Empowering Connections:

Unleashing the Potential of **AI**

HelpMate AI Project

(Business Analytics)

IIIT-B , Jan 2024

- Pallav Rajput

Introduction to HelpMate AI

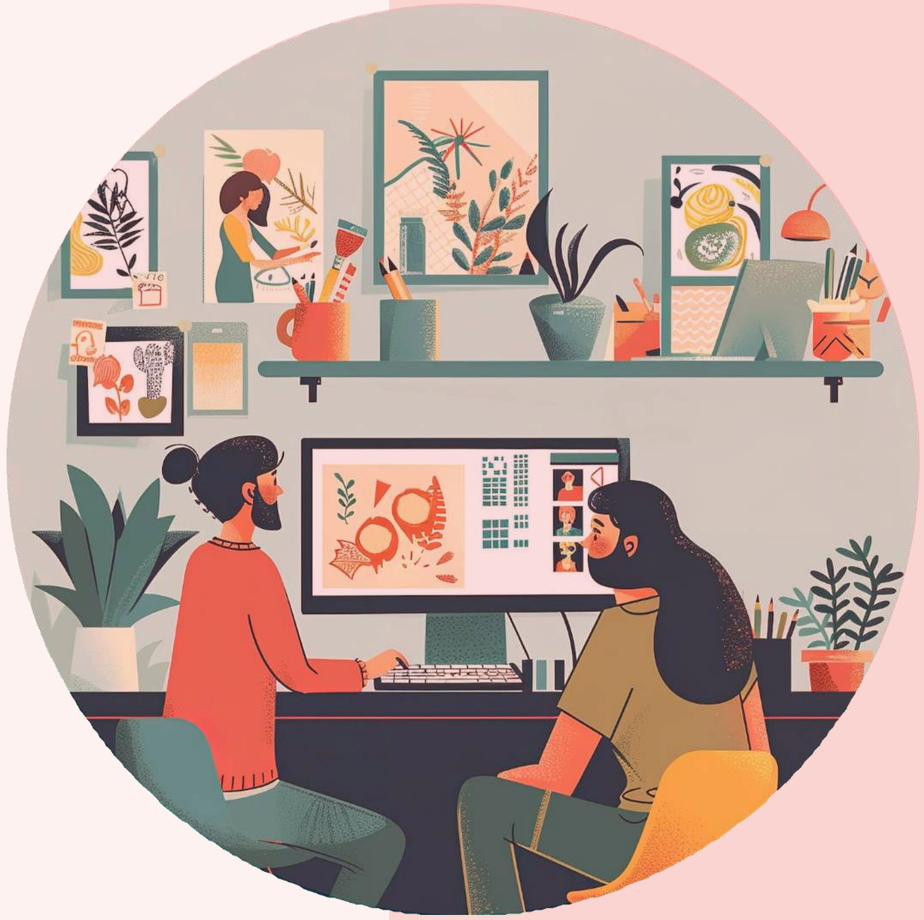
A project in the insurance domain, based on **‘Retrieval Augmented Generation with LlamaIndex’** with the goal of building a robust generative search system capable of effectively and accurately answering questions from various insurance policy documents. The LlamaIndex is used to build the generative search application that accurately answers questions from a policy document. .

What is RAG ?

(Retrieval Augmented Generation)

Retrieval Augmented Generation (RAG) is a technique that enhances the capabilities of language models by incorporating external knowledge sources during the generation process. This allows the model to access and utilize up-to-date and factual information, leading to more comprehensive and reliable outputs.

Project Goals



The primary objective of this project is to build a Retrieval-Augmented Generation (RAG) pipeline that enables efficient querying and response generation from policy documents. By combining a robust query engine with an interactive chatbot interface, the project aims to:

- Provide quick and accurate answers to user queries.
- Reference source documents and metadata for reliability.
- Support iterative improvement through a testing pipeline.

Data Sources

The input data comprises policy documents in PDF format, which are loaded and processed using a SimpleDirectoryReader.
These documents are parsed into nodes, indexed, and queried to generate responses.

Key Data Sources:

- Policy documents stored locally (e.g., Policy.pdf).
- Metadata such as file names and page numbers for contextual responses.

Design Choices

➢ **Frameworks and Libraries**

- LlamaIndex: Used for document parsing, node generation, and indexing.
- OpenAI GPT API: Powers the response generation with conversational AI.
- PyMuPDF: Supports additional document manipulation.

➢ **Query Pipeline**

- Documents are parsed into nodes using a SimpleNodeParser.
- Nodes are indexed using VectorStoreIndex for efficient querying.
- Queries are processed by the query engine, retrieving relevant nodes and generating responses.

➢ **Interactive Chatbot**

- Implements an interactive interface for user queries. Handles iterative queries and provides feedback-driven testing.

➢ **Custom Prompt**

- Template Utilized to refine response quality and tailor answers to specific user needs.

Challenges Faced

- **Data Preprocessing –**
  - Ensuring clean parsing of complex PDF structures.
  - Handling edge cases such as missing metadata or unsupported formats.

- **Response Accuracy -**
  - Balancing retrieval precision and generative quality in responses.
  - Improving relevance through prompt engineering and feedback loops.

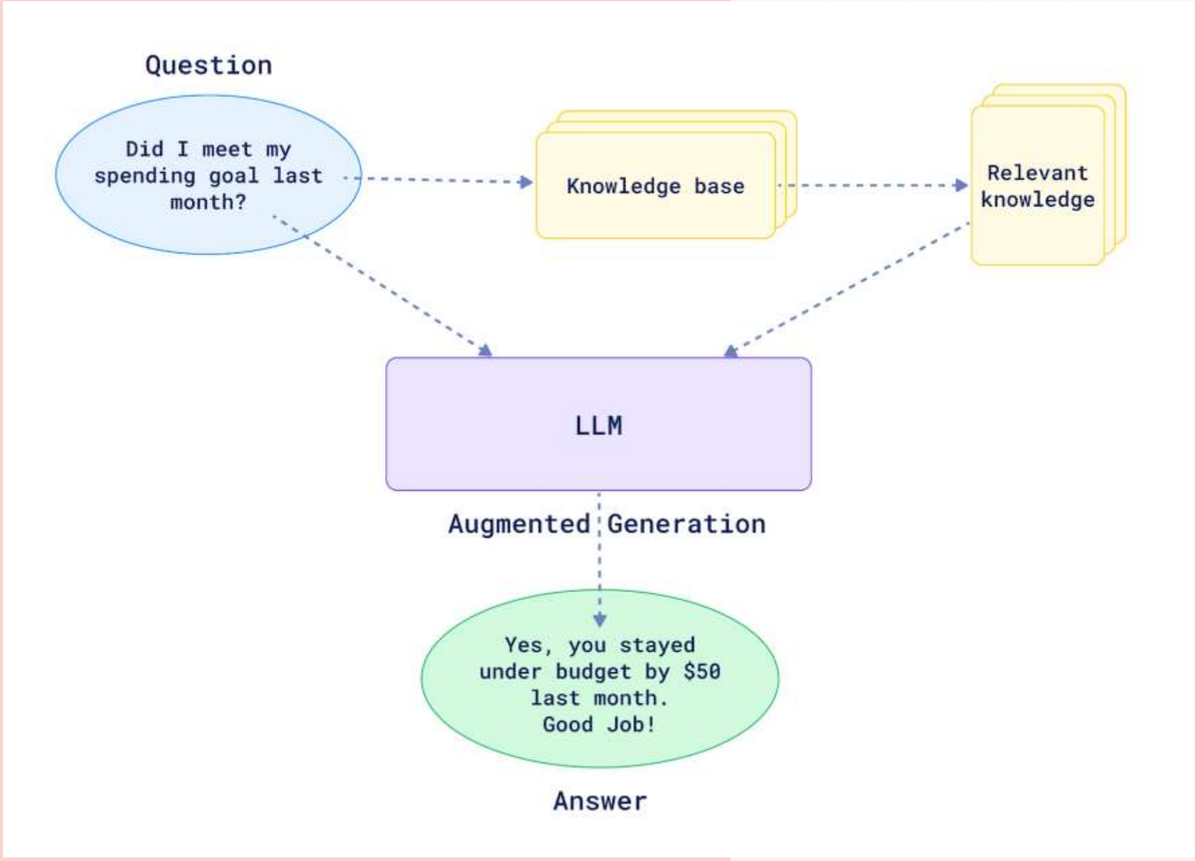
- **Scalability –**
  - Optimizing the pipeline for larger datasets and concurrent user queries.

System Architecture

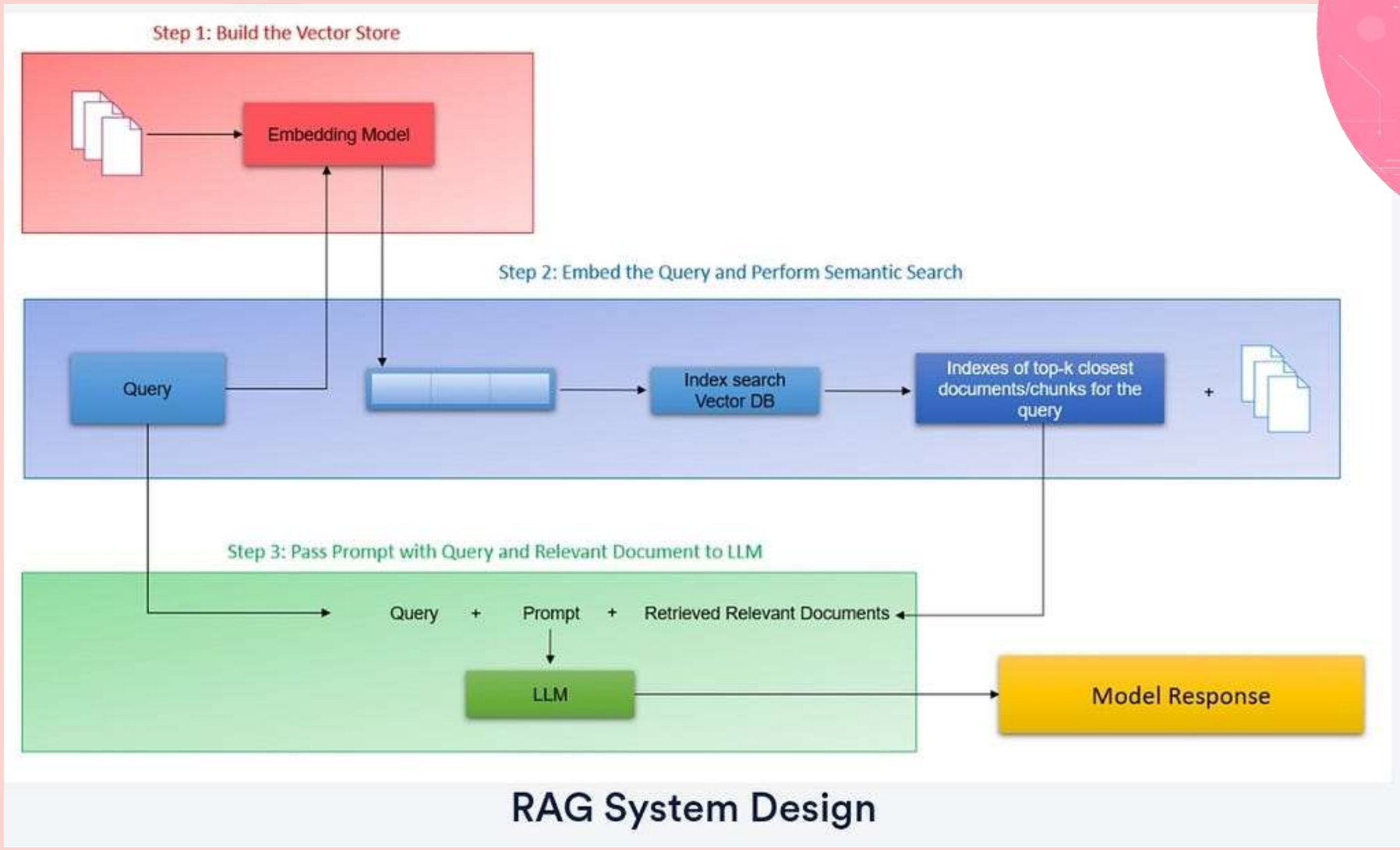
The system design includes the following layers:

Input Layer: Handles document ingestion and parsing.
Indexing Layer: Generates nodes and indexes them for efficient querying.
Query Engine: Processes user queries, retrieves relevant nodes, and generates responses.
Interactive Layer: Provides a chatbot interface for user interaction.

Flowchart
Input Data → Document Parsing → Node Generation → Index Creation → Query Engine → Response Generation.



System Design Flowchart



RAG System Design

Python Code

Step 1 : Import the necessary libraries

```
# Installing OpenAI, LlamaIndex
!pip install -U -qq llama-index openai
```

```
## Installing additional supporting libraries as required
```

```
# Importinging the LLAMA libraries
from llama_index.llms.openai import OpenAI
from llama_index.core.llms import ChatMessage
```

```
# Importing common libraries
import os, openai
import pandas as pd, numpy as np
```

Step 2: Set the API key

Key Toggle

```
api_key = 'sk-proj-v9xuO2pCmHDsNABs
```

```
%%writefile "config.py"
api_key = api_key
api_key
```

```
Overwriting config.py
```

```
#from config import api_key
openai.api_key = api_key
openai.api_key
```

Step 3 - Data Loading

```
# Importing the necessary loader
from llama_index.core import SimpleDirectoryReader
```

```
# Initiating the reader with the input directory to the files

reader = SimpleDirectoryReader(
    input_files=[r"C:\Users\ribhu\Downloads\Policy\Policy.pdf"]
)
documents = reader.load_data()

print(f"Loaded {len(documents)} docs")
```

```
Loaded 44 docs
```

Step 4 - Building the query engine

The general process for creating the query_engine is:
- Load the documents
- Create nodes from the documents
- Create index from documents
- Initialise the Query Engine
- Query the index with the prompt
- Generate the response using the retrieved nodes

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


# Creating parser / parse document to nodes
parser = SimpleNodeParser.from_defaults()
nodes = parser.get_nodes_from_documents(documents)

# Index building
index = VectorStoreIndex(nodes)

# Constructing the Query Engine
query_engine = index.as_query_engine()
```

Step 5 - Creating a Response Pipeline

A Query Response pipeline encapsulates all the necssary steps to build a RAG pipeline. Mo
`initialize_conv()` below. The `query_response` functions return the query response
and the `initialize_conv()` function creates an interactive chatbot.

```
## 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.
    response = query_engine.query(user_input)
    file_name = response.source_nodes[0].node.metadata['file_name'] + " page r

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


    # final_response (str): The final response generated by the query engine,
    # reference to the source file names and page numbers.
    final_response = response.response + '\n Check further at ' + file_name
    return final_response
```

```
def initialize_conv():
    """
```

Step 6 Initializing a conversation

```
def initialize_conv():
    """
    Initializing a conversation with the user, allowing them to ask questions
    about the policy documents. The user can type 'exit' to end the
    conversation.
    """
    print('Please ask queries about the policy, and exit when you are satisfied')
    while True:
        user_input = input()
        if user_input.lower() == 'exit':
          print('Hope your queries are resolved... Thank you.. bYe...')
          break
        else:
          response = query_response(user_input)
    return response
```

```
initialize_conv()
```

```
Please ask queries about the policy, and exit when you are satisfied
 What is the maximum age for policy?
 exit
Hope your queries are resolved... Thank you.. bYe...
'The maximum age for the policy is 54 years.\n You can check the file Policy.pdf on page no. 26,32'
```

Step 7 - Build a Testing Pipeline

Here we feed a series of questions to the Q/A bot and store the responses along with the feedback on whether it's accurate or not from the user. Creating questions and store them in the `questions` list to be queried by the RAG system using the `testing_pipeline` function.

```
# Args:     questions (list): A list of questions to be tested.
questions = ['How much will the policy cost?',  'What\'s covered if I become disabled?',
             'Will my premiums change over time?',  'Will the policy cover hospitalize bills?',
             'Will I get the reimbursement if i cancel the policy?'
            ]
```

```
def testing_pipeline(questions):
    # Conduct a testing pipeline for a series of questions, collecting user feedback on the responses provided by the bot.

    test_feedback  = []
    for i in questions:
        print(i)
        print(query_response(i))
        print('\n Your feedback for the bot is appriciated')
        user_input = input()
        page = query_response(i).split()[-1]
        test_feedback.append((i,query_response(i),page,user_input))
    #   Returns:  pd.DataFrame: A DataFrame containing the questions, their corresponding responses, the page number from the
    #   indicating whether the response was good or bad.
    feedback = pd.DataFrame(test_feedback, columns =['Question', 'Response', 'Page','Good or Bad'])
    return feedback
```

	Question	Response	Page	Good or Bad
0	How much will the policy cost?	The policy cost will depend on various factors...	33,32	Good
1	What's covered if I become disabled?	If you become disabled, the documents required...	19,7	Good
2	Will my premiums change over time?	Premiums will not change over time if the Poli...	10,7	Good
3	Will the policy cover hospitalize bills?	The policy does not mention coverage for hospi...	19,10	Good
4	Will I get the reimbursement if i cancel the p...	If you cancel the policy within the specified ...	11,10	Good

Conclusion

This project showcases **a Retrieval-Augmented Generation (RAG) system** tailored for querying **insurance policy documents**. By leveraging **LlamaIndex and OpenAI GPT**, the system provides accurate, context-aware answers to user queries. Its ability to reference specific sections within policy documents enhances reliability, making it a valuable tool for understanding complex insurance policies. With further enhancements, this project has the potential to streamline policy analysis and improve customer interactions in the insurance sector.

Thanks!

Pallav Rajput
+91-9888143002
ribhu.s18@gmail.com