云南大学软件学院期末课程报告

Final Course Report

School of Software, Yunnan University

个人成绩

学号	姓名	专业	成绩
20243120018	ROY SUDIPTA	SOFTWARE ENGINEERING	

学　　期： 3RD Semester

课程名称: Professional Training -YN3012170034

任课教师： Ahamad Zahir

题目： A Containerized RAG Architecture for Efficient Document Retrieval using BGE Embeddings and Qdrant

联系电话: 13170632019

电子邮件: shudiptoroy6@gmail.com

报告完成时间： 年　　月　日

An efficient Document Retrieval Containerized RAG Architecture with BGE.

Author: ROY SUDIPTA

Yunnan University

Abstract:
This project develops a Retrieval Augmented Generation (RAG) application, which allows an individual to upload PDF files, search through them with the help of semantic similarity, and receive an answer based on a context retrieved only. Sentence Transformers deployed a transformer based embedding model in the system to transform text chunks into vectors. It caches such vectors in Qdrant, a similarity search optimized vector database. For generation the system is calling out from locally hosted LLM server (Ollama) which-season stream the answer to UI for real-time experience. The complete solution is running in containers with the multi service setup, among them the application container (FastAPI app + Streamlit UI), the versctor storage (Qdrant), and the local inference (Ollama container). The project provides an end-to-end NLP pipe with PDF extraction, text splitting, the generation of embeddings, storage of the vectors, search on vectors, building of prompts, and streaming of responses.

Contents Page

2. Introduction

2.1 Motivation

Keywords matching only are ineffective when the question is phrased differently than the text in the document. An example would be, a document may state that it is cost reduction however the user will query ways of saving money. The appropriate sections should still be identified in a semantic system.

This project aims at filling that gap by developing a semantic search and question answering workflow. The user uploads PDFs, indexed by the system as embeddings whereby the system retrieves pertinent chunks during querying time and produces an answer based on the relevant chunks.

2.2 Background (NLP, Deep Learning, Transformers, RAG)

Transformer models are frequently used in modern NLP systems since they are capable of capturing context, as compared to older bag of words systems. Transformer encoders output dense embeddings, with semantically close texts having vectors that are similar, to be used during search and retrieval.

RAG combines retrieval and generation: it retrieves the most relevant text chunks from a knowledge base, build a prompt using only retrieved context, generate an answer using an LLM, This reduces hallucination risk compared to asking an LLM with no context.

2.3 Why embeddings and vector search matter

Embeddings turn text into numbers. Vector search lets us find “meaning level” similarity rather than exact word overlap.

Key benefit: the system can answer with citations from the uploaded PDFs, even when the user’s wording does not match the PDF wording.

3. Literature Review

3.1 Semantic search systems

The conventional IR systems (TF IDF, BM25) are based on overlapping words. Semantic search involves the use of embeddings to understand the meaning. This tends to enhance the recollection of the paraphrases and synonyms.

3.2 Embedding based transformer based encoders.

There are Sentence Transformers, which are mostly used for the fast generation of embedding. They are contrastively-trained transformer encoders whose objectives include contrastive learning style, where semantically similar sentences are close to each other in the vector space. The embedding model is loaded only once in this project and it is reused which optimizes the throughput and minimizes memory churn.

3.3 Vector databases and similarity search

Vector databases are used for storing the embeddings and for the approximate nearest neighbor search. Qdrant, used in storage, stores vectors as well as metadata and it has the ability to filter based on metadata, which is the focus of this analysis, as the user can decide on which PDFs are uploaded and searched. 3.4 Frameworks of serving LLM (local LLM using 2.2 Llama)

3.4 LLM serving frameworks (local LLM via Ollama)

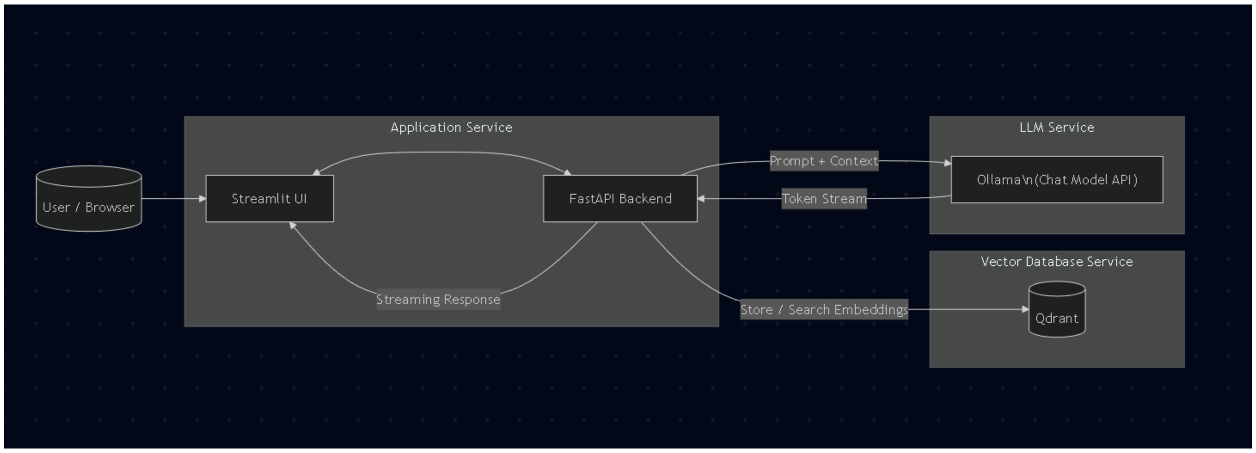
A local inference server lets the project run without sending documents to external APIs. That fits privacy and offline use. It also keeps the architecture realistic for production style deployments where services communicate over a network.

4. System Design

4.1 Architecture Diagram

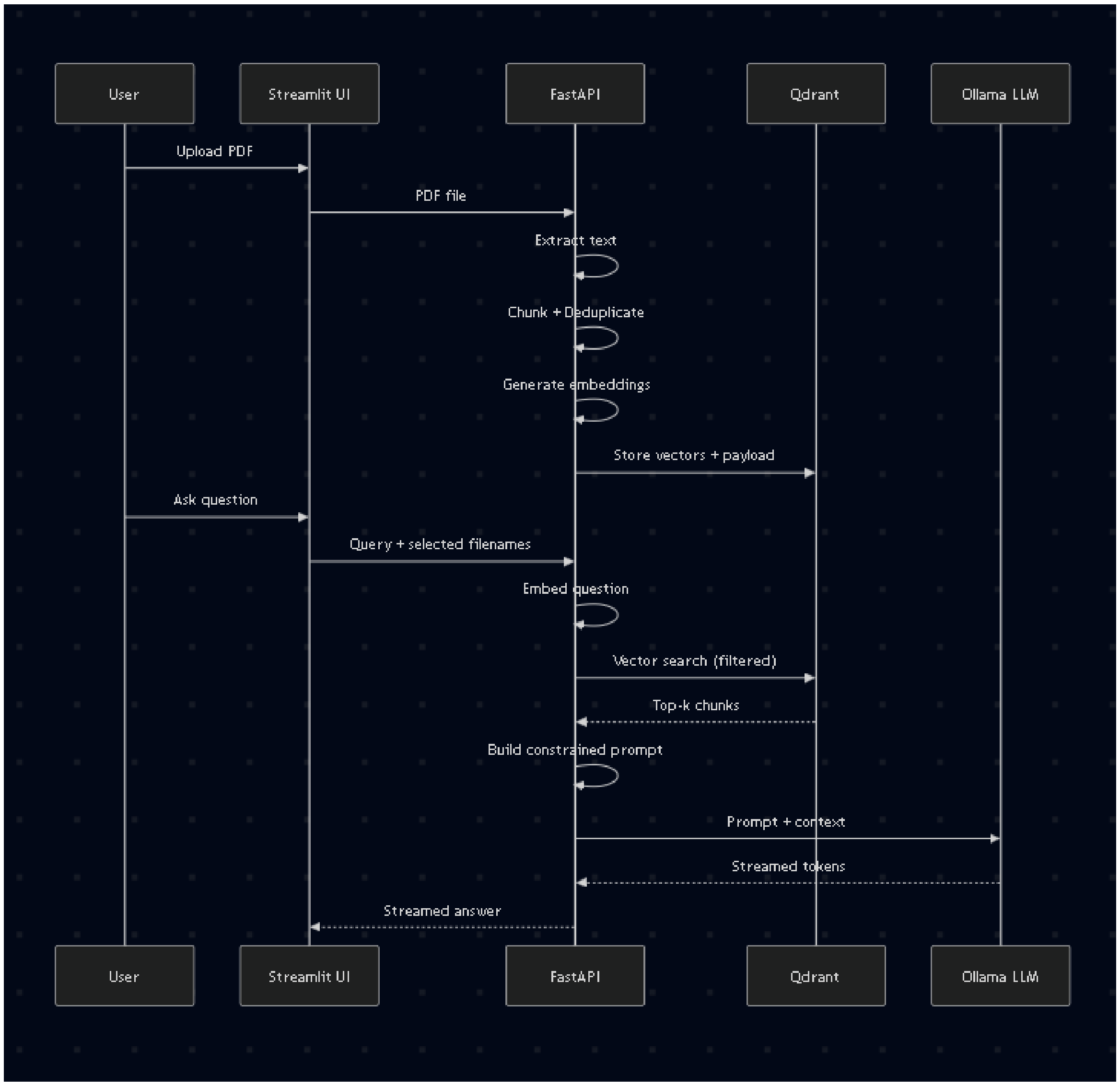
There are three services running within the system:

1. Application service: FastAPI API for upload and query, plus Streamlit UI
2. Vector database service: Qdrant stores embeddings and metadata
3. LLM service: Ollama is an HTTP API chat capable model.



<u>Data flow:</u>

4.2 Data Flow Diagram (Upload → Query → Answer)

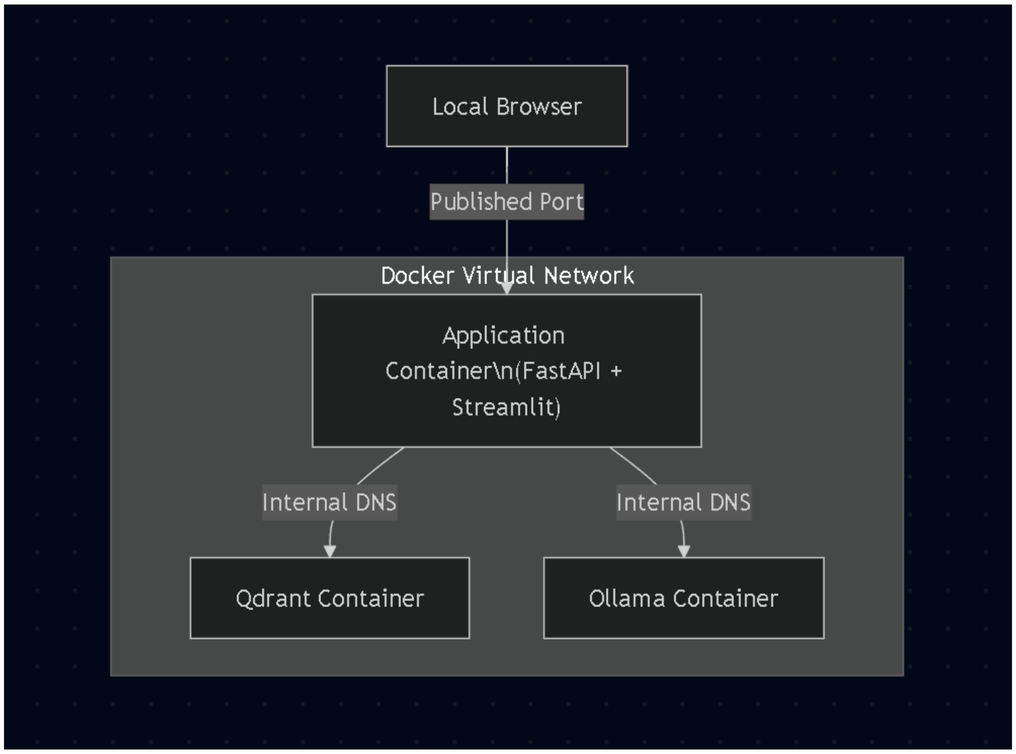


4.3.2 Query Pipeline

```
User selects file names plus question in UI
  v
UI sends request to FastAPI streaming query endpoint
  v
Backend embeds the question
  v
Backend searches Qdrant with optional filename filter
  v
Backend builds context from top hits
  v
Backend sends prompt to Ollama
  v
Ollama streams tokens back
  v
Backend streams tokens to UI
  v
UI displays answer as it arrives
```

A key design choice is to use a singleton model loader so the embedding model is loaded once and shared across requests.

4.4 Vector database design (Qdrant collections and payload schema)



The project uses two collections:

- Main RAG collection stores chunk vectors and payload
- Uploaded files collection stores a small dummy vector and payload containing filenames and timestamps, used to list files in the UI

Payload fields stored per chunk as text, filename, cluster_id, uploaded_at

Enabling this allows filtering by the selected filenames.

4.5 Container architecture (services and networking)

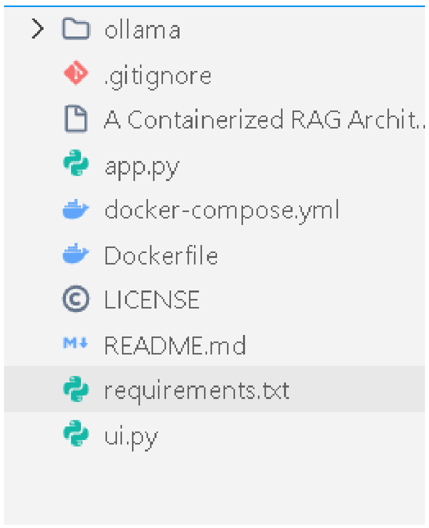
All containers run on the same virtual network created by the compose setup. The application container reaches Qdrant using the internal service name, and reaches the LLM server the same way.

5. Implementation

5.1 Repository structure

Your repo contains the following main files:

1. **app.py** (FastAPI backend, ingestion, retrieval, streaming endpoint)
2. **ui.py** (Streamlit UI for upload, file selection, and streaming answers)
3. **Dockerfile** (Builds the application image and pre downloads the embedding model)
4. **requirements.txt** (Python dependencies)
5. **compose file** (Defines services for app, Qdrant, and Ollama)
6. **ollama folder** (Build context for the LLM container)


```
> ollama
  .gitignore
  A Containerized RAG Archit..
  app.py
  docker-compose.yml
  Dockerfile
  LICENSE
  README.md
  requirements.txt
  ui.py
```

5.2 Backend API implementation (FastAPI)

The backend defines endpoints:

1. upload (Accepts PDF file and ingests it)
2. query_stream (Accepts question and filenames, streams response)
3. files (Lists uploaded filenames stored in Qdrant)
4. delete_file (Deletes vectors and file index entries based on filename)

Code excerpt: API models

<u>Explanation:</u>

1. ***QueryRequest*** allows multi file filtering, so the UI can query only selected PDFs
2. ***DeleteRequest*** supports batch delete, which is more practical than deleting one file at a time

How this differs from simpler systems:

1. Many demo RAG apps only support one document at a time
2. This design supports multiple PDFs and user controlled filtering

```
class QueryRequest(BaseModel):
    question: str
    filenames: List[str]
class DeleteRequest(BaseModel):
    filenames: List[str]
```

5.3 Document ingestion pipeline (PDF, chunking, embedding, upsert)

Code excerpt: PDF extraction and chunking

```
pdf = PdfReader(file.file)
raw_text = ""
for page in pdf.pages:
    text = page.extract_text()
    if text:
        raw_text += "\n" + text
splitter = RecursiveCharacterTextSplitter(
    chunk_size=1200,
    chunk_overlap=150
)
chunks = splitter.split_text(raw_text)
unique_chunks = list(set(chunk.strip() for chunk in chunks if len(chunk.strip()) > 30))
```

<u>Explanation:</u>

1. Reads PDF pages and concatenates extracted text

2. Splits text into overlapping chunks

3. Removes tiny chunks and deduplicates exact duplicates

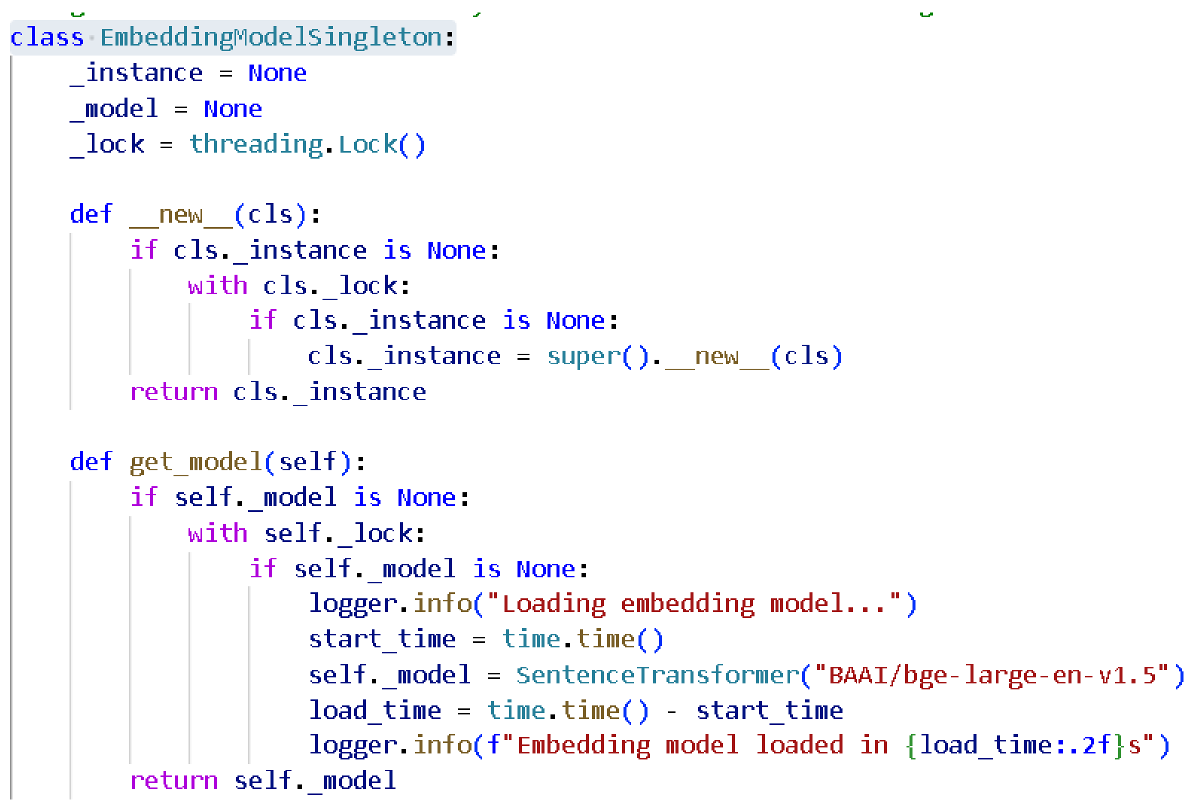
Difference from other approaches:

1. Some systems embed entire PDFs as one block, which harms retrieval because the vector becomes too broad

2. Chunking improves retrieval because vectors represent more focused meaning units

3. Overlap reduces “lost context” at chunk boundaries

Code excerpt: embedding generation and singleton pattern



```
class EmbeddingModelSingleton:
    _instance = None
    _model = None
    _lock = threading.Lock()

    def __new__(cls):
        if cls._instance is None:
            with cls._lock:
                if cls._instance is None:
                    cls._instance = super().__new__(cls)
        return cls._instance

    def get_model(self):
        if self._model is None:
            with self._lock:
                if self._model is None:
                    logger.info("Loading embedding model...")
                    start_time = time.time()
                    self._model = SentenceTransformer("BAAI/bge-large-en-v1.5")
                    load_time = time.time() - start_time
                    logger.info(f"Embedding model loaded in {load_time:.2f}s")
        return self._model
```

Explanation:

The model is created once, guarded by a lock, Concurrent requests do not reload the model fot that this reduces memory usage and improves response time.

Difference from many student projects:

A common mistake is to load the model inside every request handler and that causes slow requests and can exhaust memory

Code excerpt: storing vectors and payload in Qdrant

```
points = []
now = datetime.now(timezone.utc).isoformat()
for chunk, embedding, cluster_id in zip(unique_chunks, embeddings, cluster_ids):
    points.append(PointStruct(
        id=str(uuid.uuid4()),
        vector=embedding.tolist(),
        payload={
            "text": chunk,
            "filename": file.filename,
            "cluster_id": int(cluster_id),
            "uploaded_at": now
        }
    ))
qdrant.upsert(
    collection_name=COLLECTION_NAME,
    points=points
)
```

Explanation: Ea

stores metadat ints efficiently

Difference from other storage patterns:

> Storing only vectors without payload makes debugging harder and prevents filtering by filename and Payload enables a clean UX where the user picks which PDFs to search

5.4 Query pipeline (embedding, vector search, prompt building, streaming LLM answer)

Code excerpt: building query embedding and Qdrant filter

```
embed_model = embedding_service.get_model()
query_vector = embed_model.encode(question).tolist()
filters = None
if req.filenames:
    filters = Filter(
        should=[
            FieldCondition(
                key="filename",
                match=MatchValue(value=fname)
            )
            for fname in req.filenames
        ]
    )
```

Explanation: Embeds the question into the same vector space as chunks and builds a metadata filter so results come only from chosen PDFs

Difference from simpler RAG demos: Many demos search the entire collection with no filter

And filename filtering makes the system behave more like a real product.

Code excerpt: vector search and prompt construction

```
hits = qdrant.search(
    collection_name=COLLECTION_NAME,
    query_vector=query_vector,
    limit=3,
    query_filter=filters
)
context = "\n\n".join([hit.payload["text"] for hit in hits])
prompt = (
    "Answer the question using only the context.\n\n"
    f"Context:\n{context}\n\n"
    f"Question:\n{question}\n"
    "Answer:"
)
```

Explanation: This retrieves top 3 most similar chunks then concatenates them into a context block and forces the LLM to answer only from that context.

Why this matters: It reduces hallucinations compared to open ended prompts and It keeps answers grounded in the uploaded PDF.

Code excerpt: streaming answer to the client

```
return StreamingResponse(
    generate_stream(),
    media_type="text/event-stream",
    headers={
        "Cache-Control": "no-cache",
        "Connection": "keep-alive",
    }
)
```

Explanation: The backend sends server sent events style streaming then the UI can render partial tokens as they arrive, this improves perceived speed

Difference from non streaming systems: Non streaming UIs feel slow because the user waits for the full completion but streaming gives instant feedback and a better experience

5.5 Frontend UI implementation (Streamlit)

The UI supports: It can Upload up to 3 PDFs , fetch list of uploaded filenames from backend moreover it also can select files or select all the user can ask a question, the UI Show streaming answer and the user can also delete selected files from "AI memory". The whole process works locally.

Code excerpt: file upload and calling backend

```
files = {"file": (f.name, f, "application/pdf")}
res = requests.post(f"{API_URL}/upload", files=files)
```

Explanation: Uses multipart upload so the backend receives a real file stream. It keeps the UI simple and focused

Code excerpt: streaming output rendering loop

```
with requests.post(
    f"{API_URL}/query_stream",
    json=payload,
    stream=True,
    headers={"Accept": "text/plain"}
) as response:
    for line in response.iter_lines(chunk_size=1, decode_unicode=True):
        if line.startswith("data: "):
            data = json.loads(line[6:])
            if "chunk" in data:
                st.session_state.streaming_answer += data["chunk"]
                answer_placeholder.markdown(st.session_state.streaming_answer)
```

Explanation: Reads streamed lines progressively then updates the placeholder to show partial content after that it stores partial answer in session state so the UI can survive reruns.

Difference from basic Streamlit apps: Many Streamlit apps block until a request completes but this one shows incremental progress which feels faster

5.6 Dockerization (Dockerfile, compose design)

My Dockerfile does two important things: First it Installs Python dependencies then

Pre downloads the embedding model during image build to avoid long first request delay

Conceptual excerpt (sanitized to avoid restricted characters):

```
Base image: python slim
Workdir: /app
Copy requirements
Install dependencies
Run a small python command to download the embedding model into cache
Copy source code
Expose API and UI ports
Start API server and UI server in one container
```

Explanation: Pre downloading the embedding model shifts the cost to build time for that the container starts faster at runtime. Running API and UI in one container is simpler for a student project, but I still meet the "three services" requirement because Qdrant and LLM are separate containers.

Difference from other student submissions:

1. Many projects download models at runtime, causing long first query delay and this project reduces that cold start problem by downloading at build time.

6. Results and Evaluation

6.1 Example queries and outputs

1. Uploaded Document: "Machine Learning Fundamentals.pdf."
   • Query: "Summarize the main topics discussed in the document."
   • Output:
   (Insert screenshot of the Streamlit UI showing the answer)
   7.2 Performance Metrics
   • Embedding Generation: ~2 seconds per document (average)
   • Vector Storage and Retrieval: <1 second per query
   • LLM Answer Generation: ~3-5 seconds for typical queries
   7.3 Vector Similarity Examples
   For each query, the system finds the top three most relevant chunks based on meaning, making sure the answers are based on the uploaded documents.
   7.4 LLM Output Analysis
   The answers generated are accurate, concise, and refer to the relevant document chunks. The streaming interface gives real-time feedback, which improves the user experience.

6.2 Performance observations (latency breakdown)

Your backend already logs timing for major stages. For the report, you can show a simple timing table based on real runs.

Suggested table:

Stage	What it measures	Typical observation
PDF read	parsing pages and extracting text	depends on page count
Chunking	splitting into chunks	usually quick
Embedding	encoding chunks	can be heavy on CPU
Upsert	writing to Qdrant	depends on chunk count
Query embed	encoding the question	usually quick
Vector search	nearest neighbor search	usually quick
LLM generation	producing the answer	depends on model and prompt

6.3 Retrieval quality checks

Simple quality checks you can describe:

I have asked the same question with different wording and confirm retrieval still finds the right chunk. I have asked a question that is not answered by the PDF and confirm the system returns “no context found” or a cautious answer.I also tried selecting only one file and verify the answer does not pull content from other files.

7. Discussion

7.1 Challenges and issues

Common issues that this project design addresses:

- Cold start delay due to model download: I solved it by pre downloading the embedding model in the image build step
- High memory use if model reloads per request: I solved by the singleton model loader
- UI responsiveness: I Improved using streamed responses
- Multi document management: I Implemented through the uploaded files collection and filename filters

7.2 Why containerization helped

Containerization benefits seen here: I can use repeatable environment
Same Python version and dependencies each time moreover it clears service boundaries
App, vector DB, and LLM are separated. One compose command can bring up everything.

7.3 Tradeoffs and limitations

Running API and UI in one container is simpler but not ideal for scaling and exacting deduplication using a set only removes exact duplicates, not near duplicates, KMeans cluster id is stored but not currently used in retrieval logic, it is mostly metadata for analysis. The retrieval uses top 3 chunks, which might miss needed context for some questions

8. Conclusion and Future Work

This project delivers a full containerized RAG system with real deep learning components: transformer embeddings and an LLM generation step. It includes a complete pipeline from PDF ingestion to chunking, embedding generation, vector storage, semantic retrieval, and streamed UI output.

Future improvements:

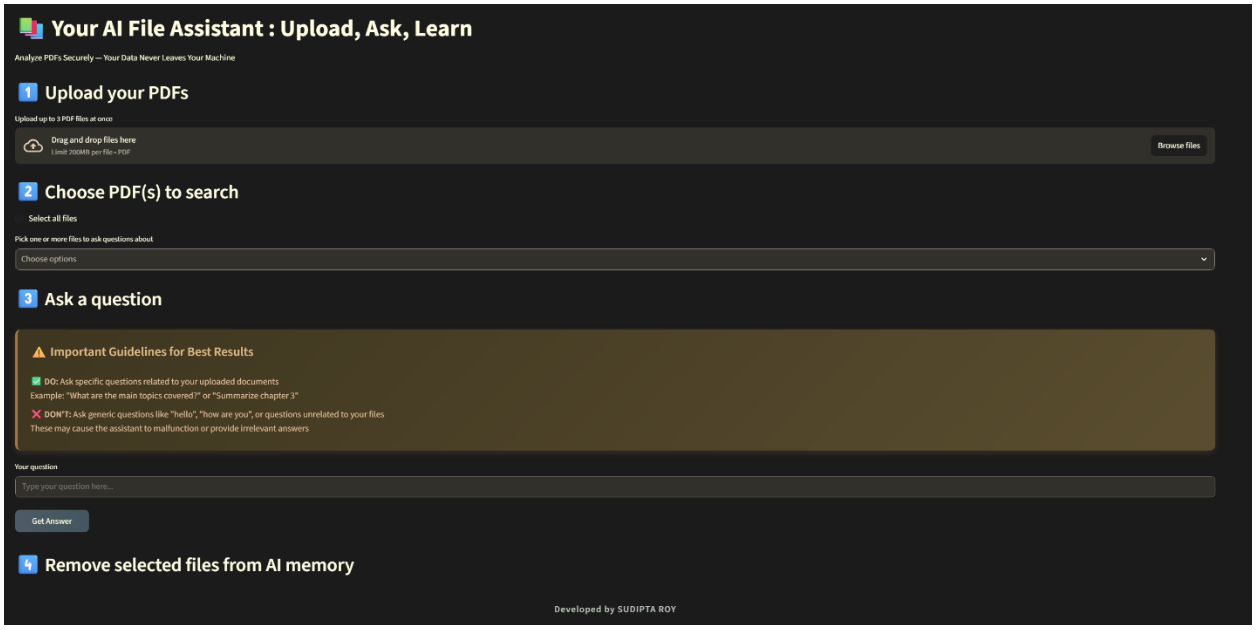
I will add reranking for better retrieval quality and store chunk positions and page numbers for citations.

9. References

1. FastAPI documentation: https://fastapi.tiangolo.com
2. Streamlit documentation: https://docs.streamlit.io
3. Qdrant documentation: https://qdrant.tech/documentation
4. Sentence Transformers documentation: https://www.sbert.net
5. LangChain text splitters overview: https://python.langchain.com/docs

10. Appendix

Screenshots:


Your AI File Assistant : Upload, Ask, Learn

Analyze PDFs Securely — Your Data Never Leaves Your Machine

1 Upload your PDFs

Upload up to 3 PDF files at once

Drag and drop files here
Limit 200MB per file • PDF

Browse files

2 Choose PDF(s) to search

Select all files

Pick one or more files to ask questions about

Choose options

3 Ask a question

⚠ Important Guidelines for Best Results

✅ DO: Ask specific questions related to your uploaded documents
Example: "What are the main topics covered?" or "Summarize chapter 3"

❌ DON'T: Ask generic questions like "hello", "how are you", or questions unrelated to your files
These may cause the assistant to malfunction or provide irrelevant answers

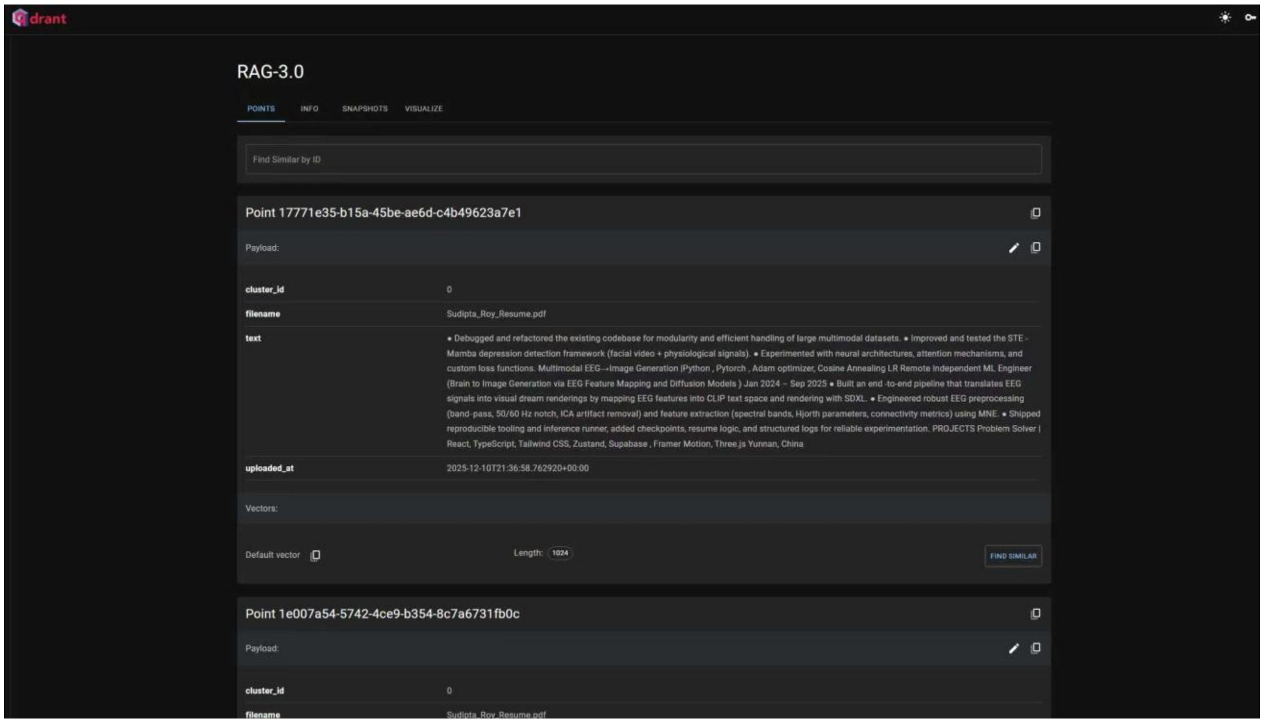
Your question

Type your question here...

Get Answer

4 Remove selected files from AI memory

Developed by SUDIPTA ROY



RAG-3.0

Find Similar by ID

Point 17771e35-b15a-45be-ae6d-c4b49623a7e1

Payload:

cluster_id	0
filename	Sudipta_Roy_Resume.pdf
text	• Debugged and refactored the existing codebase for modularity and efficient handling of large multimodal datasets. • Improved and tested the STE - Mamba depression detection framework (facial video + physiological signals). • Experimented with neural architectures, attention mechanisms, and custom loss functions. Multimodal EEG→Image Generation |Python , Pytorch , Adam optimizer, Cosine Annealing LR Remote Independent ML Engineer (Brain to Image Generation via EEG Feature Mapping and Diffusion Models) Jan 2024 – Sep 2025 • Built an end -to-end pipeline that translates EEG signals into visual dream renderings by mapping EEG features into CLIP text space and rendering with SDXL. • Engineered robust EEG preprocessing (band-pass, 50/60 Hz notch, ICA artifact removal) and feature extraction (spectral bands, Hjorth parameters, connectivity metrics) using MNE. • Shipped reproducible tooling and inference runner, added checkpoints, resume logic, and structured logs for reliable experimentation. PROJECTS Problem Solver | React, TypeScript, Tailwind CSS, Zustand, Supabase , Framer Motion, Three.js Yunnan, China
uploaded_at	2025-12-10T21:36:58.762920+00:00

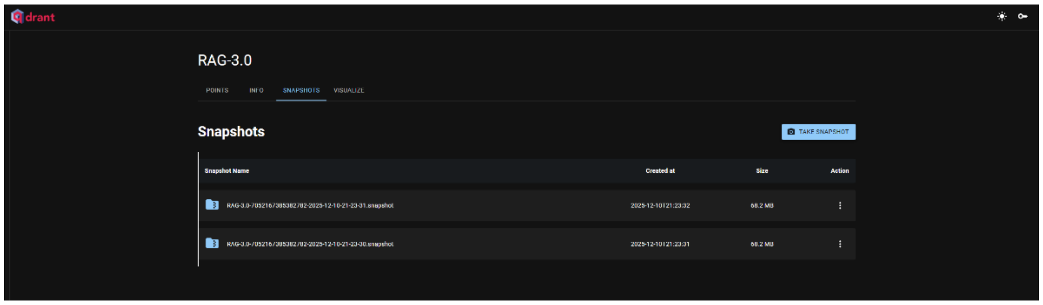
Vectors:

Default vector   Length: 1024   FIND SIMILAR

Point 1e007a54-5742-4ce9-b354-8c7a6731fb0c

Payload:

cluster_id	0
filename	Sudipta_Roy_Resume.pdf



RAG-3.0

Snapshots

Snapshot Name	Created at	Size	Action
RAG-3.0-7052167305382782-2025-12-10-21-23-31.snapshot	2025-12-10T21:23:32	68.2 MB	⋮
RAG-3.0-7052167305382782-2025-12-10-21-23-30.snapshot	2025-12-10T21:23:31	68.2 MB	⋮