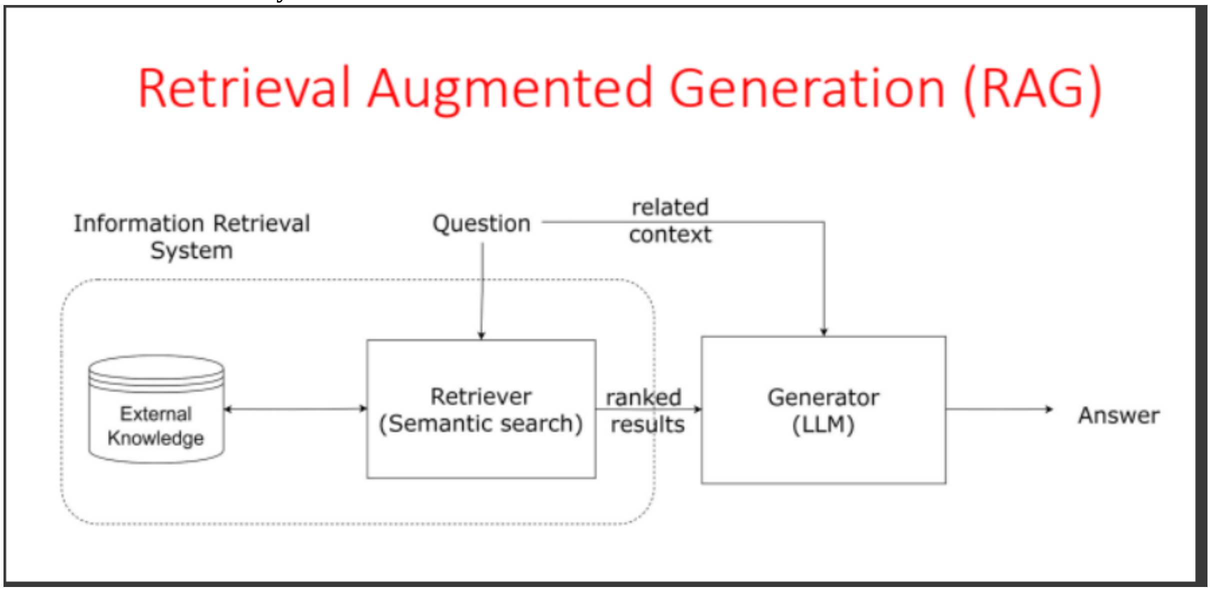
PROJECT REPORT
HELPMATE AI PROJECT – Retrieval Augmented Generation with LlamaIndex

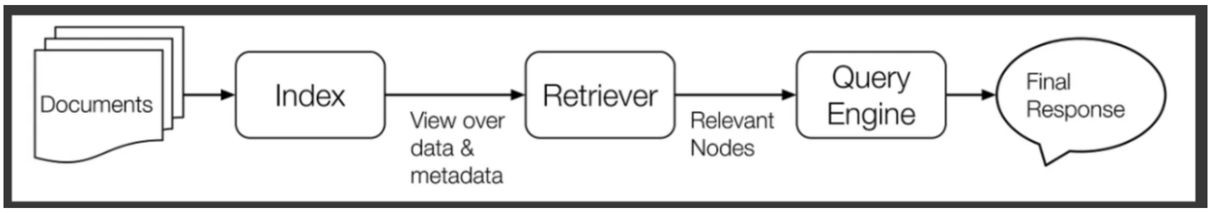
1. **Problem Statement:**
This project is focused on the insurance industry, aiming to create a highly efficient generative search system that can precisely answer questions derived from a variety of insurance policy documents. We are utilizing LlamaIndex, a data integration framework that helps organize, structure, and access private or domain-specific datasets to achieve this. LlamaIndex facilitates the connection of large language models (LLMs), like GPT-3, with specialized data sources. The tool is a Python library designed to assist in the development of data-aware applications and question-answering systems.

2. **Why Llama Index:**
LlamaIndex offers seamless integration of data from various file formats, such as text documents, websites, PDFs, and structured files like CSVs. After importing the data, it generates an index that optimizes data retrieval and querying. The platform also provides a query engine, enabling users to interact through natural language to obtain relevant answers from the indexed content. LlamaIndex is well-suited for this project because it supports multiple retrieval strategies, including vector-based searches and keyword-based queries, which can ensure the most pertinent results are retrieved based on the given input.

3. **Project Goals:**
The main objectives of the project are:
   - To develop a highly functional generative search system that answers questions accurately based on insurance policy documents.
   - Leverage LangChain or LlamaIndex to create this generative search solution.

4. **System Design**
RAG System Architecture


Retrieval Augmented Generation (RAG)
Information Retrieval System
Question
related context
External Knowledge
Retriever (Semantic search)
ranked results
Generator (LLM)
Answer

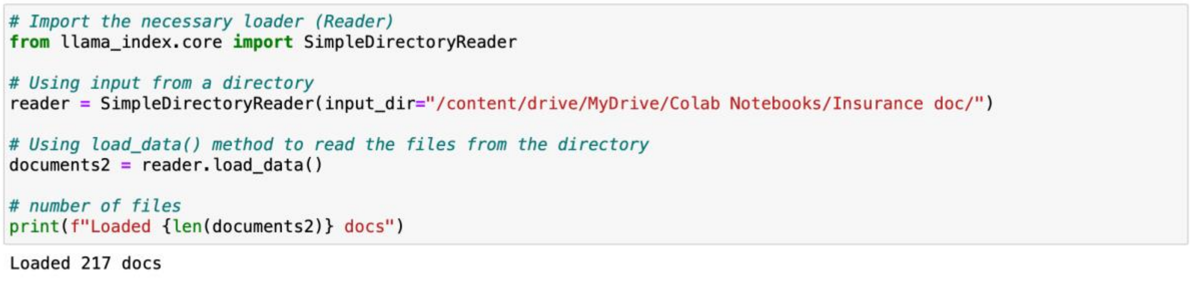


Components:

- Data Loading
- Building query engine
- Creating a Response Pipeline
- Build a Testing Pipeline
- Optimization and fine-tuning

5. Project Execution

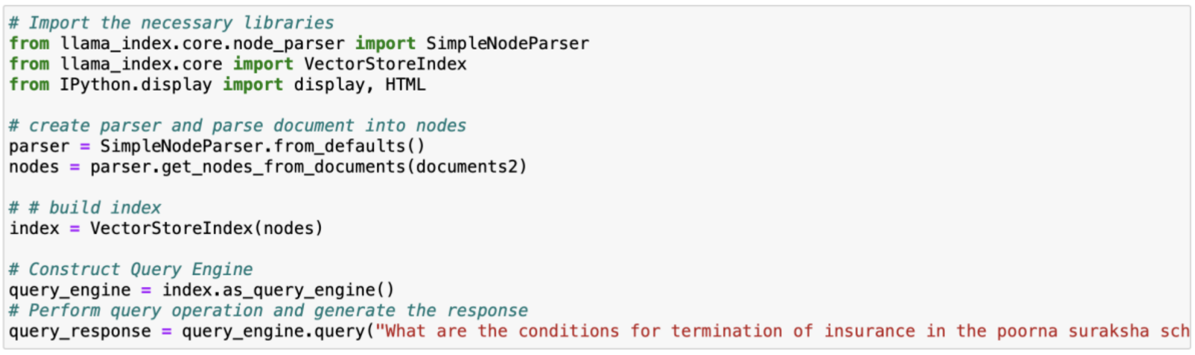
- **Data Loading:**
  The Simple Directory Reader was employed to extract data from PDFs and store it in variables for further processing.



```
# Import the necessary loader (Reader)
from llama_index.core import SimpleDirectoryReader

# Using input from a directory
reader = SimpleDirectoryReader(input_dir="/content/drive/MyDrive/Colab Notebooks/Insurance doc/")

# Using load_data() method to read the files from the directory
documents2 = reader.load_data()

# number of files
print(f"Loaded {len(documents2)} docs")
```

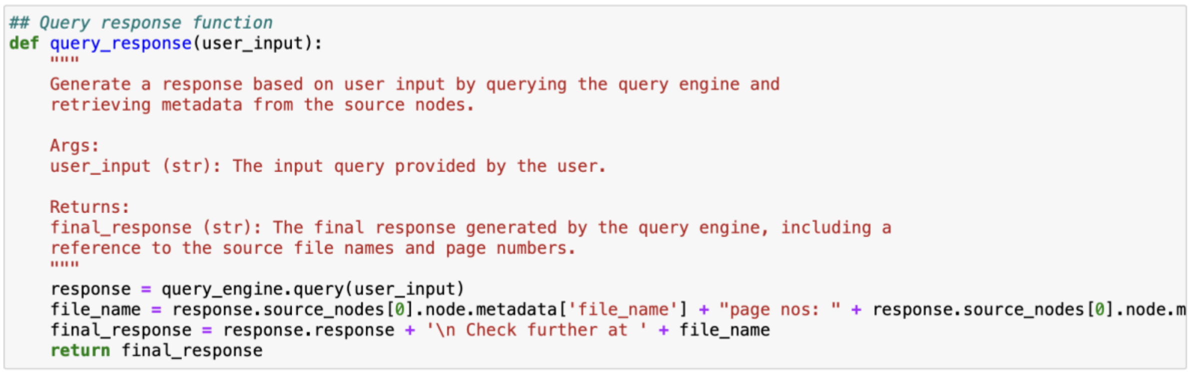
```
Loaded 217 docs
```

- **Building query engine:**
  The Query Engine combines a Retriever and a Response Synthesizer, functioning as a pipeline. It utilises the user's query to identify relevant nodes and processes them using a large language model (LLM) to generate an appropriate response.

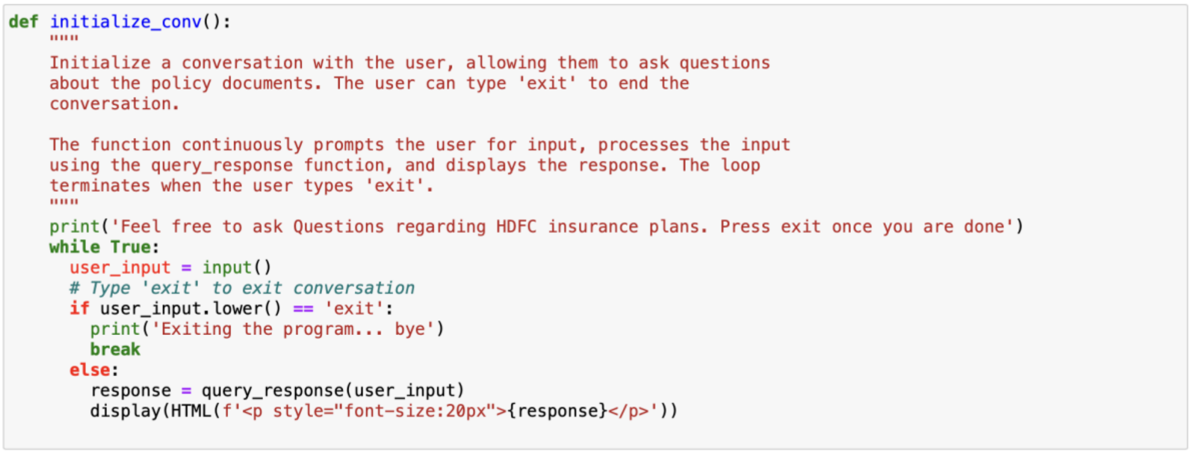


```
# Import the necessary libraries
from llama_index.core.node_parser import SimpleNodeParser
from llama_index.core import VectorStoreIndex
from IPython.display import display, HTML

# create parser and parse document into nodes
parser = SimpleNodeParser.from_defaults()
nodes = parser.get_nodes_from_documents(documents2)

# # build index
index = VectorStoreIndex(nodes)

# Construct Query Engine
query_engine = index.as_query_engine()
# Perform query operation and generate the response
query_response = query_engine.query("What are the conditions for termination of insurance in the poorna suraksha sch
```

- **Response Pipeline Creation:**
  The Response Pipeline integrates functions such as query response and initialize_conv, providing a structured approach to handle queries and generate interactive chatbot responses.



```
## Query response function
def query_response(user_input):
    """
    Generate a response based on user input by querying the query engine and
    retrieving metadata from the source nodes.

    Args:
    user_input (str): The input query provided by the user.

    Returns:
    final_response (str): The final response generated by the query engine, including a
    reference to the source file names and page numbers.
    """
    response = query_engine.query(user_input)
    file_name = response.source_nodes[0].node.metadata['file_name'] + "page nos: " + response.source_nodes[0].node.m
    final_response = response.response + '\n Check further at ' + file_name
    return final_response
```



```
def initialize_conv():
    """
    Initialize a conversation with the user, allowing them to ask questions
    about the policy documents. The user can type 'exit' to end the
    conversation.

    The function continuously prompts the user for input, processes the input
    using the query_response function, and displays the response. The loop
    terminates when the user types 'exit'.
    """
    print('Feel free to ask Questions regarding HDFC insurance plans. Press exit once you are done')
    while True:
      user_input = input()
      # Type 'exit' to exit conversation
      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>'))
```

- **Build a Testing Pipeline**:
  Three distinct queries were designed and stored in a list for use in the testing_pipeline function, which is responsible for evaluating the system's response capabilities.

- **Optimization and fine tuning:**
  Based on the feedback from the testing phase, performance improvements can be made, such as creating a tailored prompt template, refining custom notes, developing a sub-question query engine, and leveraging Agentic Systems for RAG enhancement.

6. **Challenges:**
    1. **Extracting Data from PDFs:**
       The Simple DirectoryReader was used to streamline the extraction process, automatically processing all PDF files from the target folder.
    2. **Indexing Large Documents:**
       Efficient indexing strategies were implemented, with considerations for distributed or incremental indexing for exceptionally large datasets.
    3. **Managing Domain-Specific Terminology:**
       The model was fine-tuned on a corpus of insurance-related materials to enhance its ability to understand and generate domain-specific content.
    4. **Ensuring Accurate Document Retrieval:**
       The retrieval mechanism was optimized through advanced techniques, such as SentenceSplitter from llama.index.core.node_parser, to improve relevance.
    5. **Integrating Retrieved Data with Queries:**
       A structured method was developed to combine the query and retrieved documents, ensuring the input remains within the language model's token limit.

7. **Key Takeaways**
    1. **Data Extraction is Critical:**
       Efficient and accurate extraction of data plays a pivotal role in ensuring the success of the system.
    2. **Preprocessing is Vital:**
       Data preprocessing significantly impacts the quality of results, necessitating careful attention.
    3. **Optimization of Retrieval Mechanisms:**
       The retrieval system should be continually improved to enhance the relevance of the results.
    4. **Managing Input Length Constraints:**
       Keeping inputs within the model’s token limit is essential to ensure smooth processing.
    5. **Evaluation Requires a Comprehensive Approach:**
       Effective evaluation of the system needs to take multiple factors into account to ensure it meets the project’s objectives.
    6. **Sensitive Information Handling:**
       Protecting sensitive data and ensuring privacy should be a top priority throughout the system’s development.