**Semantic Spotter: Project Report**

**1. Introduction**

**1.1 Project Goals**

Semantic Spotter is a Retrieval-Augmented Generation (RAG) chatbot designed to assist users in querying insurance policy documents. The primary goal is to enable natural language question answering over complex insurance PDFs by combining document retrieval with large language model (LLM) generation. This approach improves accuracy and relevance by grounding answers in the source documents.

**1.2 Data Sources**

The system uses insurance policy documents in PDF format as its primary data source. These documents contain detailed policy terms, conditions, and clauses. The example dataset includes the "Principal-Sample-Life-Insurance-Policy.pdf" file, which serves as the knowledge base for the chatbot.

**2. System Design and Architecture**

**2.1 Overview**

Semantic Spotter employs a multi-layered architecture combining document processing, vector embeddings, similarity search, and language generation:

* **Data Ingestion Layer:** Extracts raw text from PDF documents.
* **Preprocessing Layer:** Splits text into manageable chunks for embedding.
* **Embedding Layer:** Converts text chunks into vector embeddings using a custom embedding API.
* **Vector Store Layer:** Stores embeddings in a FAISS index for efficient similarity search.
* **Retrieval Layer:** Retrieves relevant document chunks based on user queries.
* **Generation Layer:** Uses a custom LLM API to generate natural language answers grounded in retrieved documents.
* **User Interface Layer:** Command-line interface for user interaction.

**2.2 Flowchart Description**

You can create a flowchart with the following components and flow:

1. **User Input:** User submits a query.
2. **Query Embedding:** The query is embedded using the CustomAPIEmbeddings class.
3. **Similarity Search:** FAISS vector store retrieves top relevant document chunks.
4. **Contextual Prompt Construction:** Retrieved chunks are combined with the query.
5. **LLM API Call:** The prompt is sent to the CustomLLM API for answer generation.
6. **Answer Output:** The generated answer is returned to the user.

Each layer should be represented as a box with arrows showing data flow from user input to answer output.

**3. Design Choices and Innovations**

**3.1 Use of Custom APIs**

Instead of relying on off-the-shelf embedding and LLM services, Semantic Spotter integrates custom embedding and LLM APIs. This allows flexibility in model selection, cost control, and potential customization for domain-specific tuning.

**3.2 Chunking Strategy**

The RecursiveCharacterTextSplitter is used with chunk size 1000 and overlap 100 to balance context retention and retrieval granularity. This ensures that document chunks are neither too large (which can dilute relevance) nor too small (which can lose context).

**3.3 Caching Embeddings**

To optimize performance, embeddings are cached locally in a .npy file. This avoids repeated API calls for the same documents, reducing latency and cost.

**3.4 Modular Architecture**

The system is designed with modular classes for embeddings, LLM, and chatbot logic, facilitating easy replacement or upgrading of components.

**4. Implementation Details**

* **PDF Text Extraction:** Using pdfplumber to reliably extract text from multi-page PDFs.
* **Embedding Generation:** CustomAPIEmbeddings class sends POST requests to the embedding API, handling JSON responses and errors.
* **Vector Store:** FAISS is used for fast nearest neighbor search over high-dimensional embeddings.
* **LLM Integration:** CustomLLM class handles prompt formatting, API calls, and streaming JSON response parsing.
* **RetrievalQA Chain:** LangChain's RetrievalQA is used to combine retrieval and generation seamlessly.
* **User Interface:** Simple command-line interface for interactive querying.

**5. Challenges Faced**

* **API Response Parsing:** The LLM API returns streaming JSON lines, requiring careful parsing to reconstruct the full answer.
* **Embedding Consistency:** Ensuring embeddings generated during caching and querying are consistent and compatible with FAISS.
* **PDF Text Quality:** Variability in PDF text extraction quality required robust handling of missing or malformed text.
* **Latency:** Balancing API call latency with user experience, mitigated by caching embeddings.
* **Error Handling:** Implementing robust error handling for network failures and unexpected API responses.

**6. Lessons Learned**

* Modular design greatly simplifies debugging and future enhancements.
* Caching intermediate data (like embeddings) is critical for performance in production systems.
* Handling streaming API responses requires careful design to avoid partial or corrupted outputs.
* Text chunking parameters significantly impact retrieval quality and should be tuned per dataset.
* Clear environment variable management is essential for secure and flexible deployment.

**7. Conclusion**

Semantic Spotter demonstrates an effective approach to building domain-specific chatbots by combining document retrieval with LLM generation. The system architecture balances innovation, modularity, and performance, providing a foundation for further enhancements such as multi-document support, richer UI, and advanced query understanding.

**8. Appendices**

**8.1 Environment Variables**

* EMBEDDING\_URL: URL for embedding API.
* CONVERSATION\_URL: URL for LLM API.
* API\_TOKEN: Authorization token.

**8.2 System Requirements**

* Python 3.8+
* Required libraries: langchain, pdfplumber, numpy, requests, faiss-cpu, python-dotenv

**Flowchart Diagram**

