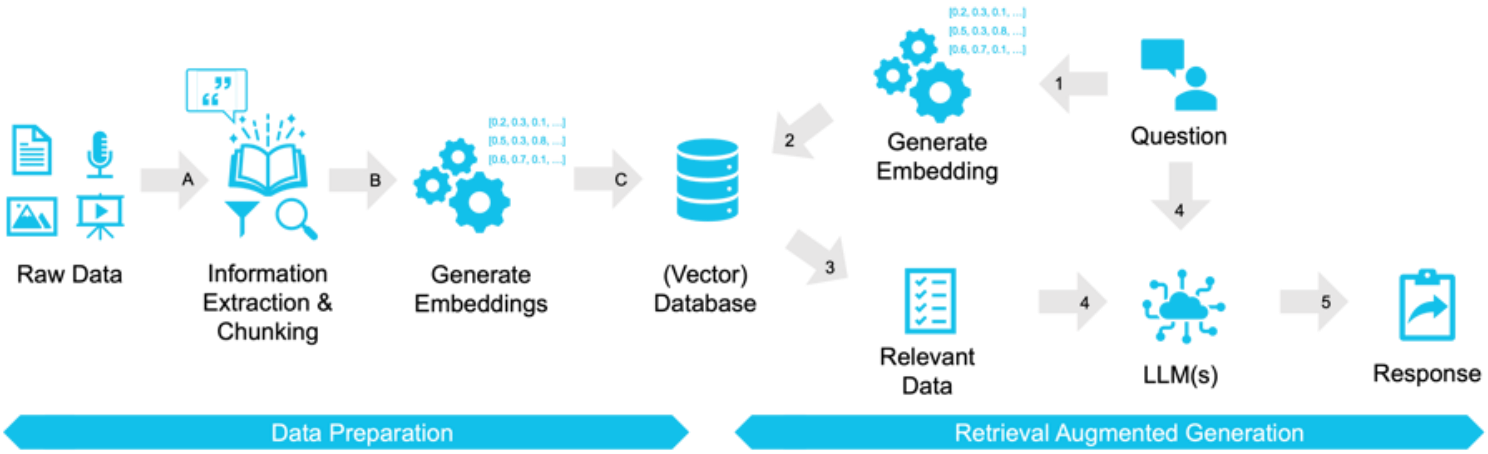
Prepared By:

M.Shahjhan Gondal

RAG PIPELINES UNDERSTANDINGS



-Errors and omissions excepted -

RAG = Retrieval-Augmented Generation

Step 1 – Store Docs all-MiniLM-L6-v2 - Chunk + embed docs using MiniLM

Step 2 – Retrieve FAISS -finds top-k relevant chunks

Step 3 – Generate LLaMA3-70B -generates final answer using retrieved context

General View Steps of RAG

Your Documents -**1**

Convert Text to Embeddings (Using all-MiniLM-L6-v2) -**2**

Store Embeddings in FAISS -**3**

User Query + Embedding -**4**

Search in FAISS -**5**

1: Your Documents

```
docs = [
    "Python is a programming language.",
    "The Eiffel Tower is in Paris.",
    "Machine learning helps computers learn from data.",
    "FastAPI is used to build APIs in Python."
]
```

2: Convert Text to Embeddings (Using all-MiniLM-L6-v2)

```
from sentence_transformers import SentenceTransformer
import faiss
import numpy as np

# Load the model
model = SentenceTransformer('all-MiniLM-L6-v2')

# Create document embeddings
doc_embeddings = model.encode(docs)
```

`doc_embeddings.shape = (4, 384)` → 4 sentences, each is a 384-dimensional vector

Example (truncated for simplicity):

```
[
  [0.12, 0.88, 0.50, ..., 384 dims],  # doc 0
  [0.91, 0.33, 0.75, ..., 384 dims],  # doc 1
  [0.42, 0.34, 0.59, ..., 384 dims],  # doc 2
  [0.71, 0.66, 0.15, ..., 384 dims],  # doc 3

  ...
]
```

3: Store Embeddings in FAISS

```
dimension = doc_embeddings.shape[1]          # 384
index = faiss.IndexFlatL2(dimension)          # Use Euclidean distance
index.add(np.array(doc_embeddings))           # Add all embeddings to index
```

Your FAISS index now looks like:

Index	Document Text	Embedding (384D)
0	Python is a programming language.	[0.12, 0.88, 0.50, ...]
1	The Eiffel Tower is in Paris.	[0.91, 0.33, 0.75, ...]
2	Machine learning helps computers learn...	[0.21, 0.67, 0.35, ...]
3	FastAPI is used to build APIs in Python.	[0.15, 0.92, 0.45, ...]

4: User Query + Embedding

```
query = "What is FastAPI used for?"
query_embedding = model.encode([query])
```

5: Search in FAISS

```
D, I = index.search(np.array(query_embedding), k=1)  # Top-1 match
print("Best match:", docs[I[0][0]])
```

Output:

Best match: FastAPI is used to build APIs in Python.

Try another:

```
query = "What is Python used for?"
query_vec = model.encode([query])
D, I = index.search(np.array(query_vec), k=1)
print("Most similar sentence:", docs[I[0][0]])
```

What is all-MiniLM-L6-v2?

---

all-MiniLM-L6-v2 is a **sentence transformer** model developed by **SentenceTransformers (SBERT**). It's widely used to convert sentences or documents into **dense vector embeddings for** tasks like:

- Semantic search
- Text similarity
- Clustering
- Retrieval-Augmented Generation (RAG)

all-MiniLM-L6-v2 = "All" + "MiniLM" + "6 layers" + "version 2"

- `all:` Trained on a diverse set of tasks, optimized for general-purpose sentence embeddings.
- `MiniLM:` A compact and efficient transformer architecture.
- `L6:` The model has 6 Transformer layers (very lightweight).
- `v2:` Second version (improved training and performance).

Why all-MiniLM-L6-v2 Creates 384-Dimensional Embeddings?

The **output vector size = 384** because:

The hidden size of the transformer model is 384 neurons.

✅ Other Versions & Alternatives:

Model Name	Dimensions	Size	Use Case
all-MiniLM-L6-v2	384	~80 MB	Fast, small, general-purpose embeddings
all-MiniLM-L12-v2	384	~120 MB	Better accuracy, 12 layers
all-distilroberta-v1	768	~250 MB	Better semantic performance
all-mpnet-base-v2	768	~420 MB	□ Highest accuracy (but slower, larger)
paraphrase-MiniLM-L6-v2	384	~80 MB	Optimized for paraphrase detection
multi-qa-MiniLM-L6-cos-v1	384	~80 MB	Trained for question-answer semantic search

✅ Specs:

Feature	Value
Architecture	MiniLM
Transformer Layers	6
Hidden Size	384
Pooling	Mean pooling
Output Size	384-dimensional vector
Speed	Very fast, CPU-friendly
Accuracy	Good for semantic tasks
Size	~80MB

How FAISS Works ?

(Manual Example with 10D Padding)

✓ FAISS vectors are padded to 10D:

```
faiss_vectors = [
  [0.12, 0.88, 0.50, 0, 0, 0, 0, 0, 0, 0],  # doc 0
  [0.91, 0.33, 0.75, 0, 0, 0, 0, 0, 0, 0],  # doc 1
  [0.74, 0.48, 0.10, 0, 0, 0, 0, 0, 0, 0],  # doc 2
  [0.56, 0.10, 0.67, 0, 0, 0, 0, 0, 0, 0],  # doc 3
]
```

✓ Your query is:

```
query_vec = [0.10, 0.90, 0.45]
query_vec_padded = [0.10, 0.90, 0.45, 0, 0, 0, 0, 0, 0, 0]
```

✓ L2 Distance Calculation (Euclidean)

L2=$\sqrt{\{(A_1 - B_1)^2 + (A_2 - B_2)^2 + \cdots + (A_n - B_n)^2\}}$

Code:
L2 = **sqrt**((A1 - B1) ^2 + (A2 - B2) ^2 + ... + (A10 - B10) ^2)

✓ Let's Manually Compare Query Embedding (`query_vec_padded = [0.10,0.90,0.45,0,0,0,0,0,0,0]`) to Each Document

Distance to doc 0: [0.12, 0.88, 0.50]

```
= sqrt((0.10 - 0.12)^2 + (0.90 - 0.88)^2 + (0.45 - 0.50)^2)
= sqrt(0.0004 + 0.0004 + 0.0025)
= sqrt(0.0033)
≈ 0.057
```

Distance to doc 1: [0.91, 0.33, 0.75]

```
= sqrt((0.10 - 0.91)^2 + (0.90 - 0.33)^2 + (0.45 - 0.75)^2)
= sqrt(0.6561 + 0.3249 + 0.09)
= sqrt(1.071)
≈ 1.034
```

Distance to doc 2: [0.74, 0.48, 0.10]

```
= sqrt((0.10 - 0.74)^2 + (0.90 - 0.48)^2 + (0.45 - 0.10)^2)
= sqrt(0.4096 + 0.1764 + 0.1225)
= sqrt(0.7085)
≈ 0.841
```

Distance to doc 3: [0.56, 0.10, 0.67]

```
= sqrt((0.10 - 0.56)^2 + (0.90 - 0.10)^2 + (0.45 - 0.67)^2)
= sqrt(0.2116 + 0.64 + 0.0484)
= sqrt(0.9)
≈ 0.949
```

Document	Distance
doc 0	√0.0033 ≈ **0.057** ✓ best match
doc 1	√0.7085 ≈ 0.841
doc 2	√1.071 ≈ 1.034
doc 3	√0.9 ≈ 0.949

✔ **Closest match is doc 0.**

✓ Final Result:

```
print("Best match:", docs[0])
# Output: Python is a programming language.
```

What is LLaMA3-70B?

- **LLaMA 3** is a family of Large Language Models (LLMs) released by **Meta AI (Facebook)** in 2024.
- The **"70B"** means it has **70 billion parameters**.
- It's one of the most powerful open-source language models currently available, known for high performance in reasoning, summarization, and instruction-following.

 LLaMA3-70B is comparable to GPT-4 in capabilities but **fully open source** (if you have compute to host it, or access via APIs).

✅ Where does LLaMA3-70B fit in RAG?

In your setup:

1-**Document Store**
Your docs are indexed using **FAISS** with embeddings from `all-MiniLM-L6-v2`.

2-**Query Phase (Retrieval)**

- User asks a question.
- You embed the question using `all-MiniLM-L6-v2`.
- FAISS searches the top-k most relevant document chunks.

3-**Generation Phase (Augmentation + LLM)**

- You **pass the top-k retrieved texts (contexts)** along with the **user's question** to **LLaMA3-70B**.
- **LLaMA3-70B** generates a final answer, informed by the retrieved context.

Usually, the top 3–5 relevant chunks are:

- **concatenated** (with separators like `\n\n` or `---`)
- then passed into LLaMA3-70B **with the question**.

Example:

RAG Query Flow Example

User:

What is the capital of France?

→ Embed query with MiniLM → retrieve top docs from FAISS
→ Send prompt to LLaMA3-70B like:
"You are a helpful assistant. Based on the following context, answer the question.

Context:

... [relevant document snippets]

Question:

What is the capital of France?"

→ LLaMA3-70B generates: "The capital of France is Paris."

Prompt Sent to LLaMA3-70B:

You are a helpful assistant. Based on the following context, answer the question:

Context:

[Top 3 FAISS chunks]

Question:

What is the capital of France?

LLaMA3 Output:

The capital of France is Paris.

Matching Multiple Chunks

User Question → Embed → Search FAISS (k=3) → Get Chunks

↓

```
Prompt = """
You are a helpful assistant. Based on the context, answer the question.

Context:
[chunk 1]
---
[chunk 2]
---
[chunk 3]

Question:
[User Question]
"""
```

↓

LLaMA3-70B → Answer

Summary: FAISS + LLM in Action

Component	Role
FAISS	Finds relevant document chunks (fast)
SentenceTransformer	Converts text to embeddings
LLaMA3-70B	Generates final, context-aware answer

Summary

Model	Max Tokens	Safe # of FAISS Chunks	Notes
LLaMA3-7B	8192	3–5 (avg size)	Yes, it works
LLaMA3-70B	32,768	10–20+	Better for long documents