## Part 1: Project Introduction

#### Project Background

The goal of the project will be to build a robust generative search system capable of effectively and accurately answering questions from a policy document.

#### Problem Statement

\*Given a policy document containing information about Life insurance, build a HelpMate AI that parses the filtered documents and provides answers to user's queries\*.

### System Design:

#### The Project will have following layers:

1. The Embedding Layer :

- The PDF document needs to be effectively processed, cleaned, and chunked for the embeddings.

- Will experiment and choose appropriate chunking strategy and embedding model.

-

2. The Search Layer

- Sesign at least 3 queries against which to test the system.

- Search ChromaDB vector database against each of these queries.

- Implementing a cache mechanism

- Implement the re-ranking block

3. The Generation Layer

- Design the final prompt (exhaustive in its instructions, provide some few-shot examples etc)

## The Embedding Layer

### Read, Process, and Chunk the PDF Files

We will be using [pdfplumber](https://https://pypi.org/project/pdfplumber/) to read and process the PDF files.

`pdfplumber` allows for better parsing of the PDF file as it can read various elements of the PDF apart from the plain text, such as, tables, images, etc. It also offers wide functionaties and visual debugging features to help with advanced preprocessing as well.

- Extracting text from multiple PDFs

Let's now try and read multiple documents, extract text from them using appropriate preprocessing, and store them in a dataframe

This concludes the chunking aspect also, as we can see that mostly the pages contain few hundred words, maximum going around 500. So, we don't need to chunk the documents further; we can perform the embeddings on individual pages. This strategy makes sense for 2 reasons:

1. The way insurance documents are generally structured, you will not have a lot of extraneous information in a page, and all the text pieces in that page will likely be interrelated.

2. We want to have larger chunk sizes to be able to pass appropriate context to the LLM during the generation layer.

## Generate and Store Embeddings using OpenAI and ChromaDB

In this section, we will embed the pages in the dataframe through OpenAI's `text-embedding-ada-002` model, and store them in a ChromaDB collection.

## Semantic Search with Cache

In this section, we will perform a semantic search of a query in the collections embeddings to get several top semantically similar results.

- query 1 - what is the period during which the policy is active? and what are the conditions during it?

- query 2 - what is th coverage scope of the insuracne?

- query 3 - what are the things to take care of to make sure insuarnce can be claimed without issues?

## Re-Ranking with a Cross Encoder

Re-ranking the results obtained from your semantic search can 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.

### Retrieval Augmented Generation

Now that we have the final top search results, we can pass it to an GPT 3.5 along with the user query and a well-engineered prompt, to generate a direct answer to the query along with citations, rather than returning whole pages/chunks.

#### Challenges Faced

- Choosing the appropriate Embedding model and chunking strategy to have relavant data in proper chunks

- Choosing the right reranking strategy so that it gives proper chunks as results.