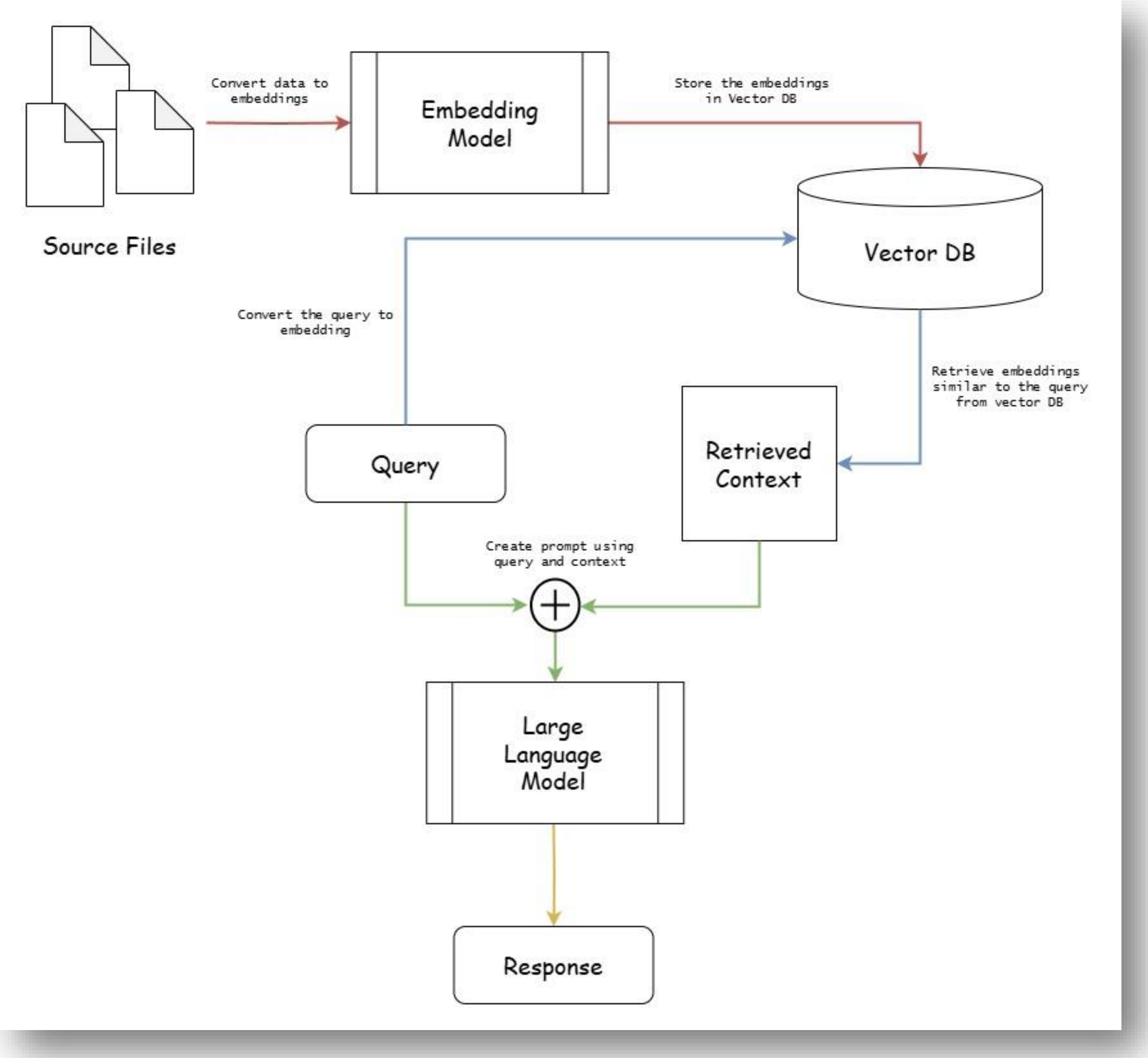
RAG System Architecture:


Source Files
Convert data to embeddings
Embedding Model
Store the embeddings in Vector DB
Vector DB
Convert the query to embedding
Retrieve embeddings similar to the query from vector DB
Retrieved Context
Query
Create prompt using query and context
Large Language Model
Response

This system processes diverse source files, including text, audio transcripts, video transcripts, CSV files, PDF documents, and images, to generate responses using a retrieval-augmented generation (RAG) pipeline. Here's an overview of its workflow:

1. **Embedding Creation**:
   - Source files are processed using the embedding model (This system uses all-MiniLM-L6-v2) to transform the text into vector representations.
   - The generated embeddings are stored in a vector database (This system uses FAISS) for efficient retrieval.

2. **Query Processing**:

   - User queries are also converted into vector embeddings using the same model (all-MiniLM-L6-v2).

   - These embeddings are passed to the vector database(FAISS) to perform a similarity search and retrieve the most relevant context.

3. **Response Generation**:

   - A prompt is created by combining the user query and the retrieved context.

   - The prompt is processed by the large language model (This system uses gemini-1.5-flash) to generate a coherent and accurate response.

Implementation Steps:

1. Data Preprocessing:

- **PDF Documents –**
  - **Library used:** PyPDFLoader from langchain community
  - The above library is used to extract text from the PDF document and split it into chunks for loading it into the vector database. (ref)

- **Text Files –**
  - **Library used:** TextLoader from langchain community
  - The above library is used to extract text from the text file(e.g., transcripts, notes) and split it into chunks for loading it into the vector database. (ref)

- **Microsoft Word Documents –**
  - **Library used:** Docx2txtLoader from langchain community
  - The above library is used to extract text from the docx file and used for loading it into the vector database. (ref)

- **CSV Files -**
  - **Library used:** CSVLoader from langchain community
  - The above library is used to extract the samples' text from csv file.(ref)

- **Image Files –**
  - **Library used:** PIL.Image and pytesseract
  - PIL.Image is used to load the image file. (ref)
  - Optical character recognition(pytesseract) is used to extract the text from image files. (ref)

- **Audio Files –**
  - **Library used:** datasets.Audio, datasets.Dataset, transformers.pipeline and nltk.sent_tokenize
  - datasets.Audio and datasets.Dataset are used to convert the audio to numerical array. (ref)
  - transformers.pipeline is used to call the Automatic Speech Recognition pipeline for extracting the text from audio. (ref)
  - nltk.sent_tokenize is used for tokenizing the extracted text. (ref)

2. Embedding Generation:

- This system uses embedding model all-MiniLM-L6-v2 (ref) from Hugging Face.
- Why this model?
  - This model maps sentences and paragraphs to dense vector space, which can be used for tasks like clustering and semantic search.
  - Model is fine-tuned on 1 billion sentence pairs dataset, this robust fine-tuning enhances its performance and reliability for large-scale search and retrieval tasks.

3. Vector Indexing:

- Why we need a vector database?
  - A vector database indexes and stores vector embeddings for fast retrieval and similarity search. (ref)
- This system uses Facebook AI Similarity Search (FAISS) (ref) for vector indexing.
- Why FAISS?
  - FAISS not only allows us to build an index and search but it also speeds up search times to enhance performance levels.
  - FAISS often outperforms other vector databases regarding scalability and search accuracy.
- In this system we are using **flat indexing(**we do not modify the vectors that we feed into the vector database**)** for exhaustive search. These indexes

produce the most accurate results, have perfect search quality, but this comes at the cost of significant search times. (ref)

- Our system uses **flat indexing** because it is designed for a smaller dataset and requires exhaustive search through the vector database. While other indexing techniques like **Locality Sensitive Hashing (LSH)** (**ref**) and **Hierarchical Navigable Small Worlds (HNSW)** (ref). are available, flat indexing is a more appropriate choice for our use case.

4. RAG Workflow:

- Source files (e.g., text, PDFs, transcripts, images) are processed using an embedding model to convert their content into vector representations. The generated embeddings are stored in a vector database (e.g., FAISS) to enable efficient similarity-based retrieval.
- User queries are converted into embeddings using the same embedding model used for the source data.
- The vector database performs a similarity search to retrieve the most relevant embeddings (context) based on the user query.
- A prompt is created by combining the user query and the retrieved context, ensuring the model has the necessary background information.
- The large language model gemini-1.5-flash(ref) processes the prompt to generate an informative and contextually accurate response.

5. Evaluation:

- Created evaluation dataset with user queries, retrieval keywords and actual response for evaluating the retrieval from vector database and response generation from large language model.

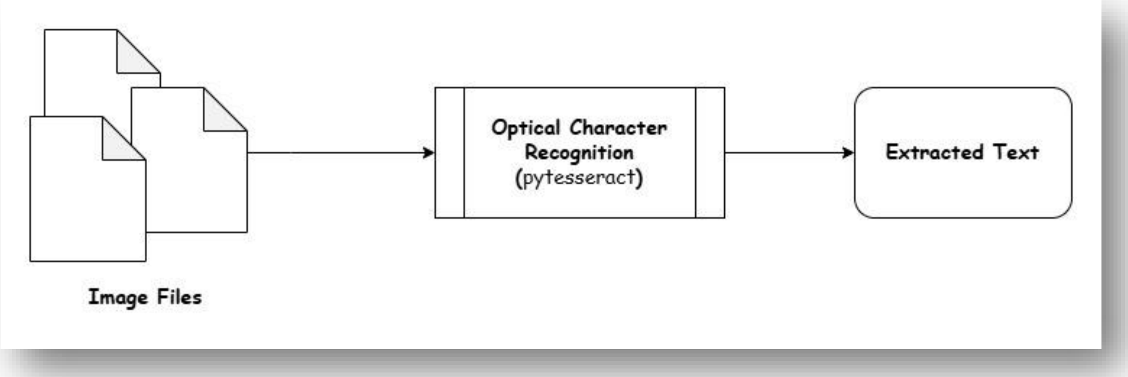
- **Retrieval Evaluation:**
  - Used Keyword matching as evaluation metric.

- **Generation Evaluation:**
  - Used cosine similarity between actual and generated response as evaluation metric.

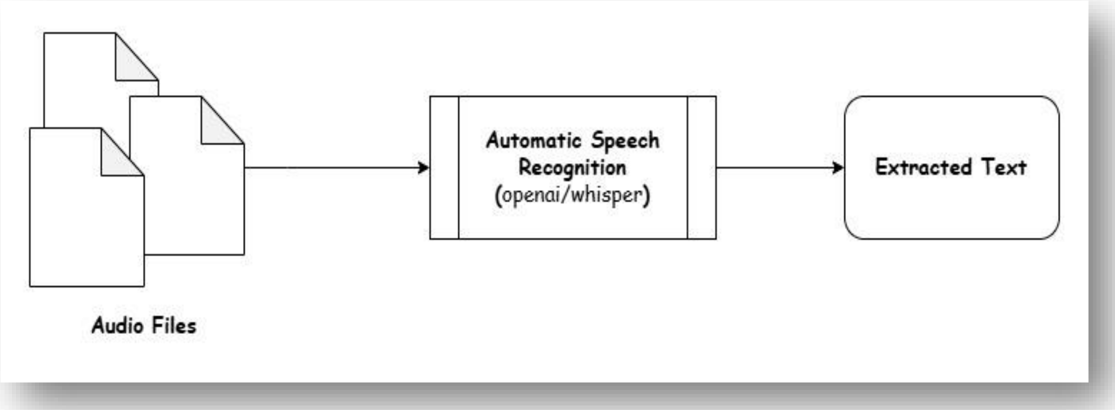
Challenges:

- **Handling Multimedia:**
  - This system handles image and audio data files with assumption that the image contains text on it and audio file is of speech.

- Image files are handled using optical character Recognition (OCR).



- Audio files are handled using Automatic Speech Recognition (ASR).



- For image files containing objects and scenes, popular object recognition models (e.g., YOLO, VGG16) can be used to identify the scene and objects within the image, converting this information into textual data.
- For video files, the video is first converted into a sequence of images, and the same approach used for image files is applied to extract information and generate text.

- **Scaling:**
  - Using **load balancers** ensures incoming user queries are evenly distributed across multiple RAG pipeline instances, preventing bottlenecks and maintaining smooth operations.

- Having **distributed and sharded vector databases**, such as FAISS, allows efficient management of large datasets and optimizes retrieval performance.
- Applying **index compression** techniques like quantization reduces embedding sizes, improving retrieval speed and storage efficiency.
- Using **distilled models**, which are smaller and more optimized versions of embedding and generation models, helps minimize computational overhead.
- Performing **batch processing** for embedding creation and similarity searches enhances efficiency and reduces processing costs.
- Implementing result **caching** for frequently occurring queries avoids redundant computations and accelerates response times.
- Adopting **approximate nearest neighbor (ANN)** techniques, such as **Hierarchical Navigable Small Worlds (HNSW)**, enables faster similarity searches with minimal accuracy trade-offs.

Challenges faced during the task:

1. Reference - Link
   - **Challenge:** Encountered a TesseractNotFoundError while using OCR (pytesseract) for text extraction.
   - **Solution:** Resolved the issue by installing Tesseract.

2. Reference - Link
   - **Challenge:** Faced an error indicating FFmpeg was not found while attempting to convert an audio file format.
   - **Solution:** Fixed the issue by installing FFmpeg.

Github Repositry: https://github.com/Ranjith-JS2803/RAG-System

Video demo: Link