**Semantic Spotter - Project Documentation**

**Project goals**

In this project we are given a set of insurance policy documents and 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. Policy documents are 7 pdfs with pages ranging from 15 to 44. It is very hard for any individual to go over all those documents and be able to remember the details to not just pick up the right section from the right document but also be able to compare information across the different documents. The generative search system provides a conversational interface to get the right information accurately and quickly. LlamaIndex has been used due to its powerful query engine, fast data processing using data loaders and directory readers as well as easier and faster implementation using fewer lines of code.

**Data Sources**

7 pdf documents are available: The pdf files range from 15 to 44 pages. The policy documents in pdf format are the source of truth. This policy documents need to be processed, cleaned and chunked for embeddings.

Vector Store Index: is used to store documents index and metadata. This helps is quick semantic searches. The index is in-memory for faster response time and it works for our use case as the dataset is not too huge. For use cases where persistence is required, LlamaIndex provides capability to store the data on disk or Vector DB like ChromaDB

**Design choices**

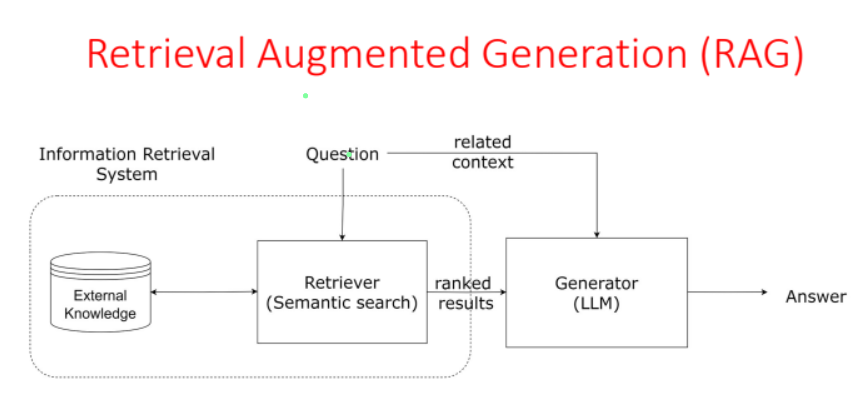
**Retrieval-Augmented Generation** (RAG) is a method that improves the responses of a language model by using information from a knowledge base. It's like giving the model a library of information to reference when it's generating a response. This makes the model's responses more accurate and relevant.

RAG combines two types of models:

- *retrieval models*, which pull data from a knowledge base, and

- *generative models*, which create the responses.

This combination makes RAG more powerful than a model that only generates responses. It can answer difficult questions and provide more informative responses.



Instead of building all the layers from scratch, we will use industry standard frameworks to work with RAG and LLM. Let us compare LlamaIndex vs LangChain:

**Feature LangChain LlamaIndex**

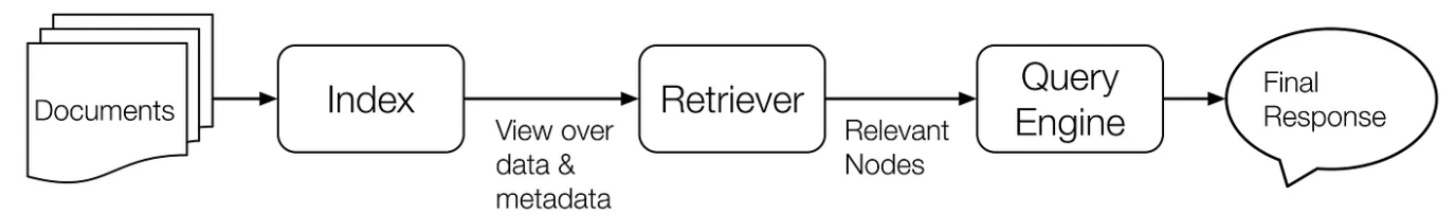
Primary Focus Workflow orchestration with LLMs Data retrieval and indexing

Strengths Agents, chains, memory integration Custom indexing, efficient retrieval

Best Use Cases Chatbots, tool use, multi-step tasks Querying external knowledge sources

Our use case is more reliant on external data sources [Policy PDFs provided]. We will need custom indexing and efficient retrieval from those external sources. So LlamaIndex is more suitable for the purpose. LangChain is better suited for complex workflows and agents. This is not a problem we need to solve in the project as the workflow is fairly basic.

**Components of LlamaIndex**:



Documents: These are the "books" in your library.

2. Index: It's the "library" of your data - Stores your data.

3. Retriever: It's the "librarian" that finds relevant data - Finds data.

4. Response Synthesizer: It's the "storyteller" that creates a response - Makes responses.

5. QueryEngine: It's the "director" that makes everything work together - Coordinates everything.

PDF parsing:

We used SimpleDirectoryReader from LlamaIndex to load all the PDFs provided. All we had to do was to provide the path to the directory and we got all the 7 pdf documents loaded. We instead get 217 documents as the reader considers each page as a single document.

Chunking:

A **Node** represents a "chunk" of a source Document, whether that is a text chunk, an image, or other. Similar to Documents, they contain metadata and relationship information with other nodes.

We choose SentenceSplitter to "parse" source Documents into Nodes through our `NodeParser` classes. This splits the documents into logical semantic boundaries.

Index:

An `Index` is a data structure that allows us to quickly retrieve relevant context for a user query. For LlamaIndex, it's the core foundation for RAG use-cases.

Under the hood, Indexes store data in Node objects, and expose a Retriever interface that supports additional configuration and automation. We use get\_nodes\_from\_documents() method from the SentenceSplitter instance and get the indexes from those Nodes using VectorStoreIndex.

Query Engine:

Query engine is a generic interface that allows you to ask question over your data.

A query engine takes in a natural language query, and returns a rich response. It is most often (but not always) built on one or many indexes via retrievers. You can compose multiple query engines to achieve more advanced capability.

A Query Engine wraps a Retriever and a ResponseSynthesizer into a pipeline. It uses the query string to fetch nodes and then sends them to the Large Language Model (LLM) to generate a response.

**Challenges**

Indexes are static unless you implement additional mechanisms for incremental updates, which may not work efficiently with dynamic or fast-changing data sources.

The effectiveness of LlamaIndex depends heavily on the chunking strategy used. Poorly chosen chunk sizes can lead to information loss or inefficiencies during retrieval.

Larger chunks provide more context but may exceed the LLM's token limits, while smaller chunks might lose semantic coherence.

Integrating LlamaIndex with existing workflows or other LLM libraries (e.g., LangChain) can require careful design and debugging.

Without explicit configuration, indexes are stored only in memory, leading to potential loss of data if not saved persistently.

Persisting large indexes (e.g., in JSON or databases) can lead to high disk usage, and reloading large indexes can be time-consuming.