RAG Chatbot Demo Explainer

# Project Summary

RAG Chatbot Demo is a modular retrieval-augmented generation reference project designed to answer questions over custom knowledge bases.

The repository demonstrates how to ingest multi-format documents, create FAISS vector indices, run retrieval strategies, and generate grounded answers through OpenAI models with FastAPI integration underway.

# Key Capabilities and Design Principles

- YAML-driven configuration in `configs/rag.yaml` keeps document processing, embeddings, retrieval, and LLM settings centralized.

- Document ingestion supports PDF, TXT, Markdown, HTML, and DOCX files with configurable chunking (size 1000, overlap 200).

- Retrieval layer wraps FAISS similarity search and Maximal Marginal Relevance for diverse yet relevant document recall.

- Prompt management composes system and user templates compatible with OpenAI chat completions.

- FastAPI backend scaffolding prepares REST endpoints and a forthcoming chat interface.

# End-to-End Workflow

The pipeline follows a retrieve-augment-generate loop orchestrated by `src/rag\_chain.py`.

## Step 1 - Document Ingestion (src/ingest.py)

Loads configuration, walks the `data/` directory, normalizes supported documents, chunks text, embeds with sentence-transformers/all-MiniLM-L6-v2, and persists vectors to `data/vector\_index`.

## Step 2 - Retrieval (src/retriever.py)

Reconstructs the FAISS index, initializes the embedding model, and exposes CLI helpers for similarity and MMR searches with configurable top-k and lambda.

## Step 3 - Prompting (src/prompt.py)

Formats retrieved chunks into chat messages, weaving document metadata into prompts while honoring the OpenAI API contract.

## Step 4 - RAG Chain (src/rag\_chain.py)

Combines retrieval, prompt assembly, and OpenAI calls into a single class with a query helper that can optionally return source context for inspection.

## Step 5 - API Layer (app.py)

Bootstraps FastAPI with lifespan hooks, Pydantic models, and CORS middleware; REST endpoints and the HTML chat interface remain to be implemented.

# Configuration and Data Assets

Primary settings live in `configs/rag.yaml`, covering chunking, embeddings, FAISS paths, retrieval parameters, LLM configuration, and logging defaults.

`data/sample.txt` provides starter content, while `data/vector\_index/` stores a prepared FAISS index (`faiss.index`, `chunks.pkl`, `config.pkl`) generated by the ingestion script.

`notebooks/` is reserved for experimentation, and the planned evaluation module (`src/eval.py`) will track relevancy and groundedness metrics.

# Running the Pipeline

Activate a Python 3.9+ environment, install dependencies from `requirements.txt`, and export an `OPENAI\_API\_KEY`.

Execute `python src/ingest.py --data-dir data` to rebuild the vector index when documents change.

Use `python src/rag\_chain.py --question "What is RAG?"` for end-to-end question answering, or `python src/retriever.py --query "RAG systems" --method mmr` to inspect retrieval output.

Start the forthcoming web app with `uvicorn app:app --reload` once REST routes and the chat UI are finalized.

# FastAPI Status and Roadmap

Current FastAPI skeleton defines application lifespan events, CORS configuration, and placeholder request and response models.

Next deliverables include `/ask` and `/health` endpoints, server-side orchestration of the RAG chain, and an HTML front end for interactive chat.

Post-API work can extend into automated evaluation, additional data sources, deployment automation, and broader LLM provider support.

# Appendix - Repository Layout

Key directories: `configs/` (YAML settings), `src/` (ingestion, retrieval, prompts, RAG chain), `data/` (documents and FAISS artifacts), `app.py` (FastAPI app), `notebooks/` (experiments). See `PROJECT\_STRUCTURE.md` for a progress-tracked tree.