalability in large-scale similarity search, it is more complex to manage and lacks built-in persistence and metadata handling out of the box. For this project, where simplicity, rapid prototyping, and local persistence are prioritized over massive scale, **Chroma** is the more suitable choice.

In [8]:
```
model_name = "sentence-transformers/all-MiniLM-L6-v2"
model_kwargs = {'device': 'cpu'}
encode_kwargs = {'normalize_embeddings': False}
hf = HuggingFaceEmbeddings(
    model_name=model_name,
    model_kwargs=model_kwargs,
    encode_kwargs=encode_kwargs
)
```

In [9]:
```
# Create a Chroma vector store
# Specify a persist_directory to save the collection locally
vectorstore = Chroma.from_documents(chunks, hf, persist_directory="./chroma_manuals_db")

# To load it later:
# vectorstore = Chroma(persist_directory="./chroma_manuals_db", embedding_function=hf)

# Display the vector store
print(vectorstore)
```

```
<langchain_community.vectorstores.chroma.Chroma object at 0x785c7aa9ef30>
```

Setting Up the Retrieval and QA Chains

This cell configures the retrieval and question-answering pipelines:

- A **retriever** fetches the top 3 relevant document chunks from the vector store.
- A **contextualization chain** converts follow-up questions into standalone queries using chat history.
- A **question-answering chain** generates concise, context-bound answers using the retrieved documents, ensuring responses are grounded in the provided manuals.

In [10]:
```
# Create a retriever from the vector store
retriever = vectorstore.as_retriever(search_kwargs={"k": 3})

# System prompt to turn a follow-up question into a standalone question
contextualize_q_system_prompt = """Given a chat history and the latest user question \
which might refer to chat history, formulate a standalone question which can be used \
to retrieve relevant documents. Do not answer the question, just reformulate it if needed \
and otherwise return it as is."""

# Prompt template for contextualizing questions
contextualize_q_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", contextualize_q_system_prompt),
        MessagesPlaceholder("chat_history"),
        ("human", "{input}"),
    ]
)

# Chain that makes the question standalone
contextualize_q_chain = contextualize_q_prompt | google_llm | StrOutputParser()

# System prompt for answering based on retrieved context
qa_system_prompt = """You are an assistant for question-answering tasks.
You must strictly use only the retrieved context below to answer.
If the answer is not contained in the provided context, say: "I don't know based on the manuals."
Do not use outside knowledge.
Use three sentences maximum and keep the answer concise.

{context}"""

# Prompt template for answering questions
qa_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", qa_system_prompt),
        MessagesPlaceholder("chat_history"),
        ("human", "{input}"),
    ]
)

# Chain that answers the question based on the context
question_answer_chain = create_stuff_documents_chain(google_llm, qa_prompt)
```

Building the Conversational RAG Pipeline

This cell assembles a **Retrieval-Augmented Generation (RAG)** chain with source tracking and chat history support:

- Combines question contextualization, retrieval, and answer generation.
- Enhances responses with source metadata (document name, page number).
- Uses `RunnableWithMessageHistory` to maintain conversational memory across turns, enabling context-aware interactions.

In [11]:
```
# Combine contextualization and question-answering chains
def with_sources(inputs):
    """Ensure output contains both the answer and the sources."""
    answer = inputs.get("answer", "")
    docs = inputs.get("context", [])

    return {
        "answer": answer,
        "sources": [
            {
                "doc_name": d.metadata.get("doc_name", "Unknown"),
                "page_number": d.metadata.get("page_number", "N/A"),
            }
            for d in docs if hasattr(d, "metadata")
        ],
    }

rag_chain = (
    RunnablePassthrough.assign(context=contextualize_q_chain | retriever)
    | {
        "answer": question_answer_chain,
        "context": lambda x: x["context"],  # keep the retrieved docs
    }
    | with_sources
)

# In-memory storage for chat history
store = {}
def get_session_history(session_id: str) -> InMemoryChatMessageHistory:
    if session_id not in store:
        store[session_id] = InMemoryChatMessageHistory()
    return store[session_id]

# Conversational RAG chain with chat history
conversational_rag_chain = RunnableWithMessageHistory(
    rag_chain,
    get_session_history,
    input_messages_key="input",
    history_messages_key="chat_history",
    output_messages_key="answer",
)
```

Running Example Q&A Interactions

This cell demonstrates the conversational RAG system by asking a series of questions, including follow-ups and cross-document queries. The chatbot responds using only the retrieved context, cites sources, and maintains conversation history. A 5-second delay between calls ensures compliance with API rate limits.

In [13]:
```
# Example Q&A interactions
questions = [
    "What is the maximum supported RAM on the Legion 5?", # Direct factual
    "How many memory slots does it have?", # Follow-up
    "What is the maximum refresh rate of the Samsung Odyssey G9?", # Different manual
    "How to upgrade the RAM on the ZBook 14u G6?", # Another manual
    "What is the capital of France?" # Out-of-scope
]

session_id = "example_session" # Use a consistent session ID for follow-up questions



print("Starting Q&A interactions...\n")
for question in questions:
    print(f"User: {question}")
    result = conversational_rag_chain.invoke(
        {"input": question},
        config={"configurable": {"session_id": session_id}}
    )
    print(f"Chatbot: {result['answer']}")
    if result["sources"]:
        for src in result["sources"]:
            print(f"  ↳ Source: {src['doc_name']} (page {src['page_number']})")
    print()

    time.sleep(5)  # wait 5 sec between calls to stay under 10 RPM

print("Q&A interactions finished.")
```

```
Starting Q&A interactions...

User: What is the maximum supported RAM on the Legion 5?
Chatbot: I don't know based on the manuals. The provided context specifies the memory type and the number of slots, but it does not state the maximum supported RAM capacity.
  ↳ Source: /content/Data/legion_5.pdf (page 14)
  ↳ Source: /content/Data/legion_5.pdf (page 14)
  ↳ Source: /content/Data/legion_5.pdf (page 14)

User: How many memory slots does it have?
Chatbot: I don't know based on the manuals. The provided context does not contain information about the number of memory slots.
  ↳ Source: /content/Data/legion_5.pdf (page 0)
  ↳ Source: /content/Data/legion_5.pdf (page 0)
  ↳ Source: /content/Data/legion_5.pdf (page 0)

User: What is the maximum refresh rate of the Samsung Odyssey G9?
Chatbot: For the C49G9*T* model, which is the Samsung Odyssey G9, the maximum refresh rate is 240 Hz. This maximum refresh rate is supported with DisplayPort1 and DisplayPort2. The function is disabled when the HDMI port is in use.
  ↳ Source: /content/Data/samsung_odyssey_g9_series.pdf (page 18)
  ↳ Source: /content/Data/samsung_odyssey_g9_series.pdf (page 18)
  ↳ Source: /content/Data/samsung_odyssey_g9_series.pdf (page 19)

User: How to upgrade the RAM on the ZBook 14u G6?
Chatbot: I don't know based on the manuals. The provided context does not contain information on how to upgrade the RAM on the ZBook 14u G6.
  ↳ Source: /content/Data/zbook_14u_g6.pdf (page 51)
  ↳ Source: /content/Data/zbook_14u_g6.pdf (page 51)
  ↳ Source: /content/Data/zbook_14u_g6.pdf (page 51)

User: What is the capital of France?
Chatbot: I don't know based on the manuals.
  ↳ Source: /content/Data/samsung_odyssey_g9_series.pdf (page 43)
  ↳ Source: /content/Data/samsung_odyssey_g9_series.pdf (page 43)
  ↳ Source: /content/Data/samsung_odyssey_g9_series.pdf (page 14)

Q&A interactions finished.
```

Q&A Interaction Results

The RAG system responded to each query using only the information retrieved from the manuals:

- Correctly identified when answers were **not available** in the documents.
- Provided accurate details for supported questions (e.g., Samsung Odyssey G9 refresh rate).
- Cited sources with document name and page number for transparency.
- Handled out-of-scope questions (like general knowledge) safely by not hallucinating.

This demonstrates the system's ability to **retrieve, reason, and respond** with context-awareness and source tracking.