# **HelpMate AI Project:**

## Project Goal:

The goal of the project is to develop a generative search system that can efficiently and accurately answer questions derived for a insurance policy document. The system will retrieve relevant information from a vector database and, with the help of an LLM (currently using OpenAI's chat completion API), generate a natural language response that is clear and easy to understand.

**Data Source:** Insurance Policy document is provided by **upGrad**

**System Architecture**

A diagram of a process

Description automatically generated

System Design has below layers

1. **Embedding Layer**. This layer reads PDF and apply page level chunking strategy. Once PDF divided into pages then embeddings are generated.

**Why page level chunking strategy ?**

The page-level chunking strategy was adopted because the insurance document consists of only 64 pages, each containing a relatively small amount of data without tables or diagrams. By keeping the complete context of each page, this approach improves performance by reducing the need for multiple joins when sending data to the chat completion API. Additionally, it ensures that the API has full context, leading to more efficient and accurate results.

1. **Search Layer**: Once embeddings are generated for the PDF page level content.

* Embeddings are stored in the collection
* An empty collection is created for Cache Implementation

When a query is searched. It is first searched in the cache if cache doesn’t retrieved any data then main collection is searched in which embeddings for all the pages are stored and same query and its results are stored in cache as well so that cache can retrieved results in case similar query is searched.

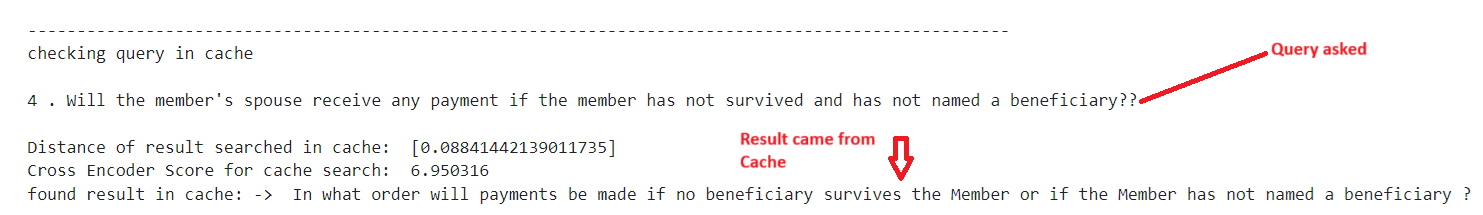
**Innovation (**Cross encoder additional Safeguard while searching Cache**)**:

When a query is searched in the cache, ChromaDB uses cosine distances to compare the current query with previously searched queries. If the cosine distance is less than the threshold (i.e., 0.2), system indicates that the two queries are similar.

As an additional safeguard, I have implemented a cross-encoder to further verify the accuracy of ChromaDB's search results. If the cross-encoder's prediction score is less than equal to zero, the results of the previously cached query are not returned, and the current query is treated as a new one, meaning it is not found in the cache.

**Queries and Cache Implementation:** I have prepared four queries: three of them are new, and one is related to one of the other three. **In below table When 4th Query is searched result should come out of Cache. Refer screenshot**

|  |  |  |
| --- | --- | --- |
| **Sno** | **Query** | **Comment** |
| **1** | In what order will payments be made if no beneficiary survives the Member or if the Member has not named a beneficiary ? | **New** |
| **2** | What is the time frame for a Member to request changes in benefits due to a change in family status ? | **New** |
| **3** | What is the amount needs to be paid to purchase the policy ? | **New** |
| **4** | Will the member's spouse receive any payment if the member has not survived and has not named a beneficiary ? | **Can be answered via Query 1 results** |

**Below screenshot shows query result returned from cache**

**Other Queries returned from Main collection**

**A screenshot of a computer code

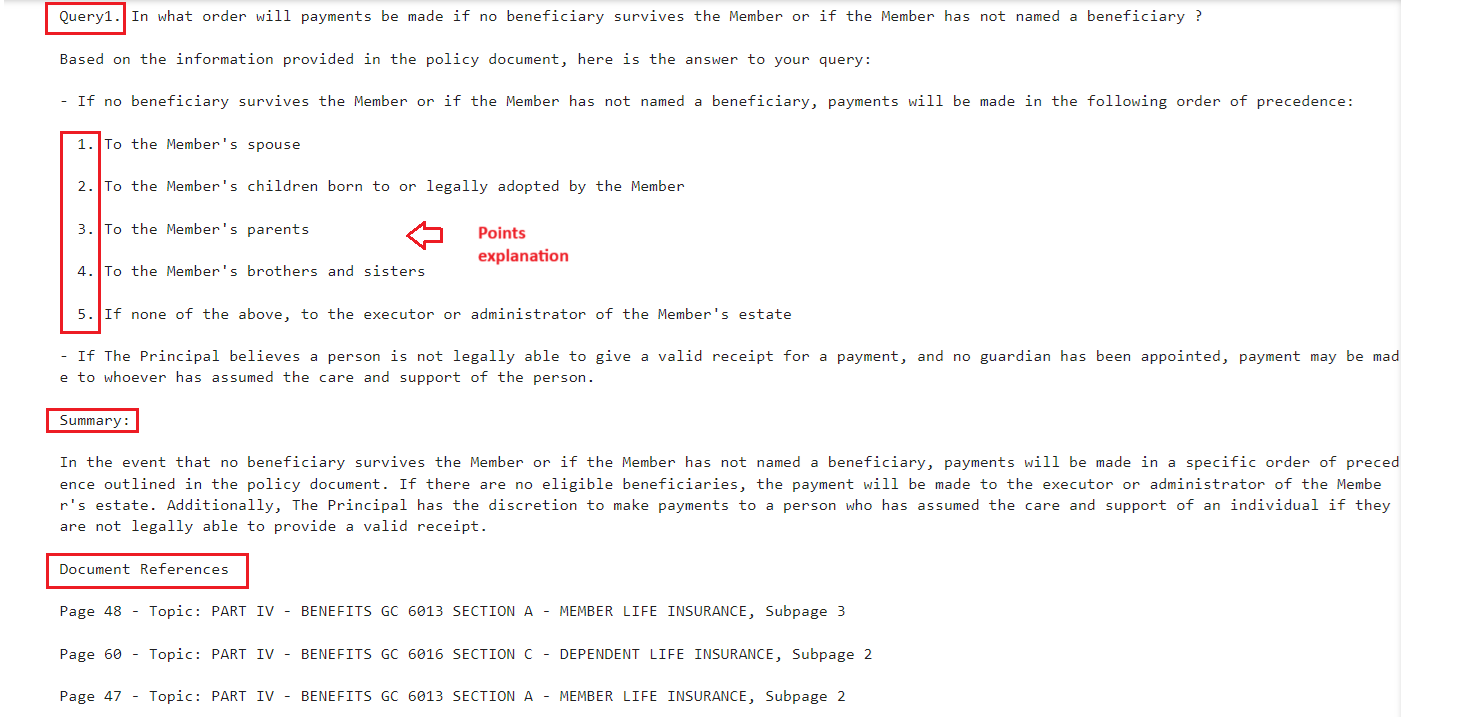
Description automatically generated**

1. **Generative Layer:** This finally generates the answer based on the query and the chromadb searched text.

**Generated Answer structure:**

1. Pointwise explanation of result.
2. Provide summary at the end.
3. References from PDF

Generated Answer structure



**Innovation/ Challenge:**

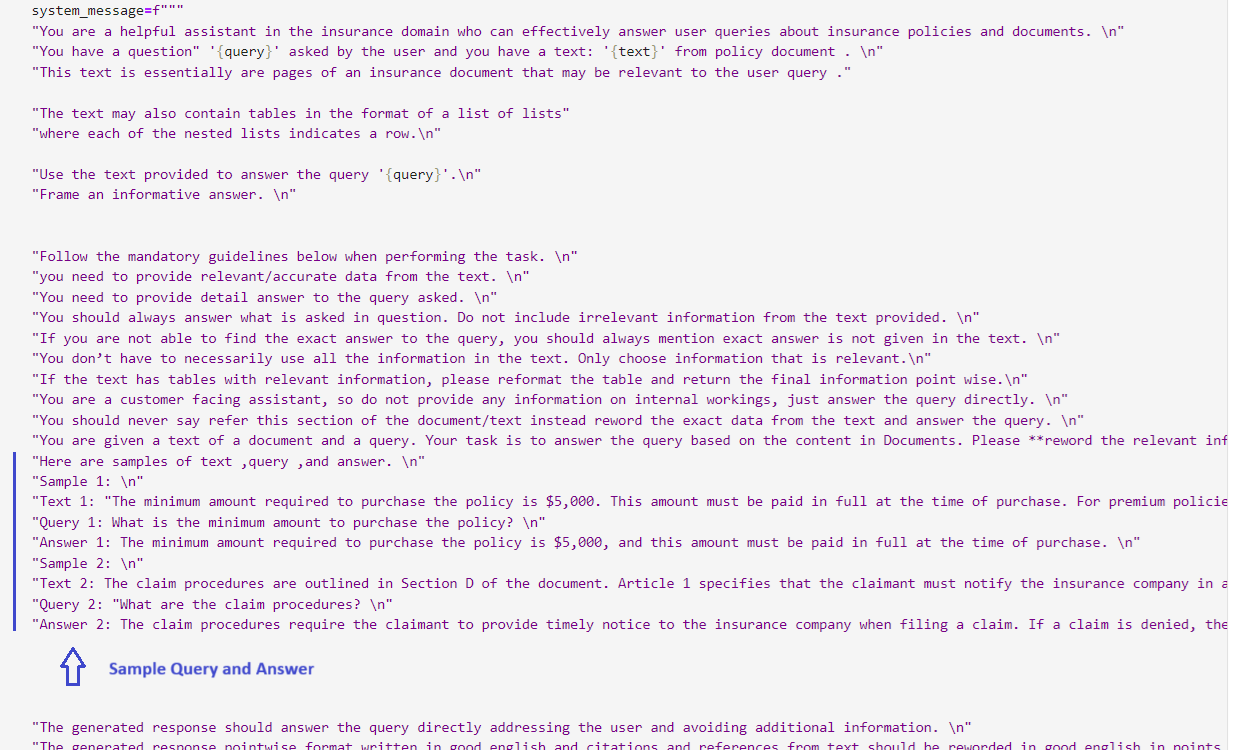
* **context:** I passed the text as a DataFrame containing two columns: metadata and the text of each PDF page. The metadata was a JSON object containing the page number and topic of the text. This approach was challenging because the prompt became complex, making it difficult to explain to the API how to effectively use the metadata.
* **Resolution:** I combined the text from all the searched pages and included the metadata. This approach simplified the prompt, requiring only two inputs: the query and the text. As a result, it became easier to explain how to search for the answer using the provided text. Additionally, this approach improved the quality of the API-generated answers while maintaining a consistent structure throughout.

**Challenges faced**

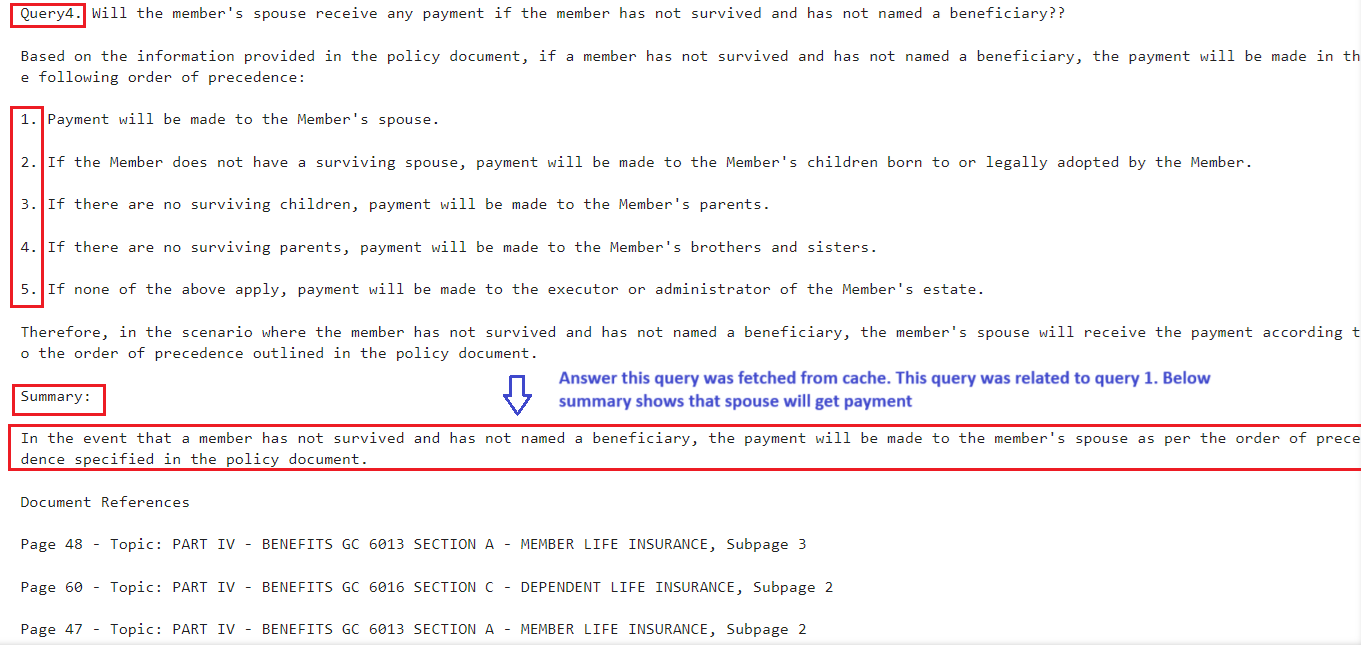
1. Prompt without example: The results were inconsistent, and the chat completion struggled to maintain a structured response.
2. Prompt with example: The answers generated were more consistent, and the structure of the responses was maintained.

**Lesson Learned:** we need to think of the eviction of records from the cache collection because cache can also become huge if every different query is stored in cache. Cache can almost equivalent to main collection over the period. We can use the logic where Cache can have only frequently asked questions.

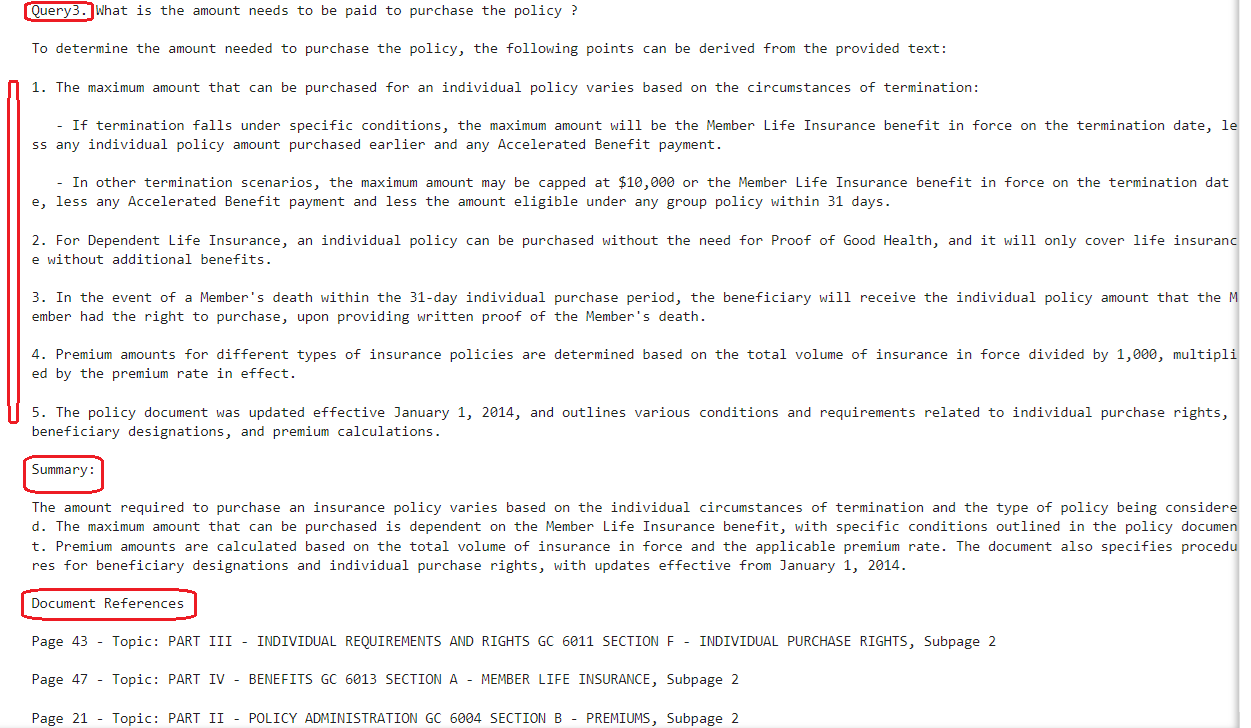
**Screenshot displaying sample query and answer in prompt**

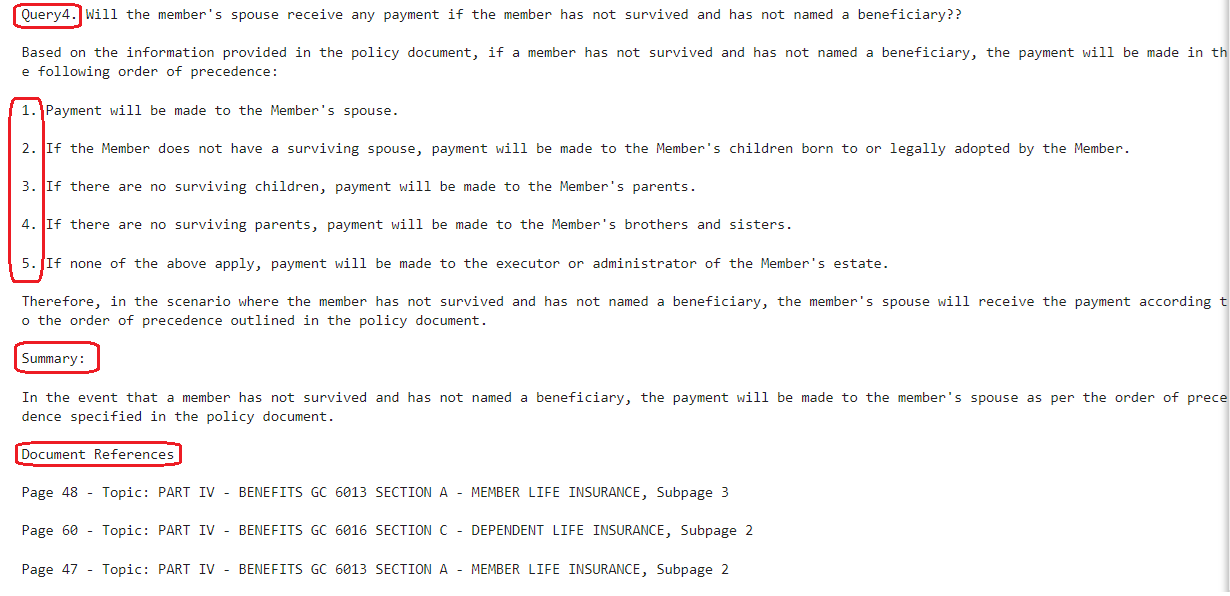
****

**Chat completion api generated response for query result fetched from Cache**

****

**Other answers generated by chat completion API**

****

****