Project Report

1. Introduction

1.1 Problem Statement

The core objective of this project is:

“Build a Retrieval-Augmented Generation (RAG) system to help users answer queries on policy documents.”

In the insurance domain, policy documents are:

- Long and complex
- Filled with dense legal terminology
- Difficult for customers and even employees to interpret quickly

A system capable of answering natural-language questions about these documents can:

- Improve customer support efficiency
- Reduce dependency on human agents
- Increase accuracy and consistency of information
- Empower users to self-serve

A RAG system is ideal here because it combines:

- **Accurate retrieval of facts from the actual policy documents**
- **Natural-language generation from an LLM**

This ensures the responses are both *grounded* and *explainable*—critical requirements in insurance data handling.

---

2. System Design

The RAG architecture follows the workflow illustrated in the **System Design – LLM** reference.

2.1 Architecture Overview

The system comprises five major components:

1. **Data Ingestion**
2. **Indexing**
3. **Retrieval**
4. **Generation (LLM)**
5. **Post-Processing (Reranking + Caching)**

Below is a detailed description of each.

2.2 Data Ingestion

- Policy PDFs and DOCX files are stored in **`/content/drive/MyDrive/hdfc-insurance-policy/policy-docs`**
- They are loaded using `SimpleDirectoryReader`
- All documents are converted into text and passed into the indexing layer

---

2.3 Indexing (VectorStoreIndex)

- Once documents are loaded, they are chunked into “nodes” or embeddings.
- These nodes are stored in a vector index using llama-index’s `VectorStoreIndex`.
- This enables fast semantic search when a query is received.

Indexing is essential because:

- It transforms unstructured insurance documents into machine-searchable embeddings.
- It allows semantic retrieval instead of simple keyword search.

2.4 Retrieval

When the user submits a question:

- The query is converted into an embedding vector.
- These nodes represent the **most relevant portions** of the policy documents.

To improve precision, the system uses:

- **CohereRerank post-processor**
- **similarity_top_k** parameter to control how many nodes are returned

---

2.5 Generation (LLM)

The selected nodes are passed to:

- **OpenAI GPT-3.5-Turbo**

Custom prompt templates guide the LLM:

textQATemplate

This template provides:

- Context
- User query
- Instructions on how to answer based on the context

refinedTemplate

This template adds:

- Existing answer
- Additional context
- Opportunity to refine response

Prompt engineering ensures that answers:

- Are grounded in retrieved facts
- Are coherent and helpful
- Avoid hallucinations

2.6 Post-Processing

Two layers are applied:

1. **Reranking (CohereRerank)**
   - Ensures the best matches are prioritized after retrieval.
2. **Caching (diskcache)**
   - Stores previous results to accelerate repeated queries.

---

3. Technologies Used

3.1 Key Libraries and Tools

Library / Tool	Purpose
llama-index	Core framework for building RAG-based applications, including document ingestion, indexing, and retrieval.

llama-index-llms-openai	Provides integration between llama-index and OpenAI LLMs.
openai	Enables calling the GPT-3.5-Turbo model for generation.
pypdf	Used for parsing and extracting text from PDF policy documents.
docx2txt	Used for extracting text from DOCX files.
llama-index-postprocessor-cohere-rerank	Reranks retrieved nodes for improved answer accuracy.
diskcache	Key-value caching system for storing and retrieving pre-computed responses.

Each component is essential in implementing a complete and efficient RAG pipeline.

---

4. Data Ingestion and Processing

4.1 Directory Structure

Documents are placed in:

```
/content/drive/MyDrive/hdfc-insurance-policy/policy-docs
```

4.2 Loading Documents

Using:

```
SimpleDirectoryReader('policy-docs').load_data()
```

- Reads all PDF and DOCX files
- Extracts and normalizes text
- Produces a list of structured Document objects

4.3 Building the Vector Index

The retrieved documents are passed to:

```
VectorStoreIndex.from_documents(documents)
```

This step:

- Embeds text into vector space
- Constructs semantic-search-friendly index
- Prepares data for fast retrieval during user query time

---

5. Query Engine Configuration and Prompt Engineering

5.1 Query Engine Configuration

The query engine uses:

- **OpenAI(model="gpt-3.5-turbo")**
- **Retriever settings with similarity_top_k**
- **CohereRerank post-processor**

This creates a balance between:

- Retrieval accuracy
- LLM reasoning ability
- Response speed

5.2 Prompt Engineering

textQATemplate

Purpose:

- Provide the context chunks
- Guide the LLM to answer using retrieved information
- Allow fallback to general knowledge when needed

Structure:

```
Context information is provided below
---------------------
{contextStr}
---------------------
Using both the context information and your own knowledge,
answer the question: {queryStr}
If the context isn't helpful, you may answer on your own.
```

refinedTemplate

Used during response refinement:

```
The original question is as follows: {queryStr}
We have provided an existing answer: {existingAnswer}
We have the opportunity to refine the earlier answer with some more
context below.
------------
{contextMsg}
------------
Using both the new context and your own knowledge, update or repeat
the existing answer.
```

Prompt engineering ensures:

- Structured responses
- Context alignment
- Minimal hallucination risk

---

6. Caching Mechanism

6.1 Use of diskcache

The system uses:

```
./gpt_cache
```

as the caching directory.

6.2 Cache Operations

Set cache

```
cache.set(user_input, response)
```

Stores:

- Query string as key
- Response object as value

Get cache

```
cache.get(user_input)
```

If present, returns a cached response instantly.

6.3 Benefit of Caching

- Reduces repeated LLM calls
- Decreases operational cost
- Improves system latency and responsiveness

---

7. Validation and Feedback Loop

7.1 validatePipeline Function

This function:

- Iterates through a list of test questions
- Displays the RAG-generated answer
- Asks the user for feedback (“Good” or “Bad”)
- Stores results for later analysis

7.2 Output DataFrame (feedbackDf)

Columns include:

1. **Question** – User’s or tester’s query
2. **Response** – RAG system’s answer
3. **User Feedback** – Evaluation of answer quality

Importance

- Helps track accuracy
- Identifies weak document chunks
- Guides improvements in prompts, indexing, or reranking

---

8. Conclusion

The RAG system successfully integrates:

- Document ingestion
- Vector-based retrieval

- GPT-powered generation
- Cohere-based reranking
- Diskcache caching
- Human feedback collection

Key Achievements

- Built an end-to-end RAG pipeline tailored for insurance documents.
- Provided accurate, grounded answers by combining LLM reasoning with document retrieval.
- Introduced caching, increasing performance and reducing repeated model calls.
- Created a validation pipeline that enables continuous improvement.

Lessons Learned

- Prompt engineering significantly affects quality.
- Reranking helps avoid irrelevant retrieval.
- Caching is essential for real-world deployments.
- Insurance documents require careful chunking and preprocessing.

Future Implications

This system can be extended into:

- Customer-facing chatbots
- Internal support automation
- Compliance checking tools
- Automated policy comparison engines