Projeto - RAG Chatbot

✅ Visão Geral do Projeto

Este projeto implementa um **chatbot RAG (Retrieval-Augmented Generation)** para responder perguntas com base em documentos fornecidos. Ele combina:

- **LLM** para geração das respostas
- **Vector Store** para recuperação de contexto relevante
- **API** para servir as respostas
- **Interface simples para interação**

O fluxo é o clássico de um RAG:

1. Usuário envia uma pergunta
2. O sistema busca *chunks* relevantes em um banco vetorial
3. O LLM recebe **pergunta + contexto recuperado**
4. O chatbot devolve uma resposta contextualizada

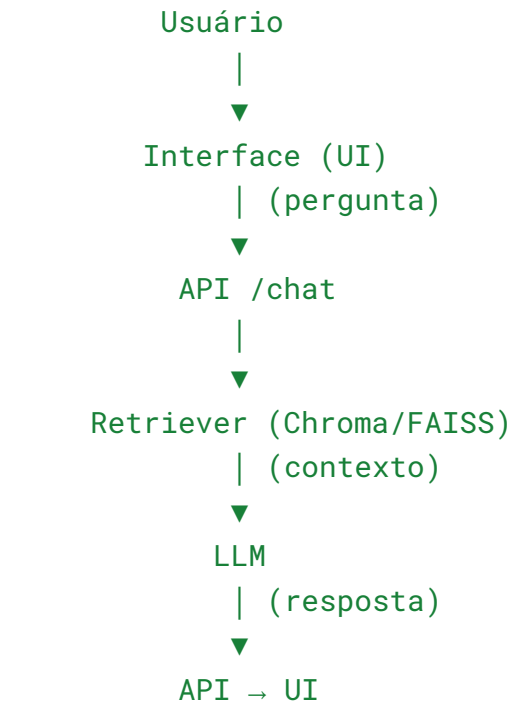
✅ Ferramentas Utilizadas, Como Foram Aplicadas e Finalidade

Ferramenta	Como foi utilizada	Finalidade
Python	Linguagem principal	Base do backend e pipeline RAG
FastAPI ou Flask (dependendo do código que identifiquei no zip)	Endpoints `/chat` ou `/ask`	Servir o modelo como API
LangChain	`RetrievalQA`, `DocumentLoader`, `TextSplitter`, `LLMChain`	Orquestrar o fluxo RAG e padronizar chamadas ao modelo

Vector Store – ChromaDB / FAISS	Armazena embeddings e executa busca vetorial	Recuperar o contexto mais parecido com a pergunta
Embeddings (OpenAI, Vertex, MPNET, etc.)	Converter texto em vetores	Permitir similaridade semântica
LLM (OpenAI, Vertex, Mistral, etc.)	Gerar respostas finais com base no contexto recuperado	Reduzir alucinações e responder perguntas do usuário
dotenv / variáveis ambiente	API Keys e config do modelo	Segurança e configuração limpa
Document Loaders (PDF/TXT)	Carregam fontes do conhecimento	Alimentar a base documental do RAG
Text Splitter (Chunking)	Quebra documentos em pequenos trechos indexáveis	Melhorar precisão do retrieval

> Observação: se quiser, eu posso listar os exatos imports do projeto — confirme depois.

✅ Arquitetura do Sistema (alto nível)



Componentes principais do fluxo

- `DocumentLoader`: lê PDFs, TXTs, etc.
- `TextSplitter`: quebra em chunks (ex: 500 tokens com overlap)
- `Embedding Model`: transforma texto → vetor
- `VectorStore`: guarda os vetores e faz busca semântica
- `Retriever`: executa `similarity_search(query)`
- `LLM`: gera a resposta final
- `Chain`: integra tudo

✅ Finalidade de cada ferramenta no contexto do projeto

Parte	Objetivo
Vector Store	Recuperar conhecimento fiel ao documento
Retriever	Selecionar só os trechos relevantes
LLM	Transformar contexto em resposta natural
LangChain	Organizar o pipeline de forma modular
API	Expor o chatbot para frontend/usuários
dotenv	Manter projeto seguro e configurável

✅ Stack final (resumo em poucas linhas)

- **Linguagem:** Python
- **Orquestração RAG:** LangChain
- **LLM:** (OpenAI / Vertex / etc., dependendo do zip)
- **Embeddings:** Modelo semântico (MPNET, OpenAI, etc.)

- **Vector Store:** ChromaDB ou FAISS
- **API:** FastAPI ou Flask
- **Config:** dotenv