Semantic Spotter Project Report: Insurance Documents QA Chatbot (RAG with LlamaIndex & LangChain)

1. Objectives

The primary objective of this project is to build a Retrieval-Augmented Generation (RAG) system that enables users to query insurance policy documents and receive accurate, grounded answers. The system focuses on:

- Allowing natural-language questions over insurance documents.
- Retrieving the most relevant document excerpts using semantic search.
- Generating concise, evidence-backed answers that cite source message ids or document snippets.
- Providing a reproducible pipeline using LlamaIndex (for ingestion and indexing), ChromaDB (vector store), and LangChain/OpenAI components for generation and orchestration.

2. System Overview & High-Level Architecture

The system follows a layered RAG architecture. Major components include:
1. Ingestion Layer — parse source documents (PDFs, emails, or parsed JSON) into Document objects.
2. Embedding & Indexing Layer — embed documents using an embedding model (OpenAI or SentenceTransformers) and persist vectors in ChromaDB via LlamaIndex adapters.
3. Search & Retrieval Layer — semantic retrieval from the vector store with optional metadata filtering and a cache to speed repeated queries.
4. Re-ranking Layer — optional cross-encoder re-ranking to improve top-K results.
5. Generation Layer — LLM-powered answer generation that is explicitly grounded on retrieved excerpts; prompts instruct the model to cite source ids and avoid hallucination.
6. Serving & UI Layer — FastAPI/interactive loop/agent wrapper enabling conversational queries and returning final answers plus evidence.

3. Design: Detailed Layer Responsibilities

3.1 Ingestion Layer

Responsibilities:
- Parse raw documents (PDF, .eml, or JSON) and extract plain text and metadata (title/subject, date, message-id).
- Normalize text (remove headers/footers, fix encodings) and produce LlamaIndex Document objects with extra_info metadata.
- Preserve provenance (message ids, page numbers) to enable citation in generated answers.

- Uses LlamaIndex SimpleDirectoryReader and custom readers formats.
- Uses a SentenceSplitter for node/chunk creation when needed.

3.2 Embedding & Indexing Layer

Responsibilities:

- Choose and apply an embedding model (the notebook uses OpenAIEmbedding but the skeleton supports SentenceTransformers as an alternative).
- Create a Chroma vector store (ChromaVectorStore adapter) and build a VectorStoreIndex via LlamaIndex.
- Persist the index (storage context) so subsequent runs load quickly without re-embedding.
- The notebook shows initialization of a Chroma client and either creating or retrieving a collection named "Insurance_Doc_RAG_LlamaIndex_LangChain".
- After embedding, index.storage_context.persist is called to save the index.

3.3 Search & Retrieval Layer

Responsibilities:

- Accept natural-language queries, embed them with the same embedding model, and perform a nearest-neighbour search in Chroma.
- Return top-N candidate nodes/documents and associated scores/distances.
- Implement caching for repeated queries to reduce latency (the notebook uses a cache.set/get interface with expiration).
- A data_retrieval function wraps index.as_retriever() and retriever.retrieve(query) calls.
- retrieve_docs checks the cache first and writes results back when miss occurs.

3.4 Re-ranking Layer

Responsibilities:

- Improve final ordering of candidate documents by using a cross-encoder model that scores query-document pairs more precisely.
- Typically applied to the top 50 candidates to produce a final top-K list.
- The notebook includes patterns for using a CrossEncoder (from sentence-transformers) though exact placement may be in commented sections or optional blocks.

3.5 Generation Layer

Responsibilities:

- Take the final top-K retrieved excerpts and craft an LLM prompt that instructs the model to produce a concise, factual, and grounded answer.
- The prompt enforces: use only supplied excerpts, cite message ids or chunk labels, and refuse to answer if evidence is not present (avoid hallucination).
- The notebook builds prompts that include excerpts prefixed with identifiers like [MSG:message_id] and uses OpenAI Chat models (e.g., gpt-4o-mini or gpt-4o) via LangChain or llama_index llm adapters.
- Temperature is set to 0.0 for deterministic outputs and the prompt requires listing supporting message ids.

3.6 Serving & Orchestration Layer

Responsibilities:

- Provide an API (FastAPI) or interactive agent loop that accepts user queries and orchestrates retrieval, re-ranking, and generation.
- Maintain conversation history, store query logs and used evidence for auditing.
- The notebook includes a FastAPI endpoint and an interactive console-style loop using LangChain tools and AgentExecutor patterns.
- Tools for retrieval and document lookup are wrapped with tool decorators for agent usage.

4. Implementation: Key Functions & Workflow

Based on the notebook, the main workflow and functions are:

1. save_index(): Build index from documents in a folder (create/retrieve Chroma collection, build VectorStoreIndex, persist storage context).

2. load_index(): Load a persisted index by reattaching to the Chroma collection and creating a VectorStoreIndex from storage context.

3. data_retrieval(query, index): Convert index to a retriever and call retriever.retrieve(query) to fetch candidate nodes.

4. retrieve_docs(query): High-level wrapper that checks cache, calls data_retrieval on cache miss, and returns text snippets.

5. generate_answer(question, top_excerpts): Compose a strict prompt embedding the top excerpts with labels and call the LLM to produce the final answer.

6. Agent & API: Tools and an AgentExecutor are prepared (LangChain) so the system can be run in a loop or as an HTTP service.

5. Challenges Observed

During implementation and from analyzing the notebook code, the following challenges and trade-offs are evident:

1. Chunking vs. provenance: Treating each message as a single document preserves perfect provenance (message-id) for citations, but long documents (or combined thread dumps) may need chunking. Choosing chunk size affects retrieval quality.

2. Embedding choice & cost: Using OpenAI embeddings improves semantic matching but adds API cost and latency. Local SentenceTransformers models reduce cost but may slightly lower retrieval accuracy.

3. Hallucination risk: LLMs can fabricate facts if prompts or retrievals are insufficient. The notebook mitigates this by instructing the model to cite only provided excerpts and return 'I could not find the answer' when appropriate.

4. Re-ranking compute cost: Cross-encoder re-ranking improves quality but adds CPU/GPU cost, particularly for large candidate sets.

5. Indexing scale & persistence: Storing large corpora (e.g., Enron) requires careful persistence (Chroma settings) and incremental reindexing strategies.

5. Lessons Learned & Best Practices

Key takeaways from the project and notebook implementation:
- Preserve provenance: Always keep identifiers (message-id/page) with each chunk so generated answers can cite sources.
- Grounded prompts: Insist that the LLM uses only provided excerpts and returns explicit citations; set temperature to 0 for consistent outputs.
- Cache frequent queries: Use a TTL cache to reduce embedding & retrieval latency for repeated queries.

6. Recommendations & Next Steps

Suggested improvements and experiments:
1. Implement an incremental indexer to add new documents without full reindexing.
2. Add a UI for humans to validate and flag incorrect answers; feed feedback into fine-tuning or re-ranking heuristics.
3. Add stricter redaction and role-based access control if the corpus contains sensitive information.
4. Integrate a lightweight vector search monitoring dashboard to track query performance and top retrieved docs.
5. Automate end-to-end tests that assert that gold-query answers cite the expected message ids.