Problem Statement

The goal of the project will be to build a robust generative search system capable of effectively and accurately answering questions from various insurance policy documents. In short, we are building a Generative Search System for Insurance Policy Documents using LangChain.

Why is LangChain the ideal framework for this project?

---

1. It enables **retrieval-augmented generation (RAG)**, ensuring more accurate answers based on policy documents.
2. Provides built-in **document loaders**, **embedding models**, **vector databases**, and **retrievers** for efficient search.
3. Supports **chaining multiple LLM operations**, enabling structured query handling.
4. Works well with **PDF processing**, allowing us to extract information from complex insurance documents.

System Design

We need to create an optimal architecture for our system. Here's a proposed high-level design:

- **Input**: User queries about an insurance policy.
    - **Processing**:
        - Load and process PDF insurance documents.
        - Chunk documents for efficient retrieval.
        - Generate embeddings and store them in a vector database.
        - Retrieve relevant chunks and re-rank results.
        - Use an LLM (GPT-based) to generate final answers.
- **Output**: A **reliable response** based on relevant insurance policy text.

Code Implementation

We will follow structured coding with clear documentation at each step:

- **Step 1**: Load & Process PDFs (`PyPDFDirectoryLoader` or `PyPDFLoader`)
- **Step 2**: Chunk Documents (`RecursiveCharacterTextSplitter`)
- **Step 3**: Generate Embeddings (`OpenAIEmbeddings with LangChain`), Cache them using `CacheBackedEmbeddings`
- **Step 4**: Store in Vector DB (`ChromaDB`)
- **Step 5**: Re-rank with `HuggingFaceCrossEncoder`, Implement Retrievers (`ContextualCompressionRetriever`)
- **Step 6**: Build LangChain Pipeline (`RAG Chain`)

Step 1: Loading & Processing PDFs (Extract and Process Text).

---

1. Use **LangChain's** `PyPDFLoader` to read multiple PDFs from a folder. Load the PDFs from Google Drive (I'll download and process them locally).
2. Since the PDFs are unstructured, I will extract raw text instead of structured metadata and clean it by removing extra whitespaces and line breaks for better readability.
3. Display a sample or structured preview (first 500 characters) to verify quality or to validate the extracted text.

I originally planned to use `PyPDFDirectoryLoader`, which is **designed to load all PDFs from a directory at once**. However, the current code is using `PyPDFLoader`, which **loads PDFs individually**.

Which One Should be Used?

- **PyPDFDirectoryLoader**: Loads all PDFs at once, reducing the need for manual file iteration. More efficient for bulk processing.
- **PyPDFLoader**: Loads PDFs one at a time, useful for custom per-file handling. Gives more control over individual document processing.

I recommend using `PyPDFLoader` as it offers better control over individual PDF files while ensuring proper data cleaning. The script extracts **source**, **page number**, and **page content** from each document and displays a preview. The preview will show the first three pages per PDF.

Step 2: Document Chunking (Chunking the text to make it suitable for retrieval and embedding)

Why Chunking?

Since LLMs have a token limit, we must break down long insurance policy documents into meaningful chunks for:

1. Efficient retrieval – Smaller chunks ensure relevant portions are retrieved instead of entire documents.
2. Better embeddings – Improves vector search by ensuring contextually relevant text is stored.

Explanation:

Chunking is a critical step in building a generative search system because it optimizes how information is stored and retrieved. Here's why it matters:

1. **Chunking for Retrieval**: When a user asks a question, the system needs to find **relevant portions** of the document instead of searching the entire text.
   - **Without chunking**: Searching across a large document may return **irrelevant** or **less precise** results.
   - **With chunking**: Breaking text into **smaller**, **meaningful sections** makes it easier to retrieve precise answers.
   - **Example**: Instead of retrieving an entire 30-page insurance document, we can retrieve only the **most relevant policy section**.
2. **Chunking for Embeddings**: Embeddings convert text into **numerical representation** so that similar text can be compared using vector search.
   - **Without chunking**: Long documents produce large embeddings, which can exceed **LLM token limits** and **reduce retrieval accuracy**.

    - **With chunking**: Each small chunk is embedded separately, making **search and comparison faster and more precise**.
    - **Example**: A **1000-token chunk** ensures that embeddings capture local context while remaining efficient for retrieval.

How This Helps Our Insurance Search System

- **Retrieves only the necessary policy sections** when answering a user's question.
- **Improves accuracy** by keeping embeddings focused on specific concepts (e.g., claims process, coverage exclusions).
- **Handles large documents efficiently**, ensuring the system can scale to multiple policies.

How to Implement Chunking

Use **LangChain's** `RecursiveCharacterTextSplitter`, which:

1. **Preserves semantic meaning**: Tries to **split at paragraph level** ("\n\n") at first, then sentences ("\n"), then words (" ").
2. Ensures **semantic consistency** or **ensures continuity** while maintaining a fixed chunk size: Allows fine-tuning chunk **size and overlap** for better retrieval performance. The chunk overlap prevents loss of context between chunks.

Here's why I didn't use other splitters (or other ways to split text using LangChain):

1. **Why Not `CharacterTextSplitter?`**:

    - Splits text by **a single character** (e.g., \n, " "), meaning it might break sentences or words in the middle, leading to poor chunk coherence.
    - **Example**: If splitting by \n, it may create incomplete chunks if text doesn't have enough line breaks.
2. **Why Not `TokenTextSplitter`?**
    - Splits by **number of tokens** (useful for LLM-specific tokenization like OpenAI's tiktoken).
    - This splitter **breaks text purely based on token count**, which is efficient but often disrupts contextual integrity (e.g., breaking mid-sentence).
    - It **doesn't prioritize creating logical/meaningful pieces of text** (e.g., paragraph or sentence), making retrieval less reliable.
3. **Why Not `NLTKTextSplitter`?**
    - **NLTK sentence tokenizer** is good for structured data but may break long texts unpredictably.
    - Used to split sentences, but it may **over-split** long content like insurance documents.
    - **NLTK-based splitting** can fail on PDFs with inconsistent formatting
4. **Why Not `SpacyTextSplitter`?**
    - Leverages **spaCy's NLP models** for sentence splitting, useful for heavy pro-NLP tasks.
    - **spaCy-based splitting** adds unnecessary processing overhead.

Why Use `RecursiveCharacterTextSplitter` Instead of Other Splitters?

Chose `RecursiveCharacterTextSplitter` because it **prioritizes preserving the document's semantic structure** while breaking it into chunks.

- It **recursively tries multiple split levels** (paragraph → sentence → word), ensuring the best **logical/meaningful** chunking possible.

- It **minimizes broken/lost context**, leading to **better retrieval and embeddings** for our generative search system.
- The **chunk overlap** (e.g., 200 tokens) ensures continuity, preventing loss of crucial details.

Code Implementation

Document chunking using `RecursiveCharacterTextSplitter` with:

1. **Chunk Size Preference**: Default
   - 1000 tokens per chunk
2. **Chunk Overlap**: 200 tokens
   - Helps maintain continuity across chunks for better retrieval
   - Prevents information loss at chunk boundaries
3. **Preview**: First 3 chunks from each PDF

Step 3: Generating Embeddings using `OpenAIEmbeddings` and Caching them

Generate **vector embeddings** for each text chunk using **OpenAI's Embeddings API**. These embeddings allow us to:

1. Compare text chunks based on meaning rather than exact words.
2. Perform semantic search to retrieve the most relevant policy sections.
3. Store embeddings in a vector database.

---

keyboard_arrow_down

Code Implementation - Generate Embeddings

1. Use `OpenAIEmbeddings` from LangChain to generate embeddings.
   - **OpenAI model for embeddings**: `text-embedding-ada-002`
2. Process each chunk and store its corresponding vector representation.
3. Preview a few embedding values to verify the process.

A **state-of-the-art application** must include **caching** to optimize performance.

Why Use Caching for Embeddings?

- If an embedding already exists in the cache, we won't store it again.
- **Improves Speed**: Reusing existing embeddings means we don’t have to recompute for the same chunks. Faster lookup by avoiding unnecessary database inserts. In short, **improve retrieval speed**.
- **Ensures Consistency across stored embeddings**: Cached embeddings remain stable, ensuring uniform search results over time. Cached embeddings remain the same across retrievals.

How Caching Will Work in ChromaDB?

1. Before storing an embedding, we check if it exists in the cache.
2. If not cached, we compute, store in ChromaDB, and update the cache.

Code Implementation - Optimized Embedding Strategy with Caching

1. Uses `CacheBackedEmbeddings` to prevent redundant API calls.
2. **Hashes each chunk** to check if embeddings exist before recomputing.
3. Uses `InMemoryCache` (can be replaced with persistent storage like SQLite or ChromaDB).
4. Displays embedding **previews for validation**

Understanding Query Embeddings in Vector Search

When a user asks a question, the system must **retrieve the most relevant document chunks** from the vector database. This is done by **implicitly** embedding the query **in the same way as the document chunks**.

1. System **implicity** converts the user query into an embedding using the same `text-embedding-ada-002` model.
2. System compares the query embedding with stored document embeddings using similarity search.
3. System returns the top-matching chunks, ensuring relevant insurance policy sections are retrieved.

This allows the system to find semantically similar text, even if the query does not exactly match the wording in the document.

Step 4: Storing Embeddings in ChromaDB

Since we have **generated and cached embeddings**, we will **store them in ChromaDB** for efficient retrieval.

Why ChromaDB?

1. **High-performance vector storage**: Optimized for fast similarity search.
2. **Metadata support**: Can store source PDF name and page number/chunk hash for better document retrieval.
3. **Flexible persistence**: Can be stored in-memory (faster) or persisted (for reusability across sessions).

Code Implementation

1. **Initialize ChromaDB as our vector store** (in-memory for speed, but configurable for persistence).
2. Attach **metadata (source PDF file and chunk hash)** to each stored embedding for better retrieval context.
3. **Runs a test query** (`"Does this insurance policy cover hospitalization expenses for accidents?"`) to verify that embeddings are stored and retrievable.

Why Use `from langchain.schema import Document`?

The `Document` class in LangChain is used to structure text data along with metadata before storing it in a vector database like ChromaDB.

Reasons for using `Document` in the code are as follows:

- Metadata Storage:

    - Each document (insurance policy chunk) needs **source details** (PDF name, chunk hash).
    - The `Document` class allows us to store and retrieve this metadata efficiently.
- Structured Data Format:
    - ChromaDB **expects input in a structured form**, and `Document` ensures a standard format.
    - It prevents storing raw text without context.
- Easy Retrieval:
    - When performing a **similarity search**, ChromaDB returns `Document` objects, allowing easy metadata access (`doc.metadata["source"]`).

Step 5: Implementing Retrievers with Cross Encoding

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.

Code Implementation

1. Uses `ContextualCompressionRetriever` to retrieve documents efficiently.
2. Retrieves top k documents with an **MMR score threshold of 0.8**.
3. Integrates `CrossEncoderReranker` (`BAAI/bge-reranker-base`) to re-rank retrieved results.
4. Returns and displays relevant documents for a given query.

Step 6: Building a RAG Chain for Final Response Generation

Now that we have a **retriever system**, we will integrate it into a **Retrieval-Augmented Generation (RAG) Chain** using LangChain.

Why Use a RAG Chain?

1. **Combines retrieval with an LLM**: Ensures responses are relevant to policy documents.
2. **Enhances accuracy**: Instead of generating from scratch, the model answers based on retrieved chunks.
3. **Maintains traceability**: Each response can be linked back to its source in the policy documents.

Code Implementation

- Use `ConversationalRetrievalChain` from LangChain to combine retrieval and generation (to connect retrieval and LLM generation).
- **Pull a prebuilt RAG prompt** (`rlm/rag-prompt`) **from LangChain Hub**.
- Integrate **OpenAI GPT for response generation**.
- Ensure the model outputs a reference to the retrieved documents. Meaning, **it returns both the generated response and source documents**.
- **Integrates** `RunnablePassthrough` and `StrOutputParser` **for structured output processing**.