

Project: Implementing a Retrieval-Augmented Generation (RAG) System using Milvus and Hugging Face LLMs

The objective of this project is to gain practical experience in building a Retrieval-Augmented Generation (RAG) pipeline. You will integrate state-of-the-art Large Language Models (LLMs) from Hugging Face with Milvus, an open-source vector database, to create an intelligent question-answering system capable of retrieving relevant information and generating accurate, contextual answers.

Overview

A Retrieval-Augmented Generation (RAG) combines the power of retrieval systems with generative language models to enhance the accuracy and relevance of generated responses. The workflow is illustrated in Figure 1. By first retrieving contextually relevant documents from an external knowledge base, RAG allows LLMs to generate informed and precise answers grounded in factual data.

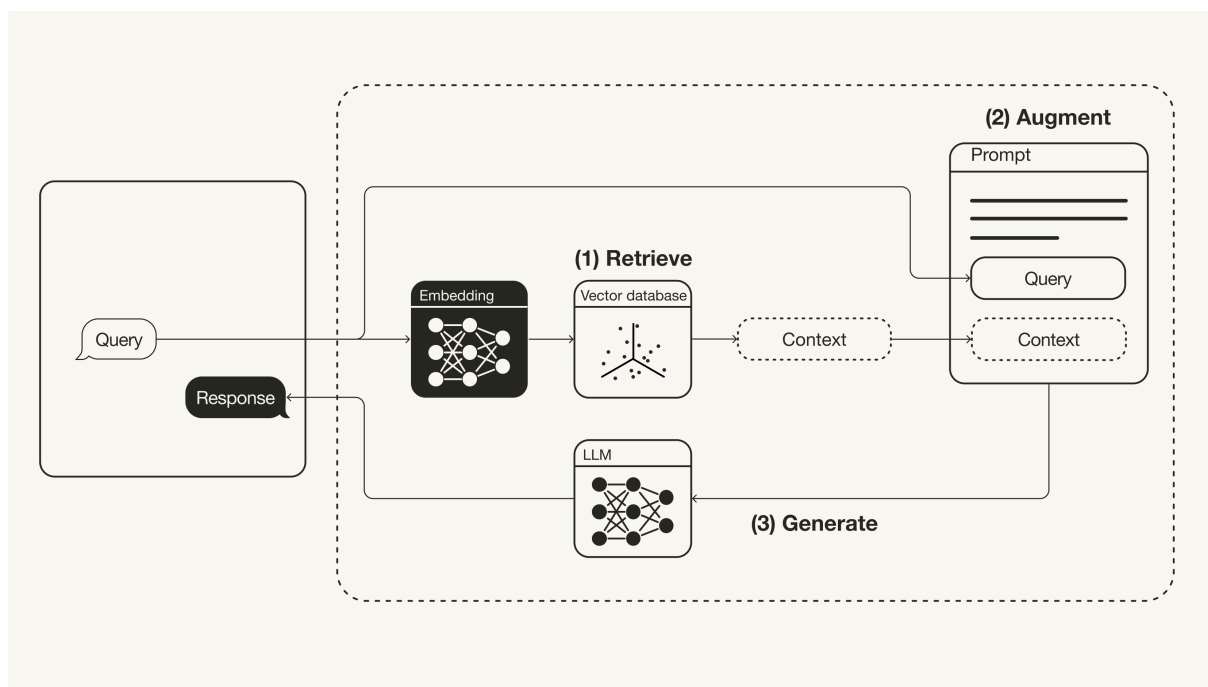


Figure 1 - RAG workflow. (source: <https://towardsdatascience.com/retrieval-augmented-generation-rag-from-theory-to-langchain-implementation-4e9bd5f6a4f2/>)

This project will guide you through the practical implementation of a RAG pipeline by combining:

- **Milvus** (<https://milvus.io/>), a scalable, high-performance vector database for efficient semantic retrieval.
- **Hugging Face** (<https://huggingface.co/>), a leading platform providing access to advanced LLMs for natural language generation tasks.

Technical components

The project involves the following technical steps:

1. Data Preparation

- Select or create a **dataset of your choice**, relevant to the project's domain (e.g., academic articles, FAQs, documentation).
- Preprocess and structure the data for embedding generation.

2. Embedding Generation

- Use embedding models from Hugging Face (e.g., Sentence Transformers) to convert textual data into semantic vectors.

2. Vector Database Setup with Milvus

- Deploy Milvus locally.
- Store embeddings generated from the dataset into Milvus for efficient retrieval.

3. Integration with Hugging Face LLM

- Choose an appropriate LLM from Hugging Face's library.
- Implement a prompt engineering strategy to effectively combine retrieved contexts with user queries.

4. System Implementation

- Build the end-to-end pipeline: query → retrieval from Milvus → prompt construction → answer generation by Hugging Face LLM.

5. Evaluation and Optimization

- Assess retrieval accuracy, response quality, and latency.
- Optimize parameters and components based on evaluation results.
- Experiment with several embedding models and LLMs.
- Experiment with various prompt templates.

Expected Outcomes

Upon completion, you should be able to demonstrate:

- A fully functional RAG system capable of answering user queries accurately by leveraging external knowledge bases.
- Proficiency in using Milvus for semantic search tasks.
- Practical experience deploying Hugging Face LLMs in real-world scenarios.
- Critical analysis skills regarding system performance metrics such as accuracy, relevance, response time, and scalability.

The deadline for this project is **Tuesday 8, April, 2025**. Each student will present his/her project during a slot of 15 minutes (slides presenting the project and a demo). The code and slides will be made available on a gitlab repository, or sent by mail at emmanuel.dellandrea@ec-lyon.fr.

Examples of codes

Here are some examples of codes that may help you in building your RAG pipeline. The use of LLM from Hugging Face has been studied in practical session 7.

0. First, install some useful Python modules:

```
!pip install transformers pymilvus sentence-transformers huggingface-hub langchain_community  
langchain-text-splitters pypdf
```

1. Extracting text from pdf

```
from langchain_community.document_loaders import PyPDFLoader  
from langchain_text_splitters import RecursiveCharacterTextSplitter  
  
loader = PyPDFLoader("any_document.pdf")  
docs = loader.load()  
  
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)  
chunks = text_splitter.split_documents(docs)  
  
text_lines = [chunk.page_content for chunk in chunks]
```

2. Embedding a sentence

```
from sentence_transformers import SentenceTransformer  
  
embedding_model = SentenceTransformer("all-MiniLM-L6-v2") # This is one example  
  
s = text_lines[0]  
e = embedding_model.encode([s])
```

3. Creating a Milvus data collection

```
from pymilvus import MilvusClient  
  
milvus_client = MilvusClient(uri="./my_milvus_db.db")  
  
collection_name = "rag_collection"  
  
milvus_client.create_collection(  
    collection_name=collection_name,  
    dimension=embedding_dim, # The size of the embedding  
    metric_type="IP", # Inner product distance  
    consistency_level="Strong", # Strong consistency level  
)  
  
data = []  
  
# In the following example, emb_text is a function that needs to be written, based on  
# an embedding model  
for i, line in enumerate(text_lines):  
    data.append({"id": i, "vector": emb_text(line), "text": line})  
  
insert_res = milvus_client.insert(collection_name=collection_name, data=data)
```

4. Retrieving data for a query

```
question = "What is the best practice mentionned in the document?"

search_res = milvus_client.search(
    collection_name=collection_name,
    data=[
        emb_text(question)
    ],
    limit=3, # Return top 3 results
    search_params={"metric_type": "IP", "params": {}}, # Inner product distance
    output_fields=["text"], # Return the text field
)
```

5. Creating a prompt

```
PROMPT = """
Use the information enclosed in <context> tags to provide an answer to the
question enclosed in <question> tags.
<context>
{context}
</context>
<question>
{question}
</question>
"""

prompt = PROMPT.format(context=context, question=question)
```