Module 4 Lab Session.ipynb

In [6]:

```
#!pip install openai langchain faiss-cpu sentence-transformers
#!pip install -U langchain-community
```

In [ ]:

```
# Step 1: Import Required Libraries for LLM + Document Retrieval Workflow
import os
import torch
from transformers import pipeline
from langchain.document_loaders import TextLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS
from langchain.llms import HuggingFacePipeline
from langchain.chains import RetrievalQA
```

LangChain is a framework for building applications with language models (LLMs).

- It simplifies the process of integrating LLMs with external data sources (like documents, databases, or APIs)
- It allows you to create intelligent apps such as chatbots, retrieval-augmented generation (RAG) systems, and more.

LangChain acts as the "glue" that connects:

- Your documents (TextLoader)---------------------------------------> Loads plain text documents into LangChain.
- Your text processing (TextSplitter)-------------------------------> Splits long text into manageable chunks.
- Your embeddings (HuggingFaceEmbeddings)---------------------------> Generates embeddings for document chunks.
- Your retrieval system (FAISS)------------------------------------> Stores and retrieves embeddings efficiently using similarity search.It helps you find the most similar text chunks from your documents when someone asks a question.
- Your LLM (HuggingFacePipeline)------------------------------------> Wraps a Hugging Face model as an LLM.
- And your question-answering logic (RetrievalQA)-------------------> Connects retrieval and LLM to answer questions based on documents.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [11]:

```
# Step 2: Load the document. A text loader is used to load the document as raw text into memor

loader = TextLoader("llm_notes.txt", encoding="utf-8")
documents = loader.load()

# Step 3: Split the document. Each chunk has 300 characters, and 50 characters overlap with th

text_splitter = RecursiveCharacterTextSplitter(chunk_size=300, chunk_overlap=50)
docs = text_splitter.split_documents(documents[:5])  # limit size to reduce memory

# Step 4: Embed using Hugging Face sentence transformer

embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
db = FAISS.from_documents(docs, embeddings)

# Step 5: Load flan-t5-small on CPU (safest config). Hugging Face’s pipeline is wrapped into a

device = torch.device("cpu")

qa_pipeline = pipeline(
    "text2text-generation",
    model="google/flan-t5-small",
    max_length=256,
    device=device,
    do_sample=False
)

llm = HuggingFacePipeline(pipeline=qa_pipeline)

# Step 6: Build the Retrieval QA chain. First retrieves top 3 relevant text chunks from FAISS,
#It enhances the LLM's ability to answer questions by grounding it in specific documents.

retriever = db.as_retriever(search_kwargs={"k": 3})
qa_chain = RetrievalQA.from_chain_type(
    llm=llm,
    retriever=retriever,
    return_source_documents=True
)

# Step 7: Ask a question

query = "What is tokenization in LLMs and why is it important?"
result = qa_chain(query)

# Step 8: Show results

print("\n Answer:")
print(result["result"])

print("\n Source Documents:")
for i, doc in enumerate(result["source_documents"], 1):
    print(f"\n--- Source {i} ---")
    print(doc.page_content)
```

```
Device set to use cpu
```

```
 Answer:
Tokenization is the process of splitting text into smaller units called tokens.

 Source Documents:

--- Source 1 ---
Tokenization is the process of splitting text into smaller units called tokens.
Large language models typically use subword tokenization for efficiency.
Example: 'unbelievable' → ['un', 'believ', 'able'].
```

In [ ]: