Semantic Spotter Assignment: PolicyMate - AI-Powered Insurance Policy Q&A System

Description: Insurance policy documents are notoriously dense, lengthy, and filled with legal jargon. Policyholders, agents, and customer service representatives often struggle to quickly find specific information regarding coverage, exclusions, claim procedures, definitions, or specific clauses within these documents. Manually searching through PDFs or paper documents is time-consuming, inefficient, and prone to errors or omissions. This leads to frustration, potential misunderstandings of coverage, and increased operational costs for insurers (e.g., longer call times for support).

System Architecture & Workflow (Flowchart Description):

1. Data Ingestion:
    - Input: Collection of public domain insurance policy documents (PDF, TXT).
    - Process: Use LangChain DocumentLoader (e.g., PyPDFLoader, DirectoryLoader) to load the documents into memory.
2. Preprocessing & Chunking:
    - Input: Loaded documents.
    - Process: Use LangChain TextSplitter (e.g., RecursiveCharacterTextSplitter with appropriate chunk size and overlap) to break down documents into smaller, manageable chunks. Potential Innovation: Add metadata to chunks (e.g., original filename, section title if identifiable).
3. Embedding Generation:
    - Input: Document chunks.
    - Process: Use a LangChain Embeddings model interface (e.g., OpenAIEmbeddings, HuggingFaceEmbeddings with a model like sentence-transformers/all-MiniLM-L6-v2) to convert each chunk into a numerical vector representation.
4. Vector Storage:
    - Input: Embeddings and corresponding text chunks (with metadata).
    - Process: Store these embeddings in a LangChain VectorStore (e.g., FAISS, Chroma, Pinecone). This creates an indexed, searchable knowledge base.
5. Query Processing & Retrieval:
    - Input: User question (string).
    - Process (User Query): Embed the user's question using the same embedding model used for the documents.
    - Process (Retrieval): Use the VectorStore's similarity search capability (interfaced via LangChain Retriever) to find the chunks whose embeddings are most similar to the query embedding. Retrieve the top 'k' relevant chunks. Potential Innovation: Hybrid search or re-ranking here.
6. Context Augmentation & Prompting:
    - Input: User question and retrieved document chunks.
    - Process: Construct a prompt for the LLM. The prompt includes the original user question and the content of the retrieved chunks as context. Use LangChain PromptTemplate or ChatPromptTemplate for structuring this. Example: "Based only on the following context from the insurance policy

documents, answer the question. If the context does not contain the answer, say 'I cannot answer based on the provided documents'.\n\nContext:\n{context}\n\nQuestion:\n{question}\n\nAnswer:"

7. LLM Generation:
    - Input: Formatted prompt.
    - Process: Send the prompt to an LLM (e.g., ChatOpenAI, HuggingFaceHub) via the LangChain interface. The LLM generates an answer based on the provided context and question.
8. Output:
    - Input: LLM-generated answer.
    - Process: Return the answer to the user. Optionally include source document references derived from chunk metadata.

Appropriate Use of LangChain Components:

- DocumentLoaders: PyPDFLoader, UnstructuredFileLoader, DirectoryLoader for ingesting data.
- TextSplitters: RecursiveCharacterTextSplitter or CharacterTextSplitter for appropriate document chunking.
- Embeddings: OpenAIEmbeddings, HuggingFaceInferenceAPIEmbeddings, or SentenceTransformerEmbeddings for vectorization.
- VectorStores: FAISS (local, fast), Chroma (local, persistent), or cloud-based like Pinecone for storing and searching embeddings.
- Retrievers: Using the as_retriever() method on the vector store, potentially configuring search parameters (k, score threshold).
- Prompts: PromptTemplate or ChatPromptTemplate for structuring the input to the LLM effectively.
- LLMs/ChatModels: ChatOpenAI, HuggingFaceHub, or local models via Ollama for the generation step.
- Chains/LCEL: Using RunnablePassthrough, RunnableParallel, StrOutputParser, and composing the RAG logic with LCEL for a clear and maintainable pipeline. The standard RetrievalQA chain could be a starting point, but a custom LCEL chain offers more control.

Project Goals

The primary goal of the PolicyPal project is to develop a robust and accurate Generative AI-driven system capable of answering user questions based on the content of insurance policy documents. This system aims to address the challenge of extracting specific information efficiently from dense and complex legal documents.

Specific Objectives:

- Ingest and process insurance policy documents (initially PDFs).
- Implement a Retrieval-Augmented Generation (RAG) pipeline using the LangChain framework.

- Enable users to ask natural language questions about policy details (e.g., coverage, deductibles, exclusions, definitions).
- Provide answers generated by a Large Language Model (LLM), grounded strictly in the context retrieved from the source documents.
- Ensure the system is modular and configurable (e.g., easy to change models or data sources).

Data Sources

- Insurance policy document pdfs were gathered from upgrad learn portal.
- Format: The initial implementation focuses on PDF documents, utilizing PyPDFLoader. The system can be extended to handle other formats (.txt, .docx) by adding appropriate LangChain DocumentLoaders.
- Characteristics: The documents are expected to be text-based, potentially multi-column, and contain structured sections (though the system doesn't rely heavily on explicit structure beyond text flow). The complexity and length will vary depending on the specific policies used.

Design Choices

- Framework: LangChain was chosen due to its comprehensive tools and abstractions specifically designed for building LLM applications, particularly RAG systems. Its modularity (Loaders, Splitters, Embeddings, VectorStores, Retrievers, LLMs, Chains) allows for flexible development and experimentation. The LangChain Expression Language (LCEL) provides a powerful and declarative way to compose the RAG pipeline.
- Data Ingestion: DirectoryLoader combined with PyPDFLoader is used for loading PDF documents from a specified folder (documents/). This allows easy addition or removal of policy files. Multithreading is enabled for potentially faster loading.
- Text Splitting: RecursiveCharacterTextSplitter is employed to break documents into manageable chunks (chunk_size=1000, chunk_overlap=150). This method attempts to keep related text together by splitting recursively based on semantic separators (paragraphs, sentences, etc.). add_start_index=True adds metadata about the chunk's origin within the document.
- Embedding Model: OpenAIEmbeddings with the text-embedding-3-small model is used. This is a widely used, high-performance model suitable for generating semantic vector representations of text chunks. API key is managed via environment variables (.env file).
- Vector Store: FAISS (Facebook AI Similarity Search) is selected as the vector store. It's highly efficient for similarity searches, runs locally (using faiss-cpu), and integrates well with LangChain. The index is persisted locally in the vector_store/ directory, allowing reuse without reprocessing documents on every run unless forced (--rebuild flag).
- Retriever: The FAISS vector store is converted into a VectorStoreRetriever using vector_store.as_retriever(). It's configured (RETRIEVER_K=5) to fetch the top 5 most semantically similar chunks to the user's query embedding.
- LLM: ChatOpenAI with the gpt-3.5-turbo model is used for the generation step. This model provides a good balance of performance and cost. Temperature is set low (0.1) for more deterministic and factual answers based on the context. Max tokens are limited (512) to control output length. API key is managed via environment variables.

- RAG Pipeline (LCEL): The core logic is built using LCEL for clarity and composability:
    1. RunnableParallel fetches relevant context using the retriever (and formats it using format_docs) while simultaneously passing the original question through.
    2. The results are fed into the ANSWER_PROMPT (a ChatPromptTemplate).
    3. The formatted prompt is sent to the ChatOpenAI LLM.
    4. The LLM's output is parsed into a string using StrOutputParser.
- Prompt Engineering: A specific ANSWER_PROMPT is designed to instruct the LLM to:
    - Act as an insurance expert.
    - Answer only based on the provided context.
    - Explicitly state if the answer isn't found in the context.
    - Avoid hallucination.
- Configuration: Key parameters (paths, model names, chunk settings, API keys) are centralized in config.py and loaded from environment variables where appropriate (.env), promoting maintainability.
- User Interface: A command-line interface (main.py using argparse) is provided for interaction, supporting single queries and an interactive loop.

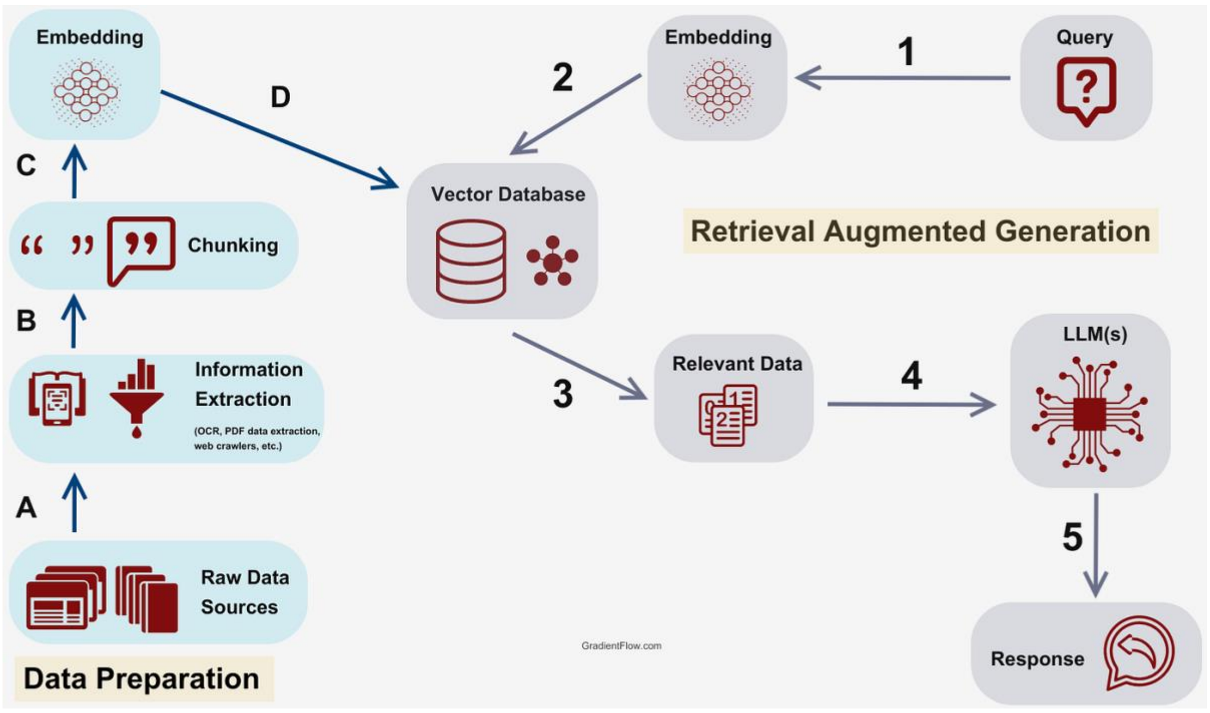
Flowchart Description

(This describes the flow; a visual diagram should accompany this in a full report)

1. Start: User initiates the process (e.g., runs main.py).
2. Check/Build Vector Store:
    - Does the FAISS index exist in vector_store/?
        - Yes: Load the existing index using FAISS.load_local() and the OpenAIEmbeddings model. Proceed to Step 6.
        - No (or --rebuild flag): Proceed to Step 3.
3. Load Documents: Use DirectoryLoader(PyPDFLoader) to read all PDF files from the documents/ directory.
4. Split Documents: Use RecursiveCharacterTextSplitter to break the loaded documents into smaller text chunks.
5. Embed & Store:
    - Initialize OpenAIEmbeddings.
    - Generate embeddings for each text chunk.
    - Create a FAISS index from the chunks and their embeddings.
    - Save the FAISS index locally to vector_store/.
6. Initialize LLM: Create an instance of ChatOpenAI.
7. Create Retriever: Create a retriever from the loaded/built FAISS vector store (vector_store.as_retriever()).
8. Build RAG Chain: Construct the query processing chain using LCEL, combining the retriever, prompt template, LLM, and output parser.
9. User Input: Receive a question from the user (either via command-line argument or interactive prompt).
10. Process Query (RAG Chain Execution):
    - Embed the user's question using the same OpenAIEmbeddings model.

- Use the retriever to find the top 'k' relevant document chunks from the FAISS index based on similarity to the query embedding.
- Format the retrieved chunks and the original question into the ANSWER_PROMPT.
- Send the formatted prompt to the ChatOpenAI LLM.
- Receive the generated answer text from the LLM.

11. Display Output: Print the LLM's generated answer to the user.
12. Loop/End: If in interactive mode, go back to Step 9. Otherwise, end the process.



Challenges Faced & Potential Improvements

- PDF Parsing Quality: PDFs can be complex. PyPDFLoader might struggle with unusual formatting, tables, images, or scanned documents requiring OCR. Mitigation: Use more robust loaders like UnstructuredPDFLoader (might require additional dependencies) or pre-process PDFs.
- Optimal Chunking: Finding the ideal chunk_size and chunk_overlap is crucial. Too small, and context is lost; too large, and irrelevant information dilutes the context passed to the LLM or exceeds context windows. Mitigation: Experiment with different values based on document structure and query types. Explore semantic chunking methods.
- Retrieval Relevance: Vector similarity search might sometimes retrieve chunks that are semantically similar but not directly relevant to answering the specific question nuance. Mitigation: Experiment with different embedding models. Increase k (number of retrieved chunks) cautiously. Implement a re-ranking step (e.g., using cross-encoders) after initial retrieval. Explore hybrid search (keyword + vector).
- LLM Hallucination/Accuracy: Despite prompt instructions, the LLM might occasionally hallucinate or misinterpret the provided context. Mitigation: Refine the ANSWER_PROMPT to be even more restrictive. Use more advanced models (e.g.,

GPT-4). Include source tracking (linking answers back to specific document chunks/pages).

- Context Window Limits: If many relevant chunks (k) are retrieved, their combined length might exceed the LLM's context window. Mitigation: Use LLMs with larger context windows. Implement summarization techniques for retrieved context before passing it to the final LLM call (e.g., using a Map-Reduce or Refine chain approach).
- Scalability: FAISS runs locally. For very large document sets, a cloud-based or distributed vector database (e.g., Pinecone, Chroma in client-server mode) might be necessary.
- Evaluation: Quantitatively evaluating the RAG system's accuracy requires a curated set of questions and expected answers (ground truth), which can be time-consuming to create. Improvement: Implement evaluation metrics like RAGAs or use LLM-based evaluation.

Example Runs:

```
(base) vijaymallepudi@Vijays-MacBook-Pro
Semantic_Spotter_Project % python main.py "Explain the HIV
cover
age"
Configuration loaded successfully.
2025-04-29 21:55:02,088 - INFO - Setting up PolicyPal RAG
system...
2025-04-29 21:55:02,088 - INFO - Initializing embedding model:
text-embedding-3-small
2025-04-29 21:55:02,120 - INFO - Loading existing vector store
from
/Users/vijaymallepudi/Documents/Semantic_Spotter_Project/vecto
r_store/policy_index...
2025-04-29 21:55:02,120 - INFO - Loading vector store from:
/Users/vijaymallepudi/Documents/Semantic_Spotter_Project/vecto
r_store/policy_index
2025-04-29 21:55:02,121 - INFO - Loading faiss.
2025-04-29 21:55:02,133 - INFO - Successfully loaded faiss.
2025-04-29 21:55:02,135 - INFO - Failed to load GPU Faiss:
name 'GpuIndexIVFFlat' is not defined. Will not load
constructor refs for GPU indexes.
2025-04-29 21:55:02,138 - INFO - Vector store loaded
successfully.
2025-04-29 21:55:02,138 - INFO - Vector store loaded
successfully.
2025-04-29 21:55:02,138 - INFO - Initializing LLM: gpt-3.5-
turbo
2025-04-29 21:55:02,151 - INFO - LLM initialized successfully.
2025-04-29 21:55:02,151 - INFO - Retriever configured to fetch
top 5 chunks.
2025-04-29 21:55:02,151 - INFO - Creating RAG chain...
```

```
2025-04-29 21:55:02,151 - INFO - RAG chain created
successfully.
2025-04-29 21:55:02,151 - INFO - System setup completed in
0.06 seconds.
2025-04-29 21:55:02,151 - INFO - Processing query: Explain the
HIV coverage
2025-04-29 21:55:03,304 - INFO - HTTP Request: POST
https://api.openai.com/v1/embeddings "HTTP/1.1 200 OK"
2025-04-29 21:55:04,499 - INFO - HTTP Request: POST
https://api.openai.com/v1/chat/completions "HTTP/1.1 200 OK"
```

--- Query ---
Explain the HIV coverage

--- Answer ---
Any sexually transmitted disease, or any condition related to HIV or AIDS is not covered under the insurance policy.
2025-04-29 21:55:04,521 - INFO - Query processed in 2.37 seconds.

```
(base) vijaymallepudi@Vijays-MacBook-Pro
Semantic_Spotter_Project %
```

```
(base) vijaymallepudi@Vijays-MacBook-Pro
Semantic_Spotter_Project % python main.py "Explain Maturity
Benefit "
Configuration loaded successfully.
2025-04-29 21:56:44,131 - INFO - Setting up PolicyPal RAG
system...
2025-04-29 21:56:44,131 - INFO - Initializing embedding model:
text-embedding-3-small
2025-04-29 21:56:44,174 - INFO - Loading existing vector store
from
/Users/vijaymallepudi/Documents/Semantic_Spotter_Project/vecto
r_store/policy_index...
2025-04-29 21:56:44,174 - INFO - Loading vector store from:
/Users/vijaymallepudi/Documents/Semantic_Spotter_Project/vecto
r_store/policy_index
2025-04-29 21:56:44,175 - INFO - Loading faiss.
2025-04-29 21:56:44,198 - INFO - Successfully loaded faiss.
2025-04-29 21:56:44,202 - INFO - Failed to load GPU Faiss:
name 'GpuIndexIVFFlat' is not defined. Will not load
constructor refs for GPU indexes.
2025-04-29 21:56:44,207 - INFO - Vector store loaded
successfully.
2025-04-29 21:56:44,207 - INFO - Vector store loaded
successfully.
2025-04-29 21:56:44,207 - INFO - Initializing LLM: gpt-3.5-
turbo
2025-04-29 21:56:44,220 - INFO - LLM initialized successfully.
```

```
2025-04-29 21:56:44,220 - INFO - Retriever configured to fetch
top 5 chunks.
2025-04-29 21:56:44,220 - INFO - Creating RAG chain...
2025-04-29 21:56:44,221 - INFO - RAG chain created
successfully.
2025-04-29 21:56:44,221 - INFO - System setup completed in
0.09 seconds.
2025-04-29 21:56:44,221 - INFO - Processing query: Explain
Maturity Benefit
2025-04-29 21:56:45,745 - INFO - HTTP Request: POST
https://api.openai.com/v1/embeddings "HTTP/1.1 200 OK"
2025-04-29 21:56:47,307 - INFO - HTTP Request: POST
https://api.openai.com/v1/chat/completions "HTTP/1.1 200 OK"

--- Query ---
Explain Maturity Benefit

--- Answer ---
The Maturity Benefit is the benefit paid to the Policyholder
on the Policy Maturity Date, provided the Policy remains in
force and the Life Assured survives until the Policy Maturity
Date. The amount of the Maturity Benefit depends on the
Guaranteed Benefit Option chosen by the Policyholder as
specified in the Policy Schedule.
2025-04-29 21:56:47,321 - INFO - Query processed in 3.10
seconds.
(base) vijaymallepudi@Vijays-MacBook-Pro
Semantic_Spotter_Project %
```