Semantic Spotter: RAG Q&A System for Insurance Policies using LangChain

Background

In the insurance industry, life insurance policy documents are often lengthy and complex involving legal terminology. This makes it difficult for customers\policyholders to quickly find answers to specific questions such as eligibility, exclusions, claim procedures, or grace periods. Answers to user queries **exist in the documents**, but are often buried across different sections and may vary between policy types. Traditional search systems struggle to interpret natural language queries.

Semantic Spotter project aims to implement such a system for life insurance documents. By leveraging embedding models, vector databases like ChromaDB, cross-encoder reranking, LangChain framework and OpenAI's GPT models, the system delivers accurate, context-aware answers to user queries. Additionally, it implements a caching mechanism to optimize repeated query performance, improving both response speed and cost efficiency.

Problem Statement

The goal of the project is to build a robust generative search system capable of effectively and accurately answering questions from a bunch of policy documents using LangChain framework.

Objectives

To build a Retrieval-Augmented Generation (RAG) system that:

- Accepts user questions in natural language.
- Retrieves relevant sections from one or more insurance policy documents.
- Generates an accurate, grounded answer using only the retrieved content.

Why LangChain?

LangChain is a framework that simplifies the development of LLM applications LangChain offers a suite of tools, components, and interfaces that simplify the construction of LLM-centric applications. LangChain enables developers to build applications that can generate creative and contextually relevant content LangChain provides an LLM class designed for interfacing with various language model providers, such as OpenAI, Cohere, and Hugging Face.

LangChain is an open-source framework that makes it easier to build powerful and personalizeable applications with LLMs relevant to user's interests and needs. It connects to external systems to access information required to solve complex problems. It provides abstractions for most of the functionalities needed for building an LLM application and also has integrations that can readily read and write data, reducing the development speed of the application. LangChains's framework allows for building applications that are agnostic to the underlying language model. With its ever expanding support for various LLMs, LangChain offers a unique value proposition to build applications and iterate continuously.

LangChain framework consists of the following:

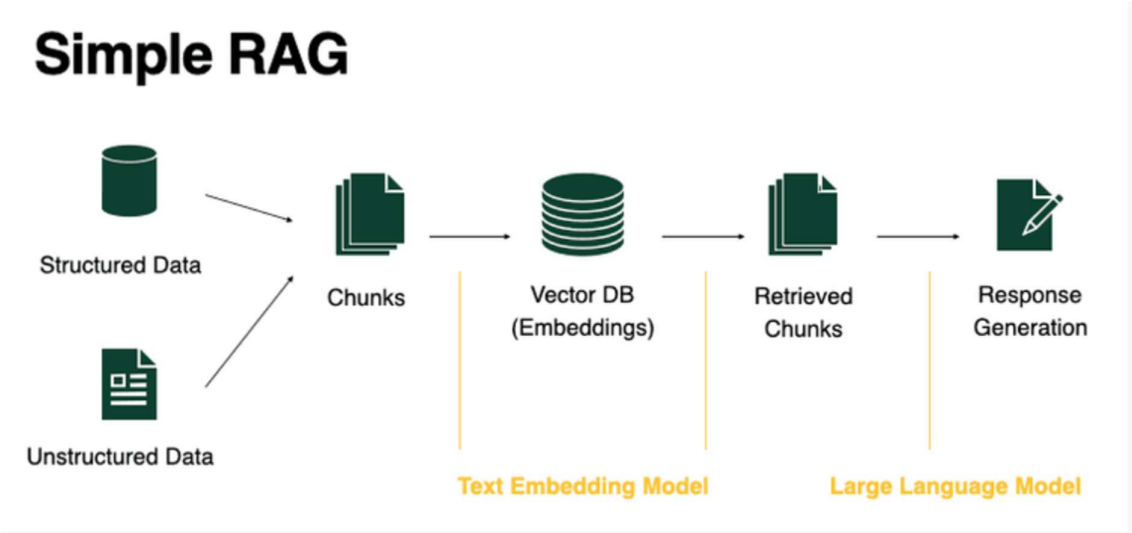
- **Components**: LangChain provides modular abstractions for the components necessary to work with language models. LangChain also has collections of implementations for all these abstractions. The components are designed to be easy to use, regardless of whether you are using the rest of the LangChain framework or not.

- **Use-Case Specific Chains**: Chains can be thought of as assembling these components in particular ways in order to best accomplish a particular use case. These are intended to be a higher level interface through which people can easily get started with a specific use case. These chains are also designed to be customizable.

The LangChain framework revolves around the following building blocks:

- Model I/O: Interface with language models (LLMs & Chat Models, Prompts, Output Parsers).
- Retrieval: Interface with application-specific data (Document loaders, Document transformers, Text embedding models, Vector stores, Retrievers).
- Chains: Construct sequences/chains of LLM calls.
- Memory: Persist application state between runs of a chain.
- Agents: Let chains choose which tools to use given high-level directives.
- Callbacks: Log and stream intermediate steps of any chain.

RAG explained with below flowchart



Source:- https://www.bentoml.com/blog/building-rag-with-open-source-and-custom-ai-models

System Design

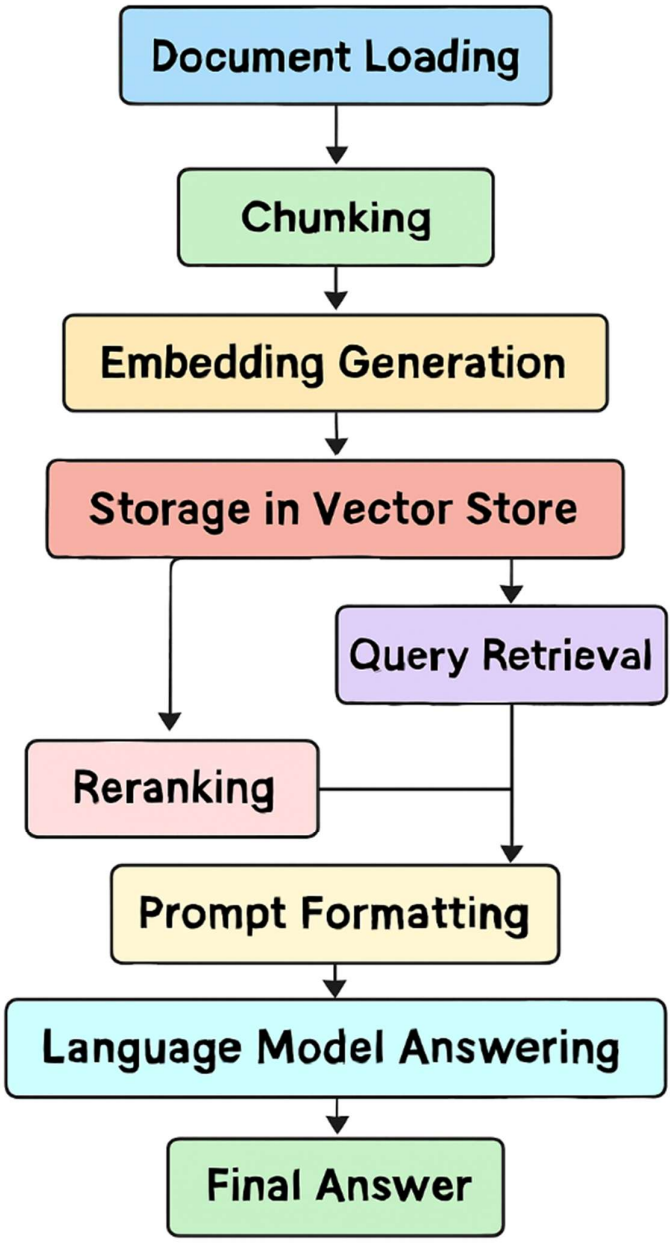
The RAG-based semantic retrieval system is designed to answer questions from PDF documents using LangChain. The system is composed of multiple modular components that work together in a pipeline to process, retrieve, and generate responses from document content.

1. PDF Reading & Processing
   PyPDFDirectoryLoader from LangChain's document loaders module is used to load\read recursively all PDF files from a specified directory. Each document is loaded with metadata (like file path and page number) which is essential for traceability and citation.
2. Document Chunking
   After loading, documents are split into smaller units (chunks) using RecursiveCharacterTextSplitter. This splitter breaks down the text intelligently by attempting to preserve semantic boundaries. It tries to split on paragraph (\n\n), line (\n), word ( ), or character ("") levels—only when absolutely necessary. This avoids breaking up clauses or definitions mid-sentence, which improves embedding quality and search recall.

3. Embedding Generation
   We use OpenAIEmbeddings from LangChain to convert each chunk of text into a dense vector representation. These embeddings encode semantic meaning and allow similarity comparison between a query and document chunk.
4. Storing Embeddings in ChromaDB
   The generated embeddings are stored in **ChromaDB**, a lightweight, local vector database integrated with LangChain. To boost performance and avoid redundant computation, embeddings are wrapped using CacheBackedEmbeddings, allowing you to cache embeddings on disk.
5. Re-Ranking with Cross-Encoder
   To further improve relevance a **re-ranker** after retrieval is applied. HuggingFaceCrossEncoder is used with the model **BAAI/bge-reranker-base**. This model evaluates each (query, passage) pair and reorders the retrieved chunks based on semantic relevance scores.It ensures that only the most contextually appropriate ones are passed to the language model.
6. RAG Chain with LangChainHub Prompt
   Connects everything using a **LangChain RAG Chain**, where:

6.1 The top-ranked context chunks are formatted with a **RAG-specific prompt**.
6.2 Pull the prompt from LangChainHub using the template: rlm/rag-prompt.
6.3 The formatted prompt and query are passed to an LLM to generate a grounded answer.

Flow diagram explains above steps:



Analysis and Interpretation of RAGAS evaluation

Evaluation is based on 4 queries and responses received by RAG system

Queries are mentioned in evaluation_inputs object which is further passed to EvaluationDataset for evaluation.

Query	Context Precision	Context Recall	Answer Relevancy
0	1.000000	1.0	0.983117
1	1.000000	1.0	0.965668
2	1.000000	1.0	0.953457
3	0.833333	1.0	0.990561

RAGAS Evaluation Quality

- **Overall**: Your retriever is doing very well — always pulling the right context (recall = 1.0) and mostly precise (precision ≥ 0.83).
- **Generation**: Answers are almost always highly relevant (>0.95), meaning the LLM is respecting the context instead of hallucinating.
- **Where to Improve**: That single 0.8333 precision shows that your retriever pulled some extra irrelevant text along with the useful chunks — not a big problem, but you could fine-tune chunk size or apply a re-ranker to prune noise.

Challenges Faced

1. Handling Long Documents
2. Choosing the Right Chunking and Embedding Strategy
3. Semantic Search Precision
4. Cross-Encoder Model Overhead
5. Prompt Sensitivity
6. Caching Strategy
7. Evaluation

Lessons Learned

1. RAG Systems Are Modular but Sensitive - changes in chunking or prompts can significantly affect output quality

2. Search and Reranking needs to be effective.
3. Prompt Engineering Is Crucial
4. Caching Matters for Efficiency - to reduce latency and cost for repeated queries.
5. Structured Design Enables Debuggability and code reusability.

Data Source

PDF files used for this RAG implementation can be downloaded and used in Google colab notebook from github link(https://github.com/Saurabh4Dev/SemanticSpotter/tree/main/data).

Project Notebook

Code (ipynb) file is submitted in zip format along with documentation and also available in github repository.

Download the insurance policy PDF files from data source to a relevant location, reference PDFs in notebook and run the code.

Pre-requisites

You should have an Openai API key and Hugging face API token with you to work along with this notebook.

Further Improvements

Few more experiments that can improve Evaluation results:

1. **Improve Retriever -** Use of hybrid retrieval (BM25 + vector).
2. **Chunking Strategy -** Adjust chunk size & overlap. Sometimes small chunks lose context, large chunks dilute relevance.
3. **LLM Guardrails -** LLM Guardrails are constraints, rules, or validation mechanisms for Large Language Model (LLM) to control its behaviour and prevent undesirable outputs like Hallucinations, Uncited claims, Off-topic responses, Unsafe / disallowed content so citation enforcement to reduce “0 relevancy hallucinations” needs to be added.