RAG Publication: Jacmate Research Assistant

Author: Ugochukwu Febechukwu

Date: 2026-01-06

Abstract

This short publication describes the Jacmate project: a Retrieval-Augmented Generation (RAG) application that ingests research publications into a persistent ChromaDB vector store, uses LangChain tools for preprocessing, embedding, and retrieval, and integrates OpenAI models (via LangChain) to produce human-readable answers to research questions.

1. Project Overview

Jacmate is a RAG system built to turn a local collection of research PDFs and text files into a searchable knowledge base.

Key components in the repository:

- - `inserting_file.py`: loads PDFs and text files into plain text using LangChain community loaders.
- - `jac_functions.py`: handles chunking, embedding, insertion into ChromaDB, retrieval, and prompt construction.
- - `insert_chroma.py`: example script that calls the ingestion flow and saves vectors into a persistent ChromaDB folder (`./research_db`).
- - `app.py`: a FastAPI server that exposes an `/ask` endpoint to query the RAG system.

2. Data ingestion into ChromaDB

Flow summary: documents are discovered and loaded from disk, split into chunks, embedded with a sentence embedding model, and inserted into a Chroma collection named `ml_publications` in a persistent database.

Implementation highlights:

- - File loading: `inserting_file.py` uses `PyPDFLoader` for PDFs and `TextLoader` for `.txt` files; it traverses folders (including subfolders) and collects page contents.
- - Chunking: `jac_functions.chunk_research_paper` uses `RecursiveCharacterTextSplitter` with 1000-character chunks and 200-character overlap to preserve context.
- - Embeddings: `jac_functions.embed_documents` constructs a Hugging Face sentence-transformers embedding model (e.g. `all-MiniLM-L6-v2`) and runs `embed_documents`. It selects `cuda`, `mps`, or `cpu` automatically when available.
- - Storage: `chromadb.PersistentClient(path="./research_db")` creates/opens a persistent store; `collection.add(...)` persists embeddings, ids, documents, and metadata.

3. LangChain usage

LangChain is used throughout the pipeline for convenience and interoperability. The project uses:

- - `langchain_community.document_loaders` (e.g., `PyPDFLoader`, `TextLoader`).
- - `RecursiveCharacterTextSplitter` from `langchain_text_splitters` for chunking.
- - `langchain_huggingface.HuggingFaceEmbeddings` to produce dense vector embeddings compatible with Chroma.
- - `langchain.prompts.PromptTemplate` to build the final question prompt that includes retrieved context.

4. Retrieval and generation (RAG)

When a question arrives, the system obtains a query vector with the same embedding model, queries Chroma for the top-k similar chunks, then constructs a prompt containing the retrieved chunks and the user question. An LLM (via LangChain OpenAI wrapper) generates the final human-readable answer.

Key functions:

- - `search_research_db(query, collection, embeddings, top_k)`: embeds a query via `embeddings.embed_query(query)` and calls `collection.query(...)` to obtain documents and distances.
- - `answer_research_question(query, collection, embeddings, llm)`: calls `search_research_db`, assembles a `PromptTemplate` with the top results, then calls the LLM to generate an answer (via `llm.invoke(prompt)`).

5. OpenAI integration (GPT model options)

The project uses LangChain's `ChatOpenAI` wrapper to call OpenAI chat models. Although the repository contains commented examples referencing multiple models, you can configure the deployed model. For example, to use GPT-3.5 (chat) with LangChain:

- Example configuration:

```
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0.7)
```

Notes:

- - The current `app.py` instantiates `ChatOpenAI(model_name="gpt-4o-mini", ...)` in the `/ask` handler; swap the `model_name` to `gpt-3.5-turbo` to use GPT-3.5.
- - Set your OpenAI key in an `.env` file: `OPENAI_API_KEY=your_key` and ensure `python-dotenv` loads it (the project already calls `load_dotenv()`).

6. Deployment and usage

To ingest files into ChromaDB:

- - Run `python insert_chroma.py` (this calls `insert_publications` and stores embeddings in `./research_db`).

To run the API server:

- - Run `python app.py` or `uvicorn app:app --host 0.0.0.0 --port 8000`.

Query example (curl):

curl -X POST "http://localhost:8000/ask" -H "Content-Type: application/json" -d '{"question": "What are the main contributions in the ML papers?"}'

7. Reproducibility & notes

- - The persistent vector DB lives in `./research_db` (including a `chroma.sqlite3` file).
- - The project is pinned to core dependencies in `requirements.txt` (LangChain, ChromaDB, Hugging Face sentence-transformers, Torch, etc.).
- - Embedding model: `all-MiniLM-L6-v2` (efficient and popular for semantic search).
- - Consider model, privacy, and cost tradeoffs when selecting an OpenAI model (GPT-3.5 vs GPT-4 family).

Quick Pointers

• Files to inspect: `inserting_file.py`, `jac_functions.py`, `insert_chroma.py`, `app.py`.

• To switch LLM models, edit `app.py` and/or supply a different `ChatOpenAI` model_name. Ensure your `OPENAI_API_KEY` is set in the environment.

A Note to Readers

Thanks for taking a look — this project is designed to be approachable for researchers, students, and engineers. The following sections give practical, copy-paste-ready examples and a small architecture diagram to make it easy to reproduce and extend.

Appendix: Quick Start Snippets

Below are concise snippets to help you reproduce the main flows. These are intentionally short; full examples are in the repository.

Ingest files (example):

```
from inserting_file import load_pdf_to_strings
from jac_functions import insert_publications
import chromadb

client = chromadb.PersistentClient(path='./research_db')
collection = client.get_or_create_collection(name='ml_publications')
```

```
pubs = load_pdf_to_strings('data/400 Level/1st Semester')
insert_publications(collection, pubs, title='400 level')
```

Query example (python):

```
import requests
resp = requests.post('http://localhost:8000/ask', json={'question':'What are
the main contributions in the ML papers?'})
print(resp.json())
```

Curl example:

curl -X POST "http://localhost:8000/ask" -H "Content-Type: application/json" -d "{"question": "What are the main contributions in the ML papers?"}"

Sample Q&A (example)

Question: What are the main contributions in the ML papers?

Example (generated by an LLM):

The papers introduce a lightweight embedding approach using all-MiniLM-L6-v2 for efficient retrieval, apply chunk-based indexing to preserve context, and show improved retrieval recall when overlapping chunks are used. They also demonstrate the practical integration of a cloud LLM (e.g., GPT-3.5) to synthesize concise, human-readable summaries for end users.

Model configuration & Client snippets

Use this snippet to configure LangChain to call GPT-3.5 (chat) via the `ChatOpenAI` wrapper:

```
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0.7)
```

Quick Python client that calls the `/ask` endpoint and prints the answer and sources:

```
import requests
payload = {'question':'Summarize the primary contributions across these
papers.'}
resp = requests.post('http://localhost:8000/ask', json=payload)
data = resp.json()
print('Answer:
', data.get('answer'))
print('
Sources:')
for s in data.get('sources', []):
```

```
    print(f"- {s.get('title')} (similarity={s.get('similarity'):.2f})")
```

GPT-3.5 sample outputs & considerations

Below are two example outputs you might get from `gpt-3.5-turbo`. The first is a concise summary, the second is a longer, cited synthesis using retrieved chunks:

Concise (short):

These papers present an efficient embedding and retrieval pipeline using an all-MiniLM-L6-v2 model, chunked indexing with overlaps to preserve context, and an LLM-based synthesizer that compiles high-level findings into readable summaries.

Cited (synthesis):

Overview:
1) Embedding & Efficiency — authors use a MiniLM family model to provide compact, meaningful vectors for scalable search.
2) Chunking Strategy — overlapping 1000-character chunks help preserve context across boundaries and improve retrieval recall.
3) Application — combining retrieval with an LLM produces concise, human-friendly summaries that cite supporting chunks when useful.

(Example inline citations: From 400 level_0 and 400 level_3)

Cost & latency notes:

- - `gpt-3.5-turbo` is typically cheaper and faster than GPT-4 models; expect low single-digit-second latencies for typical prompts, but this depends on prompt length and model load.
- - Consider truncating or summarizing retrieved context to reduce prompt size and cost if you plan to run many queries at scale.

Architecture Diagram

A small diagram summarizes the pipeline; see the `docs/diagram.png` and `docs/diagram.svg` files (if present).

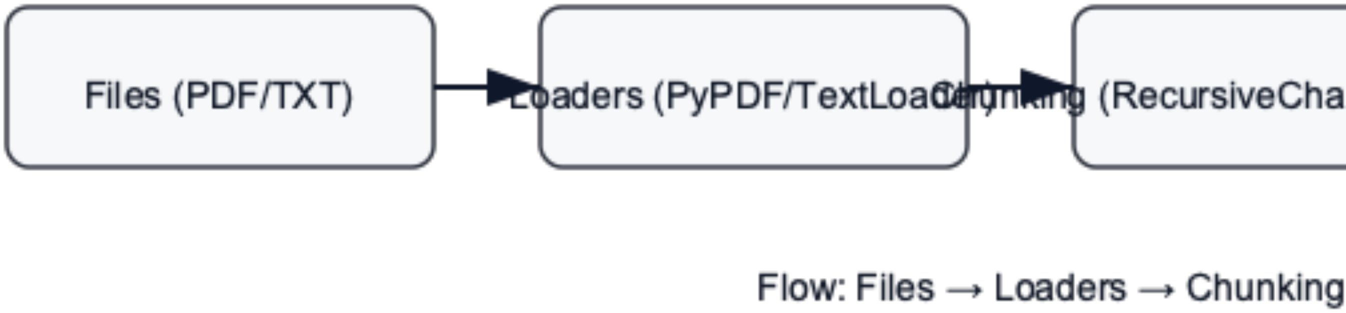

Figure: Jacmate RAG pipeline — files, loaders, chunking, embeddings, ChromaDB, retriever, and LLM.

Writing effective questions for RAG

To get concise, useful answers when querying the system, try:

- - Be specific: include target terms or desired scope (e.g., "summarize contributions about data augmentation").
- - Ask for citations: request evidence or references when you need supporting material.
- - Prefer short, focused questions for quick responses; ask follow-ups for deeper exploration.

Contributing & Next Steps

Want to improve the project? Consider:

- - Adding more diverse papers and labels for evaluation.
- - Adding unit tests for embedding and retrieval functions.
- - Adding a small UI to explore retrieved context and model outputs.

If you want, send a PR or open an issue and note the tag `help-wanted`.