Problem statement

Build a project in the insurance domain, similar to the project you saw in the ‘Retrieval Augmented Generation’ session. The goal of the project will be to build a robust generative search system capable of effectively and accurately answering questions from various policy documents. You may use LangChain or LlamaIndex to build the generative search application.

Challenges and Lessons learned

Challenges
- Complex PDF Structures: Handling diverse PDF formats.
- Insurance-Specific Terminology: Adapting models to insurance vocabulary.
- Answer Granularity: Finding the ideal balance in result length.
- API Costs: Optimising usage for cost-effectiveness.

Lessons Learned
- Thorough PDF Analysis: Pays off in robust extraction logic.
- Domain Adaptation: Crucial for language model accuracy.
- Hybrid Retrieval: Can enhance certain query types.
- Cost-Effective Architecture: Exploring a mix of APIs and local models.

System Design Overview - LangChain

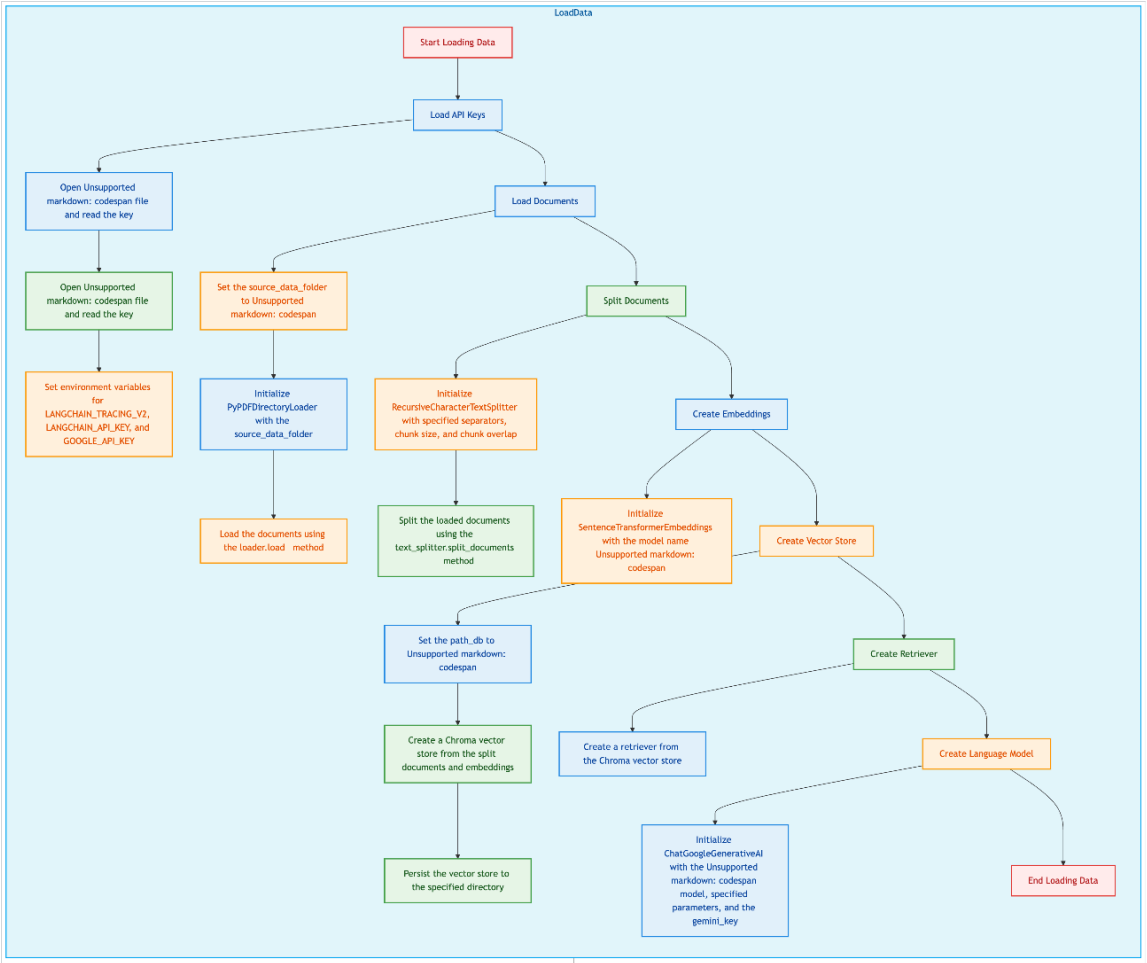
This system is designed to provide informative responses to user queries by leveraging a Retrieval Augmented Generation (RAG) approach. The system utilizes a combination of natural language processing techniques and a vector database to retrieve relevant information from a corpus of PDF documents and generate comprehensive answers.

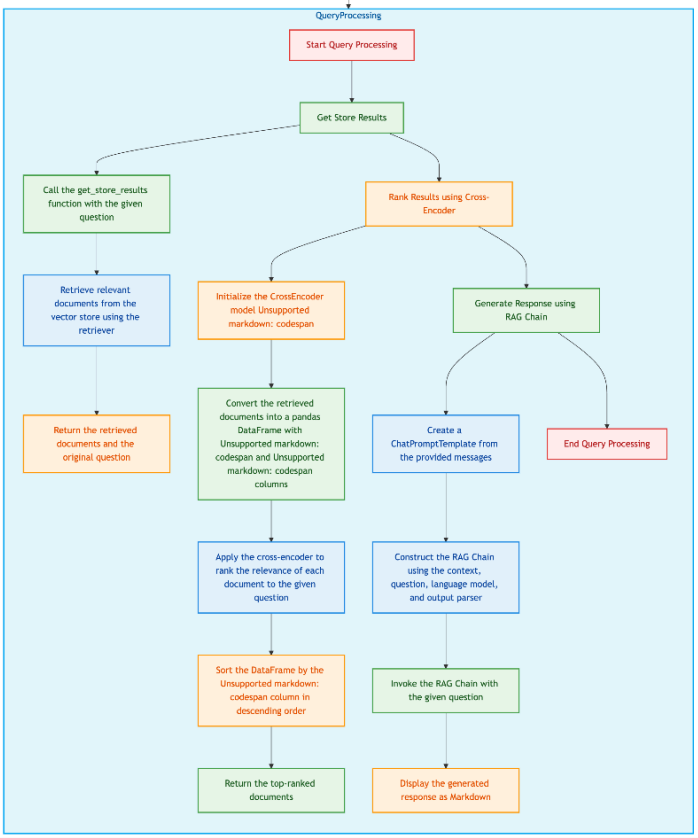
Components and Functionality

1. **Data Loading and Preprocessing:**
    - **PDF Loading:** The system loads PDF documents from a specified source folder.
    - **Text Extraction:** Text is extracted from the PDF documents.
    - **Text Splitting:** The extracted text is split into smaller chunks to improve retrieval efficiency.
    - **Embedding:** Each text chunk is embedded into a numerical vector representation using a SentenceTransformer model.
2. **Vector Database:**
    - **Creation:** A vector database (Chroma) is created to store the embedded text chunks.
    - **Indexing:** The embeddings are indexed in the database for efficient similarity search.
3. **Query Processing:**
    - **Input:** The user enters a query.
    - **Embedding:** The query is embedded into a numerical vector.
    - **Retrieval:** The vector database is searched for documents that are similar to the query embedding.
4. **Document Ranking:**
    - **Cross-Encoder:** A cross-encoder model is used to rank the retrieved documents based on their semantic similarity to the query.
5. **Response Generation:**
    - **Contextualization:** The ranked documents are provided as context to a language model (ChatGoogleGenerativeAI).
    - **Generation:** The language model generates a response based on the query and the provided context.

Workflow

1. The user enters a query.
2. The query is embedded into a numerical vector.
3. The vector database is searched for relevant documents.
4. The retrieved documents are ranked using a cross-encoder model.
5. The ranked documents are provided as context to a language model.
6. The language model generates a response based on the query and context.
7. The response is presented to the user.


LoadData
Start Loading Data
Load API Keys
Open Unsupported markdown: codespan file and read the key
Load Documents
Open Unsupported markdown: codespan file and read the key
Set the source_data_folder to Unsupported markdown: codespan
Split Documents
Set environment variables for LANGCHAIN_TRACING_V2, LANGCHAIN_API_KEY, and GOOGLE_API_KEY
Initialize PyPDFDirectoryLoader with the source_data_folder
Initialize RecursiveCharacterTextSplitter with specified separators, chunk size, and chunk overlap
Create Embeddings
Load the documents using the loader.load method
Split the loaded documents using the text_splitter.split_documents method
Initialize SentenceTransformerEmbeddings with the model name Unsupported markdown: codespan
Create Vector Store
Set the path_db to Unsupported markdown: codespan
Create Retriever
Create a Chroma vector store from the split documents and embeddings
Create a retriever from the Chroma vector store
Create Language Model
Persist the vector store to the specified directory
Initialize ChatGoogleGenerativeAI with the Unsupported markdown: codespan model, specified parameters, and the gemini_key
End Loading Data


QueryProcessing
Start Query Processing
Get Store Results
Call the get_store_results function with the given question
Rank Results using Cross-Encoder
Retrieve relevant documents from the vector store using the retriever
Initialize the CrossEncoder model Unsupported markdown: codespan
Generate Response using RAG Chain
Return the retrieved documents and the original question
Convert the retrieved documents into a pandas DataFrame with Unsupported markdown: codespan and Unsupported markdown: codespan columns
Create a ChatPromptTemplate from the provided messages
End Query Processing
Apply the cross-encoder to rank the relevance of each document to the given question
Construct the RAG Chain using the context, question, language model, and output parser
Sort the DataFrame by the Unsupported markdown: codespan column in descending order
Invoke the RAG Chain with the given question
Return the top-ranked documents
Display the generated response as Markdown

Key Technologies

- **Natural Language Processing:** SentenceTransformer for embedding, cross-encoder for ranking.
- **Vector Database:** Chroma.
- **Language Model:** ChatGoogleGenerativeAI.

Advantages

- **Relevance:** The RAG approach ensures that the generated responses are based on relevant information from the corpus.
- **Efficiency:** The vector database allows for efficient retrieval of similar documents.
- **Contextual Understanding:** The language model can leverage the provided context to generate more informative and accurate responses.

Limitations

- **Data Quality:** The quality of the responses depends on the quality of the data in the corpus.
- **Model Limitations:** The language model may have limitations in understanding complex queries or generating responses for certain topics.
- **Retrieval Efficiency:** For large datasets, retrieval efficiency can be a concern.

Future Improvements

- **Retrieval Optimization:** Explore techniques to improve retrieval efficiency for large datasets.

- **Model Fine-tuning:** Fine-tune the language model on specific domains or tasks to improve performance.
- **Contextual Augmentation:** Incorporate additional contextual information to enhance response generation.
- **Evaluation Metrics:** Develop appropriate metrics to evaluate the system's performance.

System Design Overview - LlamaIndex

Designed to answer questions based on a given set of documents. It leverages Hugging Face's Transformers library for embedding and Gemini for language modeling.

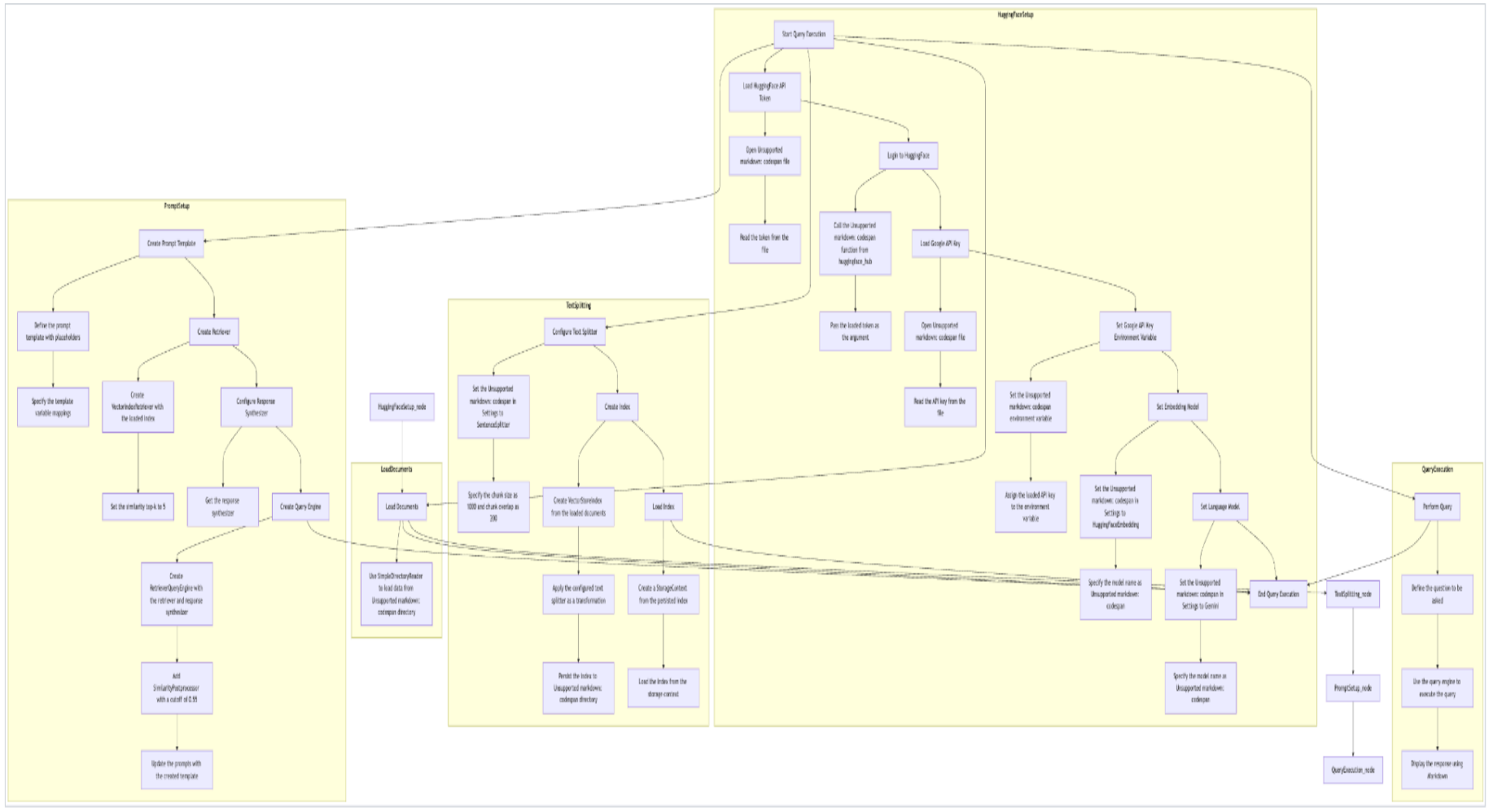
Key Components

- **Embedding:** Uses the `BAAI/bge-small-en-v1.5` model to create embeddings for the documents.
- **Language Model:** Employs the `Gemini` model for generating responses.
- **Document Indexing:** Creates a vector store index for efficient retrieval of relevant documents.
- **Query Processing:** Processes user queries by retrieving similar documents and generating responses.

System Architecture

Components:

1. **Document Ingestion:**
    - Reads documents from a specified directory.
    - Splits documents into smaller chunks for better indexing.
2. **Embedding:**
    - Embeds each document using the specified embedding model.
    - Creates a vector representation for each document.
3. **Indexing:**
    - Stores the embeddings and corresponding document metadata in a vector store index.
    - Optimises the index for efficient search.
4. **Query Processing:**
    - Receives a user query.
    - Embeds the query using the same embedding model.
    - Searches the index for the most similar documents.
    - Passes the relevant documents and query to the language model.
    - Generates a response based on the retrieved documents and query.



Further Improvements

- **Document Preprocessing:** Consider additional preprocessing steps like stemming, lemmatization, or stop word removal to improve search accuracy.
- **Hybrid Search:** Explore combining semantic search with keyword search for better results in certain cases.
- **Evaluation Metrics:** Implement metrics to evaluate the system's performance, such as accuracy, recall, and precision.
- **Scalability:** If dealing with large datasets, consider distributed indexing and query processing to improve performance.
- **Explainability:** Provide explanations for the generated responses to enhance user understanding and trust.

Additional Considerations

- **Data Privacy:** Ensure that sensitive information in the documents is handled securely.
- **Model Selection:** Evaluate different embedding and language models to find the best fit for your specific use case.
- **Error Handling:** Implement error handling mechanisms to gracefully handle unexpected situations.
- **Continuous Improvement:** Regularly update the system with new documents and consider retraining the models as needed.

By addressing these aspects, the system can be further optimised to provide more accurate and informative responses to user queries.