Conversastional RAG System

1. Overview

The PalmMind backend system is designed for intelligent document ingestion and conversational question answering using **Retrieval-Augmented Generation (RAG)**. It exposes two modular RESTful APIs via FastAPI:

- **Document Ingestion API**:
  - Supports `.pdf` and `.txt` file uploads
  - Offers selectable chunking strategies
  - Extracts and embeds text using foundation models
  - Stores embeddings in Pinecone with metadata in a database
- **Conversational RAG API**:
  - Facilitates multi-turn conversation using Redis for session memory
  - Retrieves semantically relevant chunks
  - Supports interview booking (name, email, datetime)
  - Sends confirmation emails via SMTP

---

2. Features Implemented

✅ Upload and ingest .pdf/.txt documents with support for multiple chunking strategies
✅ Text embeddings using **Google Generative AI (text-embedding-004)**
✅ Vector storage and semantic retrieval using **Pinecone**
✅ **Redis-backed memory** for stateful chat history
✅ **Interview scheduling system** with email confirmation
✅ Modular, typed, and maintainable codebase with clear separation of concerns

---

3. Evaluation Setup

Dataset

Two academic papers were used for benchmarking:

- *"Efficient Estimation of Word Representations in Vector Space"*
- *"Attention Is All You Need"*

A set of **12 question-answer pairs** were generated using an LLM, where:

- The **question** served as the query for vector retrieval (top_k = 5)
- The **answer** was treated as ground truth for evaluating similarity with retrieved chunks

Embedding Setup

- For vectorstore indexing: `llama-text-embed-v2-index` (Pinecone-native)
- For evaluation: `GoogleGenerativeAIEmbeddings` using `text-embedding-004`
- Similarity computation:
  - **Cosine Similarity** for top_k vector matches
  - **Dot Product** using `sklearn.metrics.pairwise.linear_kernel`

A **similarity threshold** of **0.80** was set to mark a match as "correct." A previous threshold of 0.75 yielded perfect scores, so it was raised for more meaningful differentiation.

---

4. Benchmark Results

Method	Accuracy	Recall	Avg. Latency (s)	Correct Answers
Recursive Chunker + Cosine Similarity	83.33%	83.33%	1.9391	10 / 12
Token Chunker + Cosine Similarity	50.00%	50.00%	1.3300	6 / 12
Fixed Size Chunker + Dot Product	75.00%	75.00%	0.4000	9 / 12
Token Chunker + Dot Product	33.33%	33.33%	0.4100	4 / 12

Note: Precision and F1 were not calculated due to the binary relevance approach (retrieved vs. not matched).

---

5. Key Findings

- **RecursiveCharacterTextSplitter** significantly outperforms token-level chunking in both **accuracy** and **recall**, regardless of similarity metric used.
- **Cosine similarity** yields better performance than dot product, likely due to alignment with the training objective of `llama-text-embed-v2-index`.
- **Token-based splitting** underperforms, possibly due to poor semantic coherence across token boundaries.
- **Latency** is notably lower for dot product methods, but at the cost of reduced retrieval quality.

6. Recommendations

- Using **RecursiveTextSplitter** with appropriate chunk overlap as the **default chunking strategy**.

- Favoring **cosine similarity** for vector-based retrieval when using embeddings aligned with that distance metric.

- Explore finer-tuned **chunk sizes and overlaps** for optimized trade-offs between performance and speed.

- Considering evaluating with **precision and F1-score** if moving toward graded relevance or multi-chunk retrieval.

- For production-grade QA systems, incorporating **windowed re-ranking** or **fusion techniques** (e.g., MaxP or ColBERT-style pooling) to further enhance answer quality.