Semantic Spotter: Build a RAG System using Llama Index

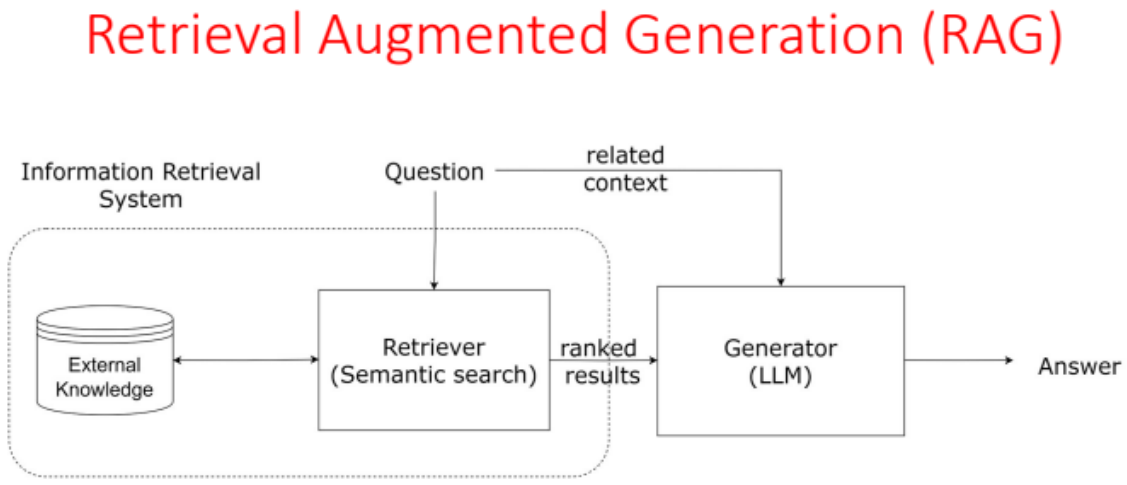
Introduction

The project is to build end-to-end project utilizing the diverse skills developed so far in the course.

Project Goal

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 using LlamaIndex to build the generative search application.

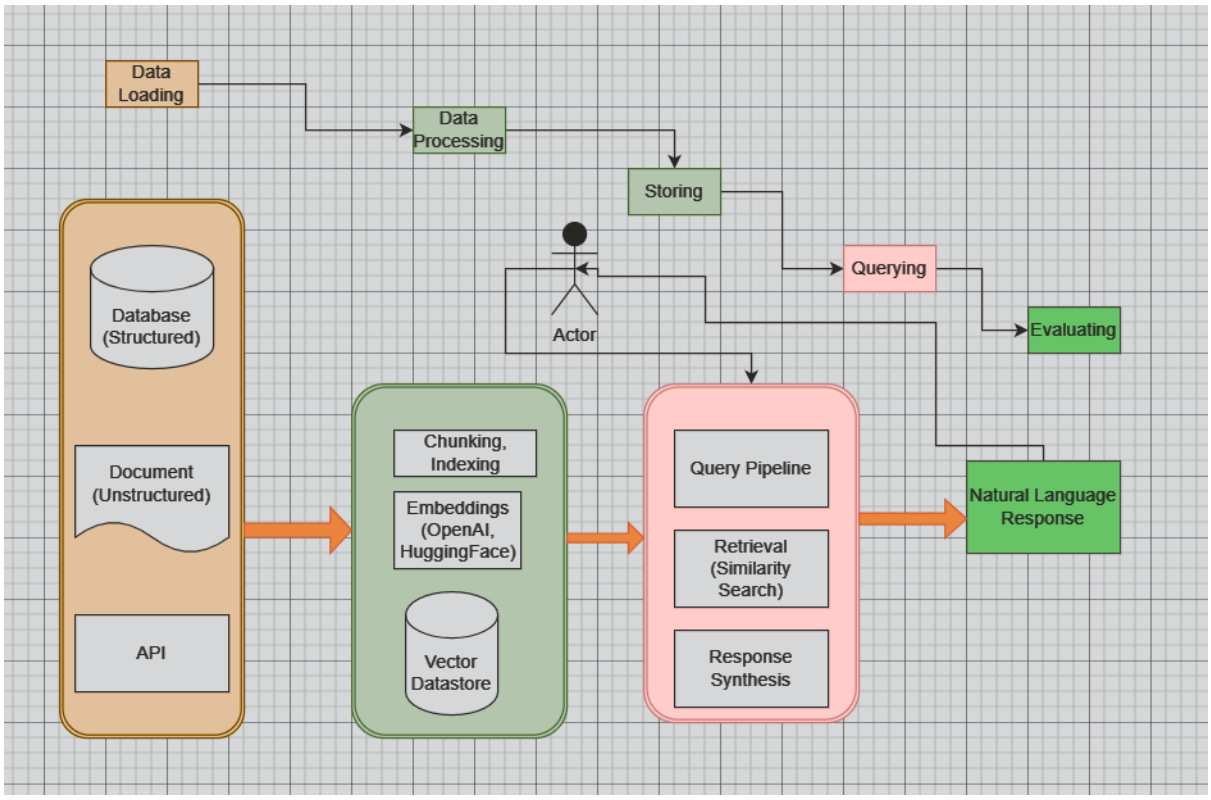
Introduction to RAG



LLMs are trained on enormous bodies of publicly available data but they aren't trained on proprietary **data** specific to our usecase. Retrieval-Augmented Generation (RAG) solves this problem by adding those data to the data LLMs already have access to.

Without RAG: LLM's don't have any information on the domain/enterprise specific query. With RAG: We are connecting an external data (ingested & transformed - embeddings - semantic search (dot product/cosine similarity) - top k relevant documents) - (context + user query)--> decorated using prompt engineering --> results

Solution Architecture



Here's a concise explanation of each RAG architecture component:

1. Data Sources:

- Database (Structured): Organized data in tables/schemas (SQL, MongoDB)
- Document (Unstructured): Raw text, PDFs, web pages
- API: Programmatic data access points

2. Data Processing:

- Chunking/Indexing: Breaks documents into smaller, manageable segments
- Embeddings: Converts text into vector representations using models like OpenAI or HuggingFace
- Vector Datastore: Stores and indexes embeddings for efficient similarity search (FAISS, Pinecone)

3. Query Pipeline:

- Query Processing: Transforms user questions into appropriate search queries
- Retrieval: Finds relevant chunks using similarity search

- Response Synthesis: Generates coherent answers using retrieved context and LLM

4. Natural Language Response:

- Final output formatted as human-readable text
- Combines retrieved context with LLM-generated response

5. Evaluation:

- Measures response quality, relevance, and accuracy
- Tracks retrieval performance and system metrics

6. Actor (User) Interaction:

- Submits queries to the system
- Receives natural language responses
- Can provide feedback for system improvement

This architecture ensures efficient information retrieval and contextual response generation using both existing knowledge and LLM capabilities.

Build RAG System in Insurance Domain using LlamaIndex

Step 1 : Import the necessary libraries

Here we need to Install and load the required libraries like llama-index, pdfplumber, openai etc.,

Step 2 : Mount Google Drive and Set the API key

We need to set the appropriate API key and load it into the openai instance

Step 3 - Data Loading

Since we might have multiple files, we will use Simple Directory Reader Just need to ensure that for reading each file type the necessary dependency libraries are already installed.

```
[25]  1  from llama_index.core import SimpleDirectoryReader
      2  from llama_index.readers.file import PDFReader

      1  # PDF Reader with `SimpleDirectoryReader`
      2  parser = PDFReader()
      3  input_directory = "/content/drive/MyDrive/Colab Notebooks/01. Assignments/6. Semantic Spotter Project/InsuranceDocuments/"
      4  file_path = "HDFC-Life-Sampoorna-Jeevan-101N158V04-Policy-Document.pdf"
      5  documents = parser.load_data(file=Path(input_directory + file_path))

[34]  1  # Output the loaded document
      2  print(f"Loaded {len(documents)} document(s)")

   Loaded 44 document(s)
```

Step 4 - Building the query engine

Let's break down the code here step by step and explain each part.

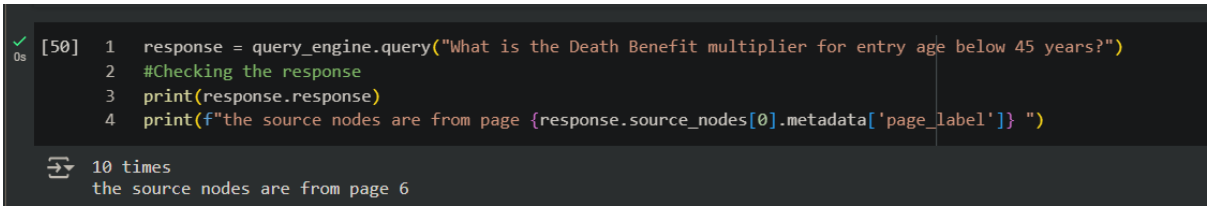
1. Parser Initialization using **SimpleNodeParser.from_defaults()**: This initializes a SimpleNodeParser with default settings. The SimpleNodeParser is used to break down documents into smaller, manageable nodes that can later be indexed and queried. These nodes are the building blocks of our index and serve as the smallest unit of information in our knowledge base. Nodes are typically segments of a document (e.g., sentences or paragraphs).
2. Creating Nodes from Documents:
    - **get_nodes_from_documents(documents)**: This function takes a list of documents (which you presumably loaded previously) and breaks each document into nodes (smaller chunks of text). It's useful because queries are typically done on nodes rather than entire documents.
    - The nodes variable will be a list of nodes that represent the documents in a more granular form, which can then be indexed and queried.
3. Creating Indices
   Here, I'm importing two types of indices from the llama_index.core package: SummaryIndex and VectorStoreIndex.
    - **SummaryIndex**: This index is designed to generate or store summaries of the documents. It's useful for generating concise answers or summaries from large sets of documents.
    - **VectorStoreIndex**: This is the more commonly used index for information retrieval, and it stores documents as vectors (numerical representations). It allows for efficient similarity search, which is crucial when you're building a query engine that retrieves information based on semantic similarity. The VectorStoreIndex uses vector embeddings to compare and retrieve relevant documents.
4. **Creating Query Engines**

Here, I'm converting the VectorStoreIndex and SummaryIndex into **query engines**.

- **index.as_query_engine()**: This method converts the VectorStoreIndex into a query engine. A query engine is a component that allows to query the indexed data. It provides an interface for querying the documents (or nodes) stored in the VectorStoreIndex. Typically, this query engine supports searching by semantic similarity, which is useful when we're querying for information based on meaning rather than exact matches.
- **summary_index.as_query_engine()**: Similarly, this converts the SummaryIndex into a query engine. The summary query engine will allow to query the summarized data (rather than the full-text data), which can be useful if we're only interested in high-level insights or answers.

Step 5 - Checking responses and response parameters using both VectorIndex and SummaryIndex

1. Using VectorIndex query engine



2. Using SummaryIndex query engine

Step 6 - Creating a response Pipeline

User receives the response and the document that they can refer to

```
[54]  ## Query response function
      def query_response(user_input):
        response = query_engine.query(user_input)
        file_name = response.source_nodes[0].node.metadata['file_name']
        page_no = response.source_nodes[0].metadata['page_label']
        final_response = response.response + '\n Check further at ' + file_name + ' document' +'page no' + page_no
        return final_response
```

```
      def initialize_conv():
        print('Feel free to ask Questions regarding Insurance policy HDFC Life Sampoorna Jeevan Plan. Press exit once you are done')
        while True:
          user_input = input()
          if user_input.lower() == 'exit':
            print('Exiting the program... bye')
            break
          else:
            response = query_response(user_input)
            display(HTML(f'<p style="font-size:20px">{response}</p>'))
```

```
initialize_conv()
```

```
Feel free to ask Questions regarding Insurance policy HDFC Life Sampoorna Jeevan Plan. Press exit once you are done
what is the policy name and what are the salient features of this policy ?
```

The policy name is HDFC Life Sampoorna Jeevan. Some of the salient features of this policy include comprehensive life cover, flexibility in choosing premium payment terms, options for additional riders for enhanced protection, and potential for bonuses and guaranteed additions. Check further at HDFC-Life-Sampoorna-Jeevan-101N158V04-Policy-Document.pdf documentpage no42

```
What are the Maturity Age Variants available for various variants of the policy ?
```

The Maturity Age Variants available for various variants of the policy are not explicitly mentioned in the provided context information. Check further at HDFC-Life-Sampoorna-Jeevan-101N158V04-Policy-Document.pdf documentpage no26

```
What are the Maturity Age Variants available?
```

Maturity Age Variants available are for policy terms ranging from 19 to 36 years and from 37 to 54 years. Check further at HDFC-Life-Sampoorna-Jeevan-101N158V04-Policy-Document.pdf documentpage no37

```
what are the variants of the policy ?
```

The variants of the policy mentioned in the context are the regular death claims and death claims arising out of accidents or unnatural deaths. Check further at HDFC-Life-Sampoorna-Jeevan-101N158V04-Policy-Document.pdf documentpage no19

```
What are the different Guaranteed Benefit Options available?
```

The different Guaranteed Benefit Options available are Lump Sum Option, Income Option, Lump Sum with Income Option, and Income with Lump sum Option. Check further at HDFC-Life-Sampoorna-Jeevan-101N158V04-Policy-Document.pdf documentpage no7

```
How many Bonus Options are available in the policy?
```

There are five Bonus Options available in the policy. Check further at HDFC-Life-Sampoorna-Jeevan-101N158V04-Policy-Document.pdf documentpage no9

```
can you explain the five bonus options available in the policy ?
```

The five bonus options available in the policy are as follows: 1. Bonus Option 1: Simple Reversionary Bonus for Term 2. Bonus Option 2: Simple Reversionary Income Bonus 3. Bonus Option 3: Cash Bonus 4. Bonus Option 4: Simple Reversionary Bonus for Premium Payment Term and Cash Bonus thereafter 5. Bonus Option 5: Simple Reversionary Income Bonus and Cash Bonus Check further at HDFC-Life-Sampoorna-Jeevan-101N158V04-Policy-Document.pdf documentpage no9

Summary of the Code Flow

1. **Document Parsing**: I start by parsing the documents into smaller chunks (nodes) using SimpleNodeParser. This helps break down the documents into more digestible units for querying.
2. **Indexing**: Then, I create two types of indices:
    - **VectorStoreIndex**: This is used for semantic search. The nodes are stored as vectors, and we can query these vectors based on similarity.
    - **SummaryIndex**: This is used for generating and storing summaries of documents.
3. **Query Engines**: Once the indices are created, I convert them into query engines, which allows us to interact with the indices and retrieve relevant data based on the queries sent.

Conclusion:

Upon comparison of the generated responses with the manual checking, the response is almost appropriate for the properly given query. Always response depends on the quality of query input that is fed to the system. Much more LLM integration with the quality input/query can make the system robust and close to the human thinking.