Legal RAG Assistant

Executive Summary & Business Value

This tool transforms static legal documents into an interactive, intelligent knowledge base.

- **High-Precision Retrieval**: By combining **Semantic Search** (understanding meaning) with **Keyword Search** (exact matching of Articles/Sections), we ensure no critical legal clause is missed.
- **Zero-Hallucination Architecture**: The system is engineered to answer *only* from uploaded documents. If the answer isn't there, it explicitly states so, preventing legal misinformation.
- **Auditability**: Every AI-generated answer is backed by **"Retrieved Evidence"**—direct excerpts from the source document displayed in the UI, allowing lawyers to verify accuracy instantly.
- **Scalable & Modular**: Built on production-grade components (Pinecone, Elasticsearch, OpenAI), the system handles complex legal datasheets without performance degradation.

System Architecture

The application implements a **Hybrid Retrieval-Augmented Generation (RAG)** pipeline:

1. **Ingestion**: Documents are sliced into overlapping chunks and indexed in parallel.
   - *Semantic Index (Pinecone)*: Stores vector embeddings for conceptual matching.
   - *Keyword Index (Elasticsearch)*: Stores raw text for precise term matching (e.g., "Article 14").
2. **Hybrid Retrieval**: Queries are broadcast to both indexes. Results are fused using a **Weighted Reciprocal Rank** algorithm to pick the absolute best matches.
3. **Context Injection (Top K)**: The **Top 5** most relevant chunks are retrieved and injected directly into the LLM's context window.
4. **Generation**: The LLM synthesizes an answer solely from the injected chunks, citing page numbers.

Why InLegalBERT? (Model Selection)

We utilize `law-ai/InLegalBERT` for embedding, rather than standard models like `bert-base` or `all-MiniLM`.

- **Domain Specificity**: Standard models are trained on Wikipedia and general web text. `InLegalBERT` is pre-trained specifically on vast datasets of legal statutes, case laws, and contracts.
- **Jargon Understanding**: It natively understands legal polysemy. For example, it knows that "Consideration" in a contract means *payment/value exchange*, not just "thinking about something."
- **Superior Retrieval**: In benchmarks, this model consistently ranks legal provisions higher than generic models, ensuring that specific clauses (like *Force Majeure* or *Termination*) are found accurately.

Reducing Hallucinations

Legal AI cannot afford to make things up. We implement a **Three-Layer Safety Protocol**:

1. **Strict Context Injection**: We pass the **Top K (5)** retrieved chunks to the LLM. The model is forced to use *only* this data as its "worldview" for the current question.
2. **Negative Constraints**: The System Prompt explicitly instructs the model: *"If the answer is not in the context, strictly state: 'I could not find the answer in the provided document.' Do not hallucinate."*.
3. **Evidence Display**: The UI includes a "View Retrieved Evidence" dropdown. This allows the human user to validate that the AI's answer is actually grounded in the text provided.

File & Folder Structure

```
legal-rag/

  app.py                 # Main Application / UI
  chunker.py             # Document Processor
  embedder.py            # AI Vector Model
  llm_answer.py          # Generation Engine

  Search & Indexing
     retrieval.py           # Vector Search Logic
     retrieval_hybrid.py    # Hybrid Fusion Logic
     elastic_search.py      # Keyword Search Logic
     pinecone_search.py     # (Utility) Pinecone connection
     pinecone_store.py      # Vector Database Manager
     elastic_store.py       # Keyword Database Manager

  dataset/               # (Folder for raw PDF/text files)
```

File Descriptions

- **`app.py`**: The Streamlit frontend. It orchestrates the entire workflow, handling file uploads, triggering the indexing process, managing the chat history, and rendering the "Evidence" verification blocks for users.
- **`chunker.py`**: Reads PDF or Text files and splits them into smaller "chunks" (300 words). It uses a **sliding window** technique (50-word overlap) so that sentences split across chunks don't lose their meaning.
- **`embedder.py`**: Loads the `InLegalBERT` model. It converts the text chunks into mathematical vectors (lists of numbers) that capture the semantic meaning of the legal language.
- **`llm_answer.py`**: Connects to OpenAI. It takes the user's question and the retrieved text chunks, combines them into a strict prompt, and returns the final answer along with citations.
- **`retrieval_hybrid.py`**: The "Brain" of the search system. It calls both Pinecone (Vector) and Elasticsearch (Keyword), normalizes their scores, and combines them (weighted 60/40) to find the most accurate results.
- **`retrieval.py`**: Performs the semantic search in Pinecone. It includes **custom boosting logic** where chunks containing "Article" or "Schedule" get a score bonus to prioritize specific legal clauses.
- **`elastic_search.py`**: Performs exact text searching in Elasticsearch. It boosts matches found in the "title" or "article" metadata fields to ensure precise lookup of known sections.
- **`pinecone_store.py`**: Manages the Pinecone vector database. It handles the creation of indexes and the "upsert" (upload/insert) of vector batches during the data ingestion phase.
- **`elastic_store.py`**: Manages the Elasticsearch database. It extracts metadata (like Article numbers) via Regex and creates searchable documents for the keyword engine.

Getting Started

Prerequisites

- Python 3.9+
- API Keys: OpenAI, Pinecone, Elasticsearch

Installation

1. **Clone & Install**:

```
git clone <repo>
pip install -r requirements.txt
```

2. **Environment Setup** (`.env`):

```
OPENAI_API_KEY=sk-...
OPENAI_ASSISTANT_ID=...

PINECONE_API_KEY=...
PINECONE_INDEX=...
PINECONE_REGION=...

ES_URL=...
ES_API_KEY=...
ES_INDEX=...
```

(Note: Ensure `PINECONE_INDEX` is consistent across all files to avoid retrieval errors.) 3. **Run**:

```
streamlit run app.py
```