Semantic Spotter

1. Project Overview:

Goal of Semantic Spotter project is to build a robust generative search system capable of effectively and accurately answering questions from various insurance policy documents.

This system was designed to answer questions specifically related to insurance policy documents. Goal for the project was set to build a Retrieval Augmented Generation session which will be able to answer queries from users accurately and use of the frameworks (LangChain or LlamaIndex) for developing this application.

2. High Level System Design

A typical RAG application has two main components:

Indexing: a pipeline for ingesting data from a source and indexing it. This usually happens offline.

Retrieval and generation: the actual RAG chain, which takes the user query at run time and retrieves the relevant data from the index, then passes that to the model.

3. Project Stages

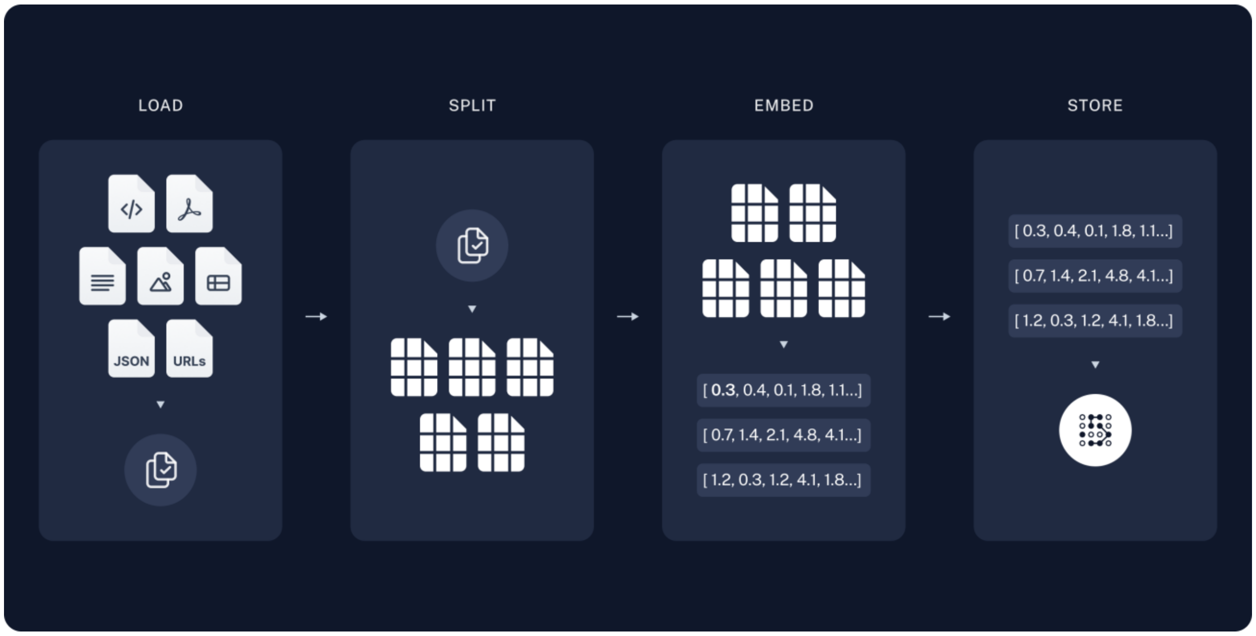
Indexing

1. **Load**: First we need to load our data. This is done with Document Loaders.
   We will be using LangChain PyPDFDirectoryLoader to read and process the PDF files from specified directory.
2. **Split**: Text splitters break large `Documents` into smaller chunks. This is useful both for indexing data and passing it into a model, as large chunks are harder to search over and won't fit in a model's finite context window.

   We will be using LangChain RecursiveCharacterTextSplitter. 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.

3. **Store**: We need somewhere to store and index our splits, so that they can be searched over later. This is often done using a VectorStore and Embeddings model.

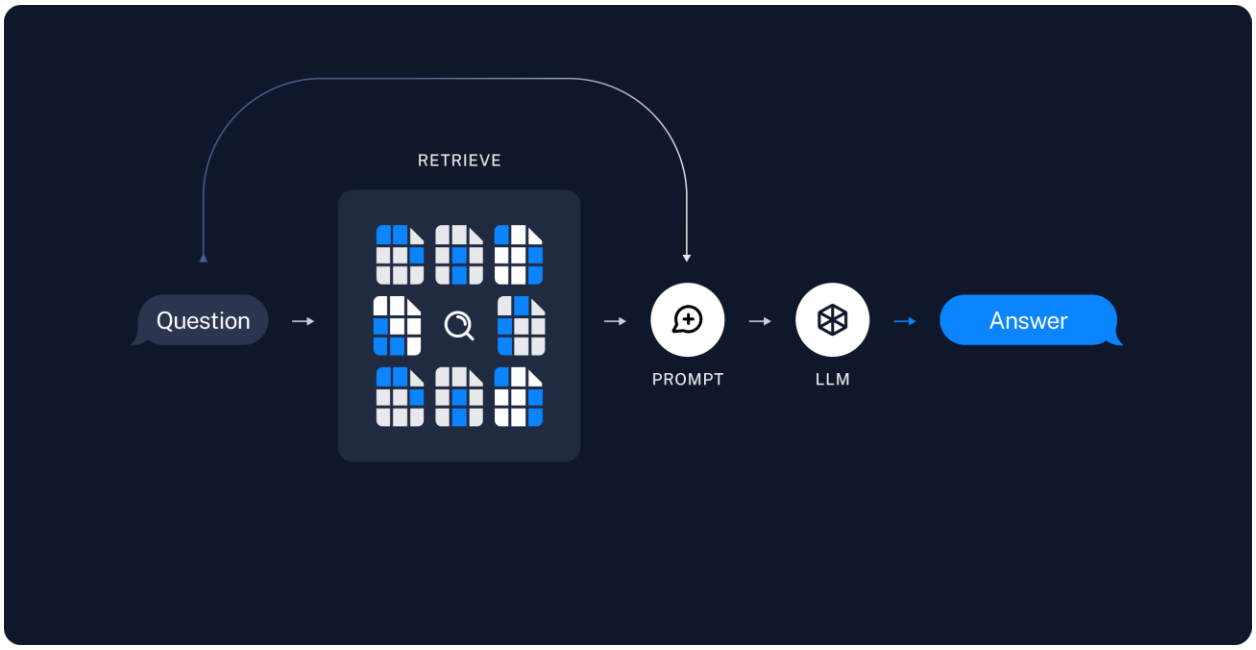


Retrieval and generation

4. **Retrieve**: 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

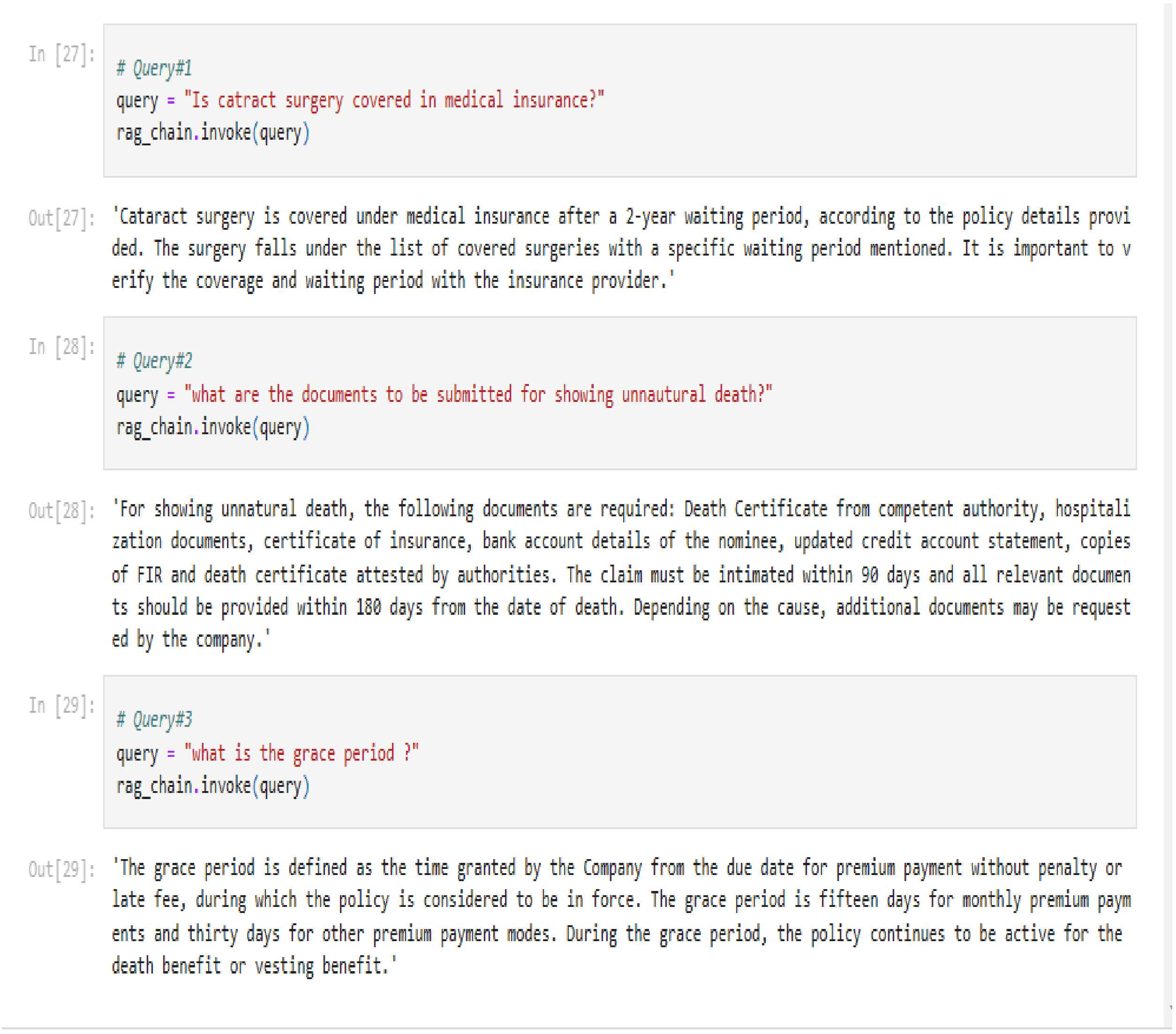
5. **Re-Ranking with a Cross Encoder**: Re-ranking the results obtained from the semantic search will sometime significantly improve the relevance of the retrieved results. This is often done by passing the query paired with each of the retrieved responses into a cross-encoder to score the relevance of the response w.r.t. the query. The above retriever is associated with HuggingFaceCrossEncoder with model BAAI/bge-reranker-bas

6. **Chains**: 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. We are using pulling prompt rlm/rag-promp from langchain hub to use in RAG chain.



Once we've indexed our data, we will use LangGraph as our orchestration framework to implement the retrieval and generation steps.

4. Screenshots



```
In [27]:  # Query#1
          query = "Is catract surgery covered in medical insurance?"
          rag_chain.invoke(query)

Out[27]:  'Cataract surgery is covered under medical insurance after a 2-year waiting period, according to the policy details provi
          ded. The surgery falls under the list of covered surgeries with a specific waiting period mentioned. It is important to v
          erify the coverage and waiting period with the insurance provider.'

In [28]:  # Query#2
          query = "what are the documents to be submitted for showing unnautural death?"
          rag_chain.invoke(query)

Out[28]:  'For showing unnatural death, the following documents are required: Death Certificate from competent authority, hospitali
          zation documents, certificate of insurance, bank account details of the nominee, updated credit account statement, copies
          of FIR and death certificate attested by authorities. The claim must be intimated within 90 days and all relevant documen
          ts should be provided within 180 days from the date of death. Depending on the cause, additional documents may be request
          ed by the company.'

In [29]:  # Query#3
          query = "what is the grace period ?"
          rag_chain.invoke(query)

Out[29]:  'The grace period is defined as the time granted by the Company from the due date for premium payment without penalty or
          late fee, during which the policy is considered to be in force. The grace period is fifteen days for monthly premium paym
          ents and thirty days for other premium payment modes. During the grace period, the policy continues to be active for the
          death benefit or vesting benefit.'
```