Semantic Spotter Project

Problem Statement

Build a project in the insurance domain. 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. Using **langchain** to build the generative search application.

Approach

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 Gemini, Cohere, and Hugging Face.

LangChain's versatility and flexibility enable seamless integration with various data sources, making it a comprehensive solution for creating advanced language model-powered applications.

LangChain's open-source framework is available to build applications in Python or JavaScript/TypeScript. Its core design principle is composition and modularity. By combining modules and components, one can quickly build complex LLM-based applications. 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:

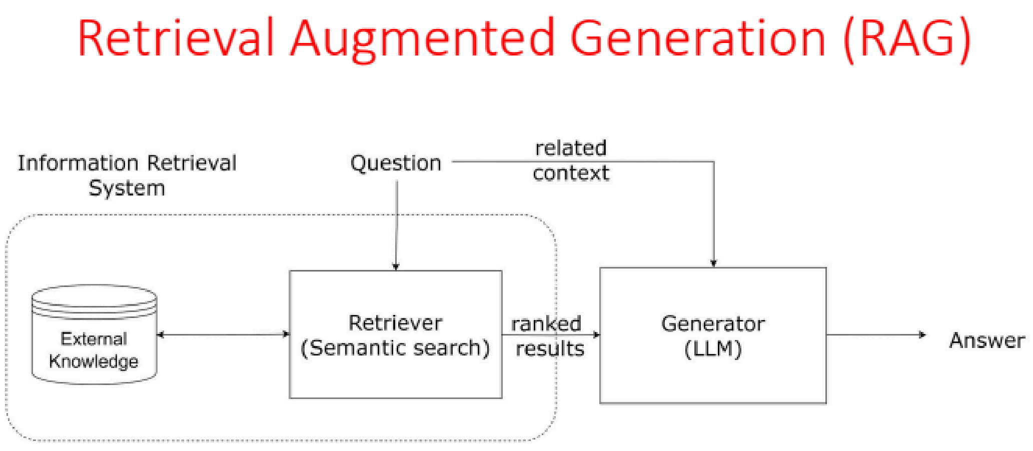
- **Components**: LangChain provides modular abstractions for the components necessary to work with language models. LangChain also has collections of implementations for all these abstractions. The components are designed to be easy to use, regardless of whether you are using the rest of the LangChain 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

RAG


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

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.

LangChain

LangChain is ideal for building RAG systems because it provides a modular, flexible framework that separates data ingestion, embeddings, vector storage, retrieval, and LLM generation. It supports a wide range of embedding models and vector databases, offers advanced retrieval methods like MMR and compression retrievers, and simplifies orchestration through chains and agents. With strong prompt management, easy customization, and production-ready features such as caching, tracing, and evaluation, LangChain makes it simple to design, optimize, and deploy robust RAG pipelines.

- **RAG Agent** — performs semantic search through a tool; flexible and general-purpose.
- **Lightweight RAG Chain** — uses a single LLM call per query; fast and effective for simple questions.

Key Concepts

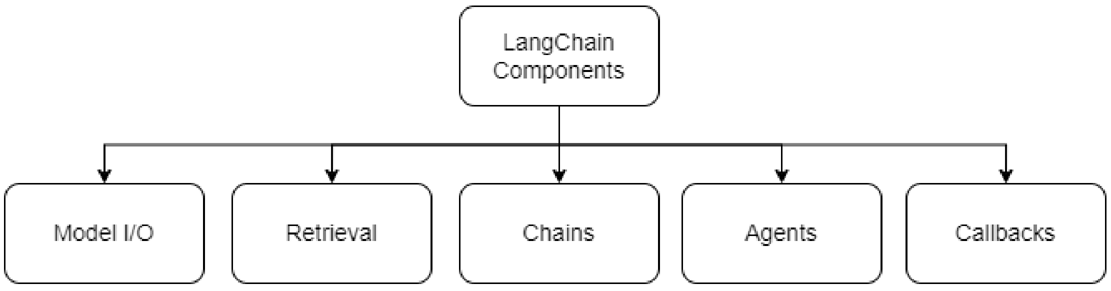
- **Indexing:** Ingest and embed your data so it can be efficiently searched.
- **Retrieval + Generation:** At query time, retrieve relevant documents and pass them to the LLM to generate accurate, grounded answers.

After indexing, an agent orchestrates the retrieval and generation steps to complete the RAG workflow.

Why Use **LangChain** for Your RAG System?

- **Model I/O** — Clean interfaces for LLMs, prompts, and structured outputs
- **Retrieval** — Document loaders, transformers, embeddings, vector stores, and retrievers
- **Chains** — Easy creation of multi-step LLM workflows
- **Memory** — Persistent context for smarter, personalized responses
- **Agents** — Enable dynamic tool and action selection
- **Callbacks** — Logging, streaming, and full visibility into chain execution

LangChain is a complete, modular, production-ready framework for powerful RAG systems.



Model I/O

LangChain's **Model I/O** module makes it easy to interface with LLMs and generate high-quality responses. It includes:

- **Language Models:** Unified interfaces for calling different LLMs
- **Prompts:** Tools to template, manage, and dynamically generate prompts
- **Output Parsers:** Utilities to structure and extract meaningful data from model outputs

This component defines the core flow of how LangChain handles inputs and outputs when interacting with language models.

1. Data Connections and Retrieval

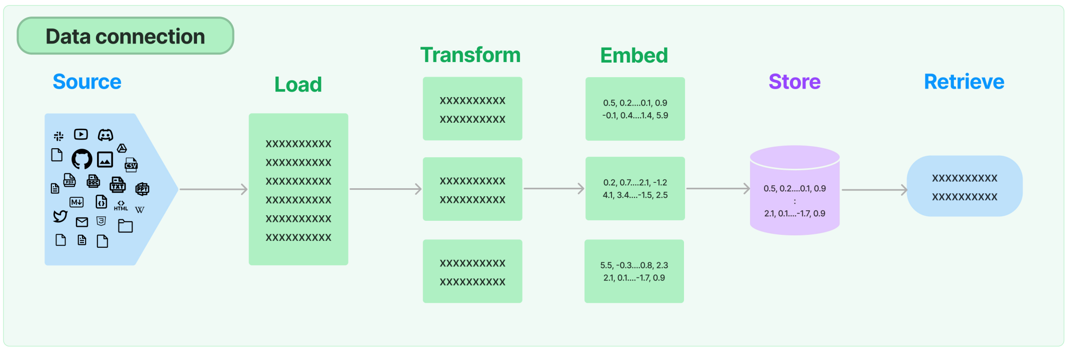
In addition to making API calls easier, LangChain also provides various methods to work with external documents efficiently.

Many LLM applications require user-specific data that is not part of the model's training set. The primary way of accomplishing this is through Retrieval Augmented Generation (RAG). In this process, external data is retrieved and then passed to the LLM when doing the generation step.

LangChain provides all the building blocks for RAG applications - from simple to complex. This section of the documentation covers everything related to the retrieval step - e.g. the fetching of the data. Although this sounds simple, it can be subtly complex. This encompasses several key modules.

The following methods provided by LangChain help process documents efficiently:

- Document Loaders
- Text Splitters
- Vector Stores
- Retrievers



Document Transformers / Text Splitters

Often times your document is too long (like a book) for your LLM. You need to split it up into chunks. Text splitters help with this.

There are many ways you could split your text into chunks, experiment with different ones to see which is best for you.

LangChain offers different text splitters for splitting the data such as:

- Split by Character
- Recursive Splitter
- Token Splitter

Split by Character - This is the simplest method. This splits based on characters (by default "\n\n") and measure chunk length by number of characters.

- How the text is split: by single character.
- How the chunk size is measured: by number of characters.

Recursive Text Splitter - This text splitter is the recommended one for generic text. It is parameterized by a list of characters. It tries to split on them in order until the chunks are small enough. The default list is ["\n\n", "\n", " ", ""]. This has the effect of trying to keep all paragraphs (and then sentences, and then words) together as long as possible, as those would generically seem to be the strongest semantically related pieces of text.

- How the text is split: by list of characters.
- How the chunk size is measured: by number of characters.

Split by tokens - Language models have a token limit. You should not exceed the token limit. When you split your text into chunks it is therefore a good idea to count the number of tokens. There are many tokenizers. When you count tokens in your text you should use the same tokenizer as used in the language model.

Text Embedding Models

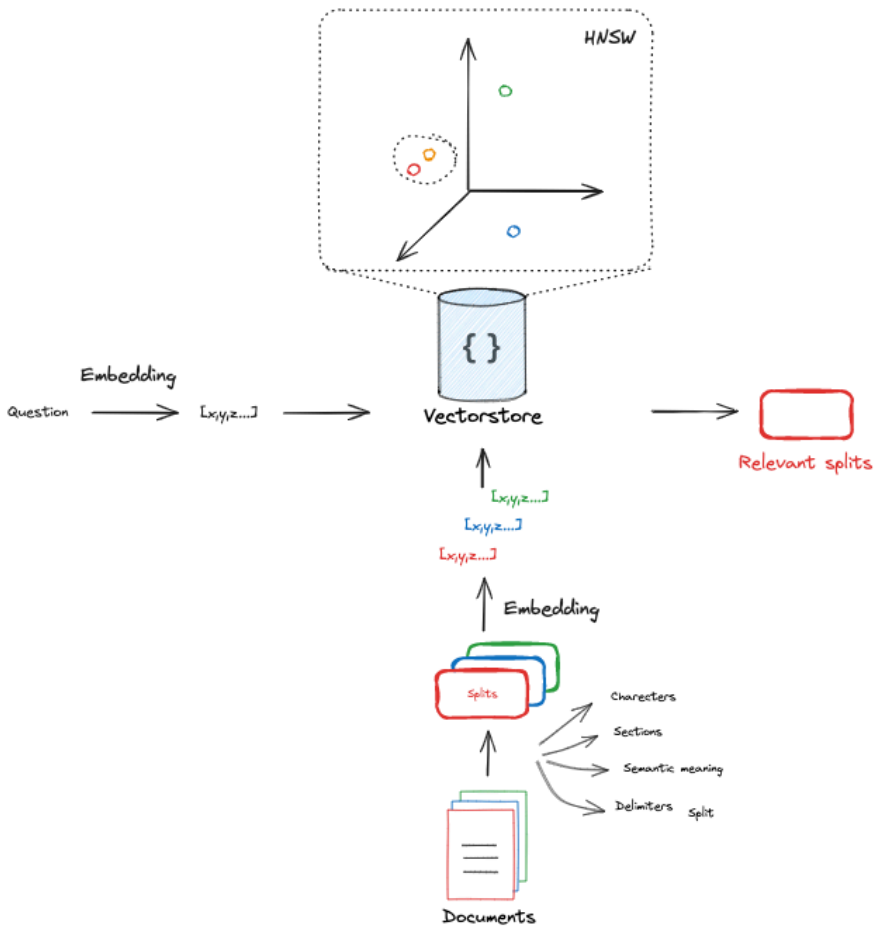
The Embeddings class is a class designed for interfacing with text embedding models. LangChain provides support for most of the embedding model providers (Gemini, huggingface) including sentence transformers library from Hugging Face.

Embeddings create a vector representation of a piece of text and supports all the operations such as similarity search, text comparison, sentiment analysis etc.

The base Embeddings class in LangChain provides two methods: one for embedding documents and one for embedding a query.

The first method takes as input multiple texts, while the second method returns the embedding representation for a single text.

Vector Stores

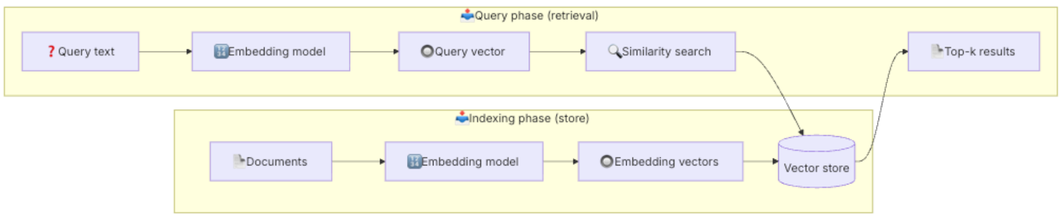


Caching

Embeddings can be stored or temporarily cached to avoid needing to recompute them. Caching embeddings can be done using a **CacheBackedEmbeddings**. This wrapper stores embeddings in a key-value store, where the text is hashed and the hash is used as the key in the cache. The main supported way to initialize a CacheBackedEmbeddings is from_bytes_store. It takes the following parameters: underlying_embedder: The embedder to use for embedding. document_embedding_cache: Any ByteStore for caching document embeddings. batch_size: (optional, defaults to None) The number of documents to embed between store updates. namespace: (optional, defaults to "") The namespace to use for the document cache. Helps avoid collisions (e.g., set it to the embedding model name). query_embedding_cache: (optional, defaults to None) A ByteStore for caching query embeddings, or True to reuse the same store as document_embedding_cache

Vector stores

A vector store stores embedded data and performs similarity search



LangChain also support all major vector stores and databases such as FAISS, ElasticSearch, LanceDB, Milvus, Pinecone etc. Refer to the API documentation for the complete list.

Retrievers

Retrievers provide Easy way to combine documents with language models.

A retriever is an interface that returns documents given an unstructured query. It is more general than a vector store. A retriever does not need to be able to store documents, only to return (or retrieve) them. Retriever stores data for it to be queried by a language model. It provides an interface that will return documents based on an unstructured query. Vector stores can be used as the backbone of a retriever, but there are other types of retrievers as well.

There are many different types of retrievers, the most widely supported is the VectoreStoreRetriever.

The official documentation and API reference contains a list of retriever integrations supported by LangChan.

Set Up Contextual Compression and Reranking

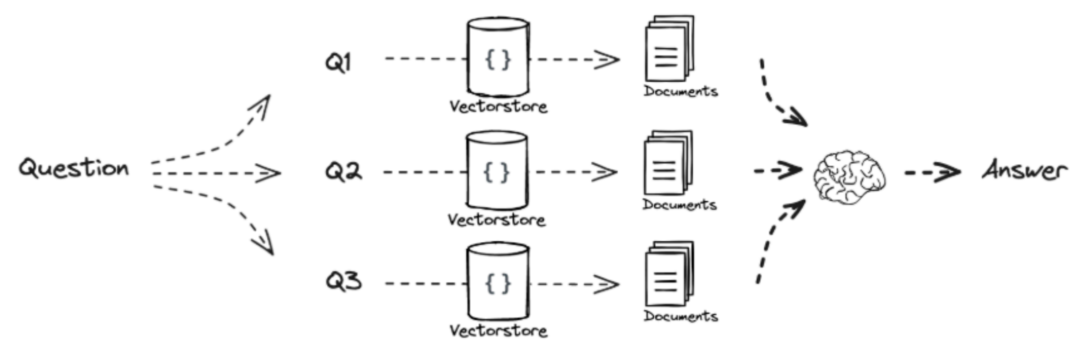
Initialize a language model with Cohere, set the reranker with CohereRerank, and combine it with the base retriever in a ContextualCompressionRetriever. This setup compresses and reranks the retrieval results, refining the output based on contextual relevance.

After adding the re-ranker, the response of your RAG system will become more refined, which will not only improve the user experience but also reduce the number of tokens used.

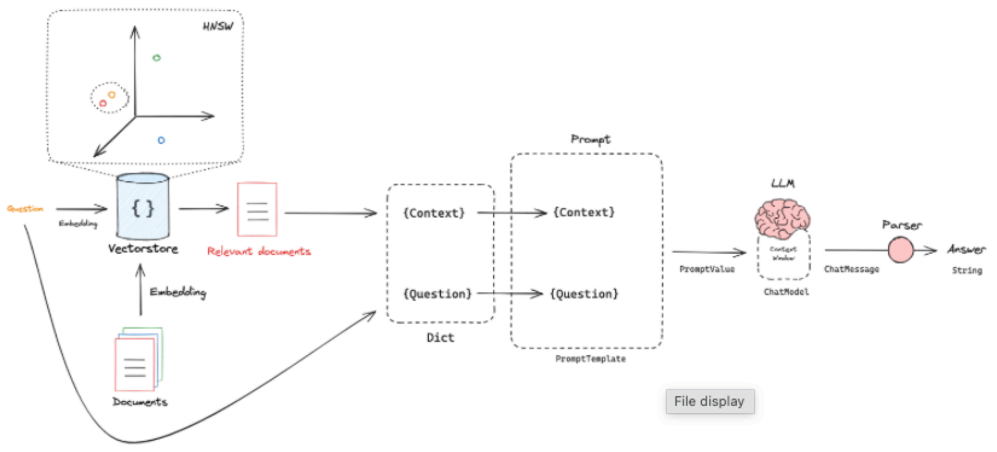
MultiQueryRetriever

Part 5: Multi Query

Flow:



Chains



File display

Using an LLM in isolation is fine for simple applications, but more complex applications require chaining LLMs - either with each other or with other components.

LangChain provides Chains that can be used to combine multiple components together to create a single, coherent application.

For example, we can create a chain that takes user input, formats it with a PromptTemplate, and then passes the formatted response to an LLM. We can build more complex chains by combining multiple chains together, or by combining chains with other components.

The fundamental unit of Chains is a LLMChain object which takes an input and provides an output.

Let's now create a simple LLMChain that takes an input, formats it, and passes it to an LLM for processing. The basic components are PromptTemplate, input queries, an LLM, and optional output parsers.

RAG Agents

One formulation of a RAG application is as a simple agent with a tool that retrieves information. We can assemble a minimal RAG agent by implementing a tool that wraps our vector store:

A common approach is a two-step chain, in which we always run a search (potentially using the raw user query) and incorporate the result as context for a single LLM query. This results in a single inference call per query, buying reduced latency at the expense of flexibility. In this approach we no longer call the model in a loop, but instead make a single pass. We can implement this chain by removing tools from the agent and instead incorporating the retrieval step into a custom prompt

System Layers

This pipeline demonstrates building a **Retrieval-Augmented Generation (RAG)** system using LangChain.

Key Highlights

Multi-Retriever Support
Combine multiple retrievers to query different data sources for richer context.

2 Step RAG Agent
Step 1: Retrieve relevant documents for the query.
Step 2: Inject retrieved context into the LLM prompt for answer generation.

Short-Term Memory
Use `InMemorySaver` to remember previous interactions in a session.

PDF Ingestion & Processing
Use `PyPDFDirectoryLoader` to read PDFs from a directory.

Document Chunking
`RecursiveCharacterTextSplitter` splits text into semantically meaningful chunks.

Embedding Generation
`HiggingFaceEmbeddings` (or other providers) convert text chunks into vector representations.

Storing Embeddings
Persist embeddings in **ChromaDB**, optionally with `CacheBackedEmbeddings`.

Retrievers
Retrieve documents from vector stores or other backends based on user queries.

Re-Ranking
Improve relevance using a cross-encoder (e.g., `BAAI/bge-reranker-base`).

Chains & Prompt Templates
Combine tools, prompts, and LLM calls using LangChain Chains.

Challenges

Handling large PDFs and splitting into semantically coherent chunks. Ensuring relevance during retrieval and re-ranking. Integrating multi-retriever setup with 2-step RAG chain. Managing session-based memory and avoiding conflicts.

Lessons Learned

Proper chunking and embedding are crucial for relevant results. Cross-encoder reranking improves accuracy significantly. Short-term memory enables smooth multi-turn interactions. A modular design (data → embeddings → retriever → agent) simplifies maintenance and scaling.

Final Thoughts

As we delve into the realm of advanced language processing, the evolution of RAG systems stands out as a testament to innovation and progress. The rapid development of more sophisticated paradigms has not only enabled customization but also furthered the performance and utility of RAG across diverse domains. From hybrid methodologies to self-retrieval techniques, researchers are exploring a myriad of avenues to enhance the capabilities of these models.

One crucial aspect that has gained attention is the role of rerankers in enhancing RAG efficacy. By refining the two-stage retrieval process through innovative approaches and reranking, RAG systems are now able to provide much better responses. The demand for better evaluation methodologies underscores the need for continuous improvement and refinement in RAG systems.

In [ ]:

```
```