# Objectives

## Problem Statement

Insurance policy documents are long, complex, and full of legal terms. Searching for specific details in these documents can be frustrating and time-consuming. Regular search methods only look for exact words and often miss important context, leading to incomplete or incorrect answers.

The goal of this project is to build a smart search feature that can accurately answer questions using information from insurance policy documents. Instead of just matching keywords, it will understand the meaning of the text and provide clear, relevant answers.

## Why LangChain is an ideal framework?

LangChain help make smart search feature by combining RAG system with LLM like Google’s Gemini.

* LangChain can load, split, and embed documents for easy searching.
* LangChain uses vectore search to find the best matches, even if the words don’t exactly match the query.
* The implementation of a LLM allows the system to generate a meaningful answer, instead of just the raw text of the document.
* LangChain can handle PDFs, websites, and many other formats.

# Design

## Directory structure

|  |  |  |
| --- | --- | --- |
| ***Level 1*** | **Level 2** | Description |
| ***api\_keys/*** | **OpenAI\_API\_Key.txt** | Store OpenAI API key |
|  | **Gemini\_API\_Key.txt** | Store Gemini API key |
| ***policy\_documents/*** |  | raw PDF documents |
| ***chroma\_db/*** |  | Vector store of the document |
| ***Embedder.py*** |  | Script for document embedding |
| ***Retriever.py*** |  | Script for query processing and response generation |
| ***requirements.txt*** |  | Required dependencies |
| ***README.md*** |  | detailing how to set up & run |

## Layers and functions

|  |  |  |
| --- | --- | --- |
| ***Layer*** | **Function** | **Description** |
| *Document Processing Layer*  *(****Embedder.py****)* | load\_pdf\_directory() | Load PDFs from a directory |
| split\_documents() | Split documents into chunks |
| OpenAIEmbeddings() | Embed chunks |
| save\_to\_chroma\_db() | Store embeddings in ChromaDB |
| *Retrieval and Response Generation (****Retriever.py****)* | Chroma() | Load stored embeddings from ChromaDB |
| get\_query\_results() | Process user query |
| similarity\_search  \_with\_relevance\_scores() | Perform similarity search |
| prompt\_template.format() | Use retrieved results as context in prompt |
| generate\_answer() | Generate response using Gemini LLM |

# Implementation

* Use a cloud storage service for PDFs.
* Run the Embedder when new PDFs are uploaded.
* Wrap the Retriever in a web interface for end-users to interact with the Retriever.

# Challenges

* Challenge: incompatible packages lead to code’s error
* Solution: implement pip freeze to fix the packages version into requirements.txt file. Any environment can refer to this requirements.txt to install the correct package and version.
* Challenge: low performance of the similarity search (semantic search) on the Vector Store
* Solution: perform semantic search on various embedding model to identify which embedding model return the most relevant chunks of data.
* Challenge: high tokens cost when using OpenAI’s flagship models
* Solution: look for alternative model like Gemini Flash Thinking, which provide similar performance in this specific use case.

# Lessons Learnt

* Successfully build a RAG + AI system can significantly improve how an organization handle customer inquiries.
* Depends on the type and language of the documents, different Embedding model will have different semantic search performance. Thus choosing the suitable model will greatly improve the overall system.