Insurance Documents QA Chatbot (RAG & LangChain)

1. Introduction

This project details the development of a Question-Answering (QA) Chatbot specifically designed for insurance documents. The core objective is to move beyond general-purpose Large Language Models (LLMs) by implementing a **Retrieval-Augmented Generation (RAG)** pipeline using the **LangChain** framework. This ensures that the chatbot provides accurate, verifiable, and context-specific answers directly from policy documents, thereby mitigating the risk of LLM "hallucinations" in a high-stakes domain like insurance.

2. Project Objectives

The primary goals of this project were:

1. **Contextual Accuracy:** To build a QA system that can answer complex questions about insurance policies with high precision, strictly sourcing information from the provided documents.
2. **Scalable Document Handling:** To implement a robust mechanism for ingesting, chunking, and vectorizing various document formats (e.g., PDF) to create a searchable knowledge base.
3. **End-to-End RAG Pipeline:** To successfully integrate multiple components—Document Loader, Text Splitter, Embedding Model, Vector Store, Retriever, and LLM—into a cohesive and functional RAG chain using the LangChain framework.
4. **Usability:** To provide a simple interface (via a Jupyter Notebook) for demonstrating the setup, execution, and query-response capabilities of the chatbot.

3. System Design and Architecture

The system is built on a layered architecture following the standard Retrieval-Augmented Generation (RAG) pattern, leveraging the modularity of **LangChain**.

3.1. Overall System Design

The system operates in two distinct phases: the **Indexing Phase** (offline) and the **Retrieval/Generation Phase** (online query).

Phase	Component	Function
Indexing	**Document Loader**	Reads and loads unstructured data (e.g., PDF, TXT) from the `Policy+Documents` directory.

Insurance Documents QA Chatbot (RAG & LangChain)

Indexing	**Text Splitter**	Breaks down large documents into smaller, semantically coherent **chunks** to fit the LLM's context window and improve retrieval relevance.
	Embedding Model	Converts each text chunk into a dense, numerical **vector** (embedding).
	Vector Store	Stores the generated vectors and their corresponding text chunks in a searchable database (**ChromaDB** in this implementation).
RAG Chain	**Retriever**	Takes the user's query, converts it to an embedding, and searches the Vector Store to find the most relevant document chunks.
	LLM & Prompt Template	Receives the user's query and the retrieved context chunks. It uses a **system prompt** to instruct the LLM to generate a final, grounded answer based *only* on the provided context.

3.2. Project Layers

The project is conceptually separated into the following layers:

1. **Data Layer:** Contains the raw insurance policy files (`Policy+Documents`) and the persisted vector database (`chroma_persistence`).
2. **Preprocessing Layer:** This layer handles the conversion of raw documents into usable data points. It includes the **Document Loader** and the **Recursive Character Text Splitter**.
3. **Knowledge Base Layer:** The heart of the retrieval system. It encompasses the **Embedding Model** (for vector creation) and the **Vector Store** (for storage and similarity search).
4. **Application Logic (LangChain) Layer:** The orchestration layer where the RAG pipeline is defined. It chains the **Retriever**, **LLM**, and **Prompt** components together to form the final QA system.
5. **Interface Layer:** The user-facing component, implemented as a **Jupyter Notebook** (`langchain_notebook.ipynb`), which handles user input, executes the RAG chain, and outputs the final answer.

Insurance Documents QA Chatbot (RAG & LangChain)

4. Implementation Details

The implementation was carried out primarily in Python, leveraging the `langchain` library.

- **Dependencies:** Managed via `requirements.txt`, including `langchain`, `pypdf`, `chromadb`, and the relevant LLM client library (e.g., `openai` or `huggingface-hub`).
- **Vector Store: ChromaDB** was chosen for its ease of setup and persistence features, allowing the indexed knowledge base to be reused without re-indexing all documents every time.
- **Embedding Model:** A suitable embedding model (e.g., a high-quality open-source model or a commercial option like OpenAI's `text-embedding-ada-002`) was integrated to convert text into vectors.
- **LLM Integration:** The LLM was configured to function as a final answer generator, with a strict prompt instruction to use *only* the retrieved context chunks. This grounding is crucial for ensuring factual accuracy in the insurance domain.
- **Code Structure:** The entire RAG workflow is encapsulated within a single Jupyter Notebook for simplified step-by-step execution, demonstrating document loading, vectorization, and querying.

5. Challenges and Solutions

Challenge	Solution Implemented
Context Overload/Hallucination	Implemented a **`Stuff`** **chain type** with a carefully crafted **System Prompt** that explicitly forbids the LLM from using its general knowledge and directs it to answer only from the provided document snippets.
Finding Relevant Chunks	Fine-tuned the **`TextSplitter`** parameters (chunk size, overlap) to ensure that semantically related information remained within the same chunk, improving the quality of retrieval.
API Key Management	Utilized a **`.env`** file and the `dotenv` library to securely load API keys, keeping sensitive credentials out of the main codebase.

Insurance Documents QA Chatbot (RAG & LangChain)

PDF Handling	Employed the `PyPDFLoader` to reliably extract text from various PDF policy documents, addressing formatting challenges inherent in complex documents.

6. Lessons Learned

1. **RAG Effectiveness is 90% Retrieval:** The success of the QA system relies heavily on the quality of the chunks stored in the Vector Store. Optimal text splitting and a robust embedding model are more critical than the choice of the LLM for achieving factual correctness.
2. **The Power of Prompt Engineering:** A strong, directive system prompt is essential for controlling the LLM's behavior. Clear instructions to "Answer only based on the following context," are necessary to prevent the model from defaulting to its pre-trained knowledge.
3. **LangChain’s Modularity:** LangChain provided an invaluable abstraction layer, allowing for the rapid prototyping and swapping of components (e.g., switching from FAISS to ChromaDB) without rewriting the entire application logic.
4. **Persistence is Key:** Persisting the Vector Store (e.g., using `chroma_persistence`) is a critical production-readiness step, drastically reducing the time required to restart the application by eliminating the need to re-process all documents.