Project – High Level Design

on

Chat With Retail Document

Course Name: GenAI

Institution Name: Medicaps University – Datagami Skill Based Course

Submitted by:

Sr no	Student Name	Enrolment Number
1.	Amit Patidar	EN22CS301114
2.	Aniket Kushwah	EN22CS301124
3.	Anuj Singh Rathore	EN22CS301166
4.	Anurag Didolkar	EN22CS301269
5.	Arsh Patidar	EN23CS3L1204
6.	Avani Gupta	EN22CS301236

Group Name: Group 09D2

Project Number: GAI-21

Industry Mentor Name:

University Mentor Name: Hemlata Patel

Academic Year: 2026

Table of Contents

1. Introduction

Retail Doc Intel is a production-grade Retrieval-Augmented Generation (RAG) chatbot designed for the retail domain. The system allows retail professionals to upload PDF documents — such as weekly sales reports, inventory sheets, and supplier invoices — and query them using natural language. Responses are strictly grounded in the uploaded document content, minimizing hallucinations.

system is fully deployed with a React frontend on Vercel and a FastAPI backend on Railway, using Google Gemini 2.5 Flash as the LLM, ChromaDB Cloud as the vector store, and Firebase for authentication and chat history persistence.

Problem Statement

Architect a complete Retrieval-Augmented Generation (RAG) system for Retail document intelligence. The solution enables users to upload professional documents (PDF reports, manuals) which are then processed through a pipeline of text chunking and vector embedding. By leveraging a high-performance Vector Database and LLM APIs, the system facilitates accurate, context-aware Q&A sessions, ensuring that responses are grounded strictly in the provided domain-specific content.

1.1 Scope of the Document

This High-Level Design document covers:

- Overall system architecture and technology stack
- Major components and their responsibilities
- Information and process flows
- API design and endpoint catalogue
- Data model and storage strategy
- Authentication, session management, and CORS strategy
- Deployment architecture on Railway and Vercel
- Non-functional requirements: security and performance
- Known limitations and planned future enhancements

This document does not cover detailed low-level implementation code.

1.2 Intended Audience

This document is intended for:

- Developers and solution architects
- DevOps and deployment engineers
- University mentors and project review committee
- Industry mentors and technical stakeholders

1.3 System Overview

The system operates across three primary workflows:

Workflow 1 — Document Ingestion

1. User uploads a PDF document via the React frontend
2. Backend extracts text using PyPDF
3. Text is split into chunks (1000 characters, 200 overlap) using LangChain RecursiveTextSplitter
4. Each chunk is embedded using Google Gemini Embedding Model (gemini-embedding-001)
5. Embeddings are stored in ChromaDB Cloud tagged with the user's session_id

Workflow 2 — Query & Response

6. User types a natural language question in the chat interface
7. Query is embedded using the same Gemini Embedding model
8. Semantic similarity search is performed on ChromaDB, filtered by session_id (top-3 chunks)
9. Retrieved chunks are injected into a RAG prompt template
10. Prompt is sent to Gemini 2.5 Flash which generates a grounded answer
11. Response and source filenames are returned to the frontend

Workflow 3 — Authentication

12. User signs up or logs in via Firebase Authentication
13. Firebase issues a JWT token to the frontend
14. All API requests include the JWT in the Authorization header
15. Backend middleware verifies the token on every protected request
16. Guest mode is supported — full functionality without account creation

Architecture characteristics: modular, API-driven, cloud-native, session-isolated, scalable.

2. System Design

2.1 Application Design

The system follows a five-layer architecture where each layer is loosely coupled to allow independent scaling and maintainability.

Layer	Name	Description
1	Presentation Layer	React 18 + Vite frontend served via Vercel CDN
2	Application Layer	FastAPI Python backend deployed on Railway
3	Processing Layer	Embedding generation, chunking, and RAG pipeline
4	Data Layer	ChromaDB Cloud (vectors) + Firestore (chat history)
5	LLM Integration	Google Gemini 2.5 Flash via LangChain ChatGoogleGenerativeAI

Frontend — React 18 + Vite + Tailwind CSS (Vercel)

- Dark-themed chat UI with drag-and-drop PDF upload (react-dropzone)
- Firebase Auth SDK handles login, logout, and guest mode
- Axios HTTP client with automatic JWT token injection via request interceptor
- react-markdown renders AI responses with code and table support
- Session-based document isolation — each chat session gets a unique UUID
- Local session history in React state; persistent history via Firestore for authenticated users
- Upload progress bar via Axios onUploadProgress callback

Backend — FastAPI Python 3.11 (Railway)

- Modular structure: routes/, services/, middleware/, core/
- Firebase JWT middleware — optional auth allows guest access without breaking security
- Upload router: PDF extraction, chunking, embedding, ChromaDB storage
- Chat router: query embedding, retrieval, RAG prompt assembly, Gemini call
- Chat history service: Firestore read/write/delete per authenticated user
- CORS configured with known production origins + allow_origin_regex for all Vercel preview URLs

2.2 Process Flow

A. Document Upload Flow

1. User selects PDF via drag-and-drop or file picker in the frontend
2. Frontend sends multipart/form-data POST to /api/upload with JWT header
3. Auth middleware verifies token (or allows guest)
4. Document Processor extracts text from PDF using PyPDF
5. LangChain RecursiveTextSplitter splits text into overlapping chunks
6. Each chunk embedded via Gemini Embedding API (gemini-embedding-001)
7. Embeddings stored in ChromaDB Cloud with metadata: {session_id, doc_name, chunk_id}
8. Success response with document name returned to frontend

B. Query Handling Flow

9. User types a question and submits to POST /api/chat with session_id and JWT
10. Auth middleware verifies token (or allows guest)
11. Query converted to embedding via Gemini Embedding API
12. ChromaDB similarity search: filter by session_id, return top-3 chunks
13. If no chunks found, return graceful fallback message
14. RAG prompt assembled: system instructions + retrieved context + user query
15. Prompt sent to Gemini 2.5 Flash (1M context window) via LangChain
16. Gemini returns grounded response
17. Source document names deduplicated and returned alongside answer
18. If authenticated, message saved to Firestore chat history

2.3 Information Flow

```
User Browser
  |— POST /api/upload (PDF + JWT) ————————▶ FastAPI Backend
  |                                  |
  |                    PyPDF → Chunker → Gemini Embed
  |                                  |
  |                    ChromaDB Cloud (store vectors)
  |
  |— POST /api/chat (query + session_id + JWT) —▶ FastAPI Backend
  |                                  |
  |                   Gemini Embed (query vector)
  |                                  |
  |                  ChromaDB similarity search
  |                   (filter: session_id, top-3)
  |                                  |
  |               RAG Prompt → Gemini 2.5 Flash
  |                                  |
  |◀— AI response + source filenames ————————┘
  |
  |— Firebase Auth SDK ———————————————▶ Firebase Auth
     (get JWT token)              (verify JWT in middleware)
```

2.4 Components Design

Component	Technology	Responsibility
Document Processor	PyPDF + LangChain RecursiveTextSplitter	Extract PDF text, split into 1000-char chunks with 200-char overlap
Embedding Generator	gemini-embedding-001 via LangChain	Convert text chunks and queries into 768-dim vector embeddings
Vector Store	ChromaDB Cloud	Store and retrieve embeddings; session_id filter isolates user docs
RAG Pipeline	chat_service.py	Orchestrate retrieval, prompt assembly, LLM call, response formatting
LLM Service	Gemini 2.5 Flash (ChatGoogleGenerativeAI)	Generate grounded natural language answers from context + query
Auth Middleware	Firebase Admin SDK	Verify JWT tokens; optional auth supports guest mode
Chat History Service	Google Firestore	Persist and retrieve per-user chat messages across sessions
API Layer	FastAPI with /api prefix	REST endpoints for upload, chat, auth, and history management
Frontend State	React Context (AuthContext, ChatContext)	Manage auth state, session history, messages, and upload state

2.5 Key Design Considerations

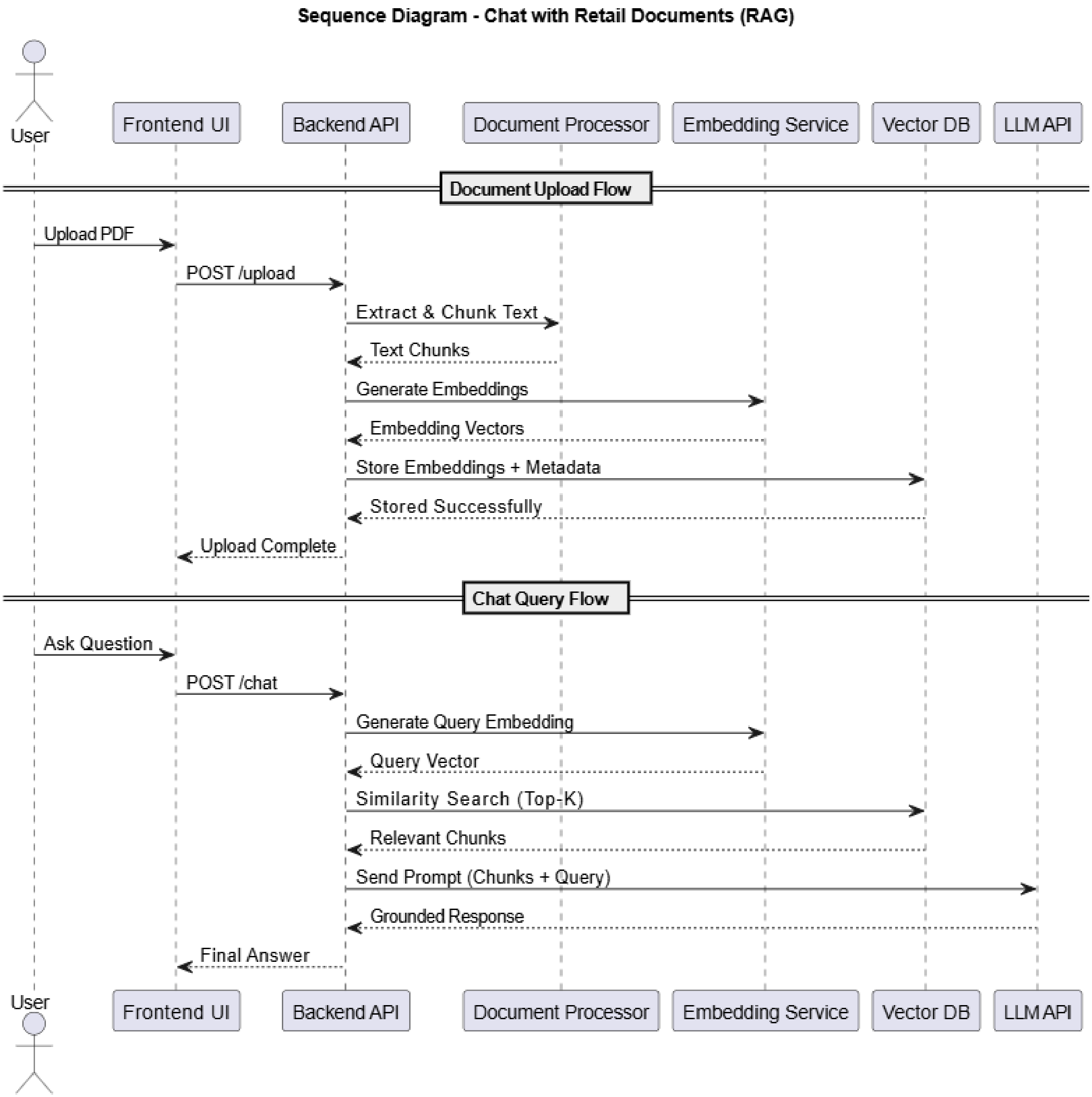
- Session isolation via session_id metadata filter in ChromaDB — one user's documents never mix with another's
- Optional authentication middleware — guest users get full RAG functionality; authenticated users additionally get persistent history
- Strict RAG prompting — Gemini is instructed to answer only from retrieved context, not general knowledge
- Dynamic CORS via allow_origin_regex — automatically supports all Vercel preview deployment URLs without manual reconfiguration
- Chunk size of 1000 characters with 200 overlap balances context quality against ChromaDB token limits
- Top-3 chunk retrieval per query minimizes noise while fitting comfortably within Gemini's context window
- Firebase credentials stored as JSON environment variable on Railway — no credential files in the repository

2.6 API Catalogue

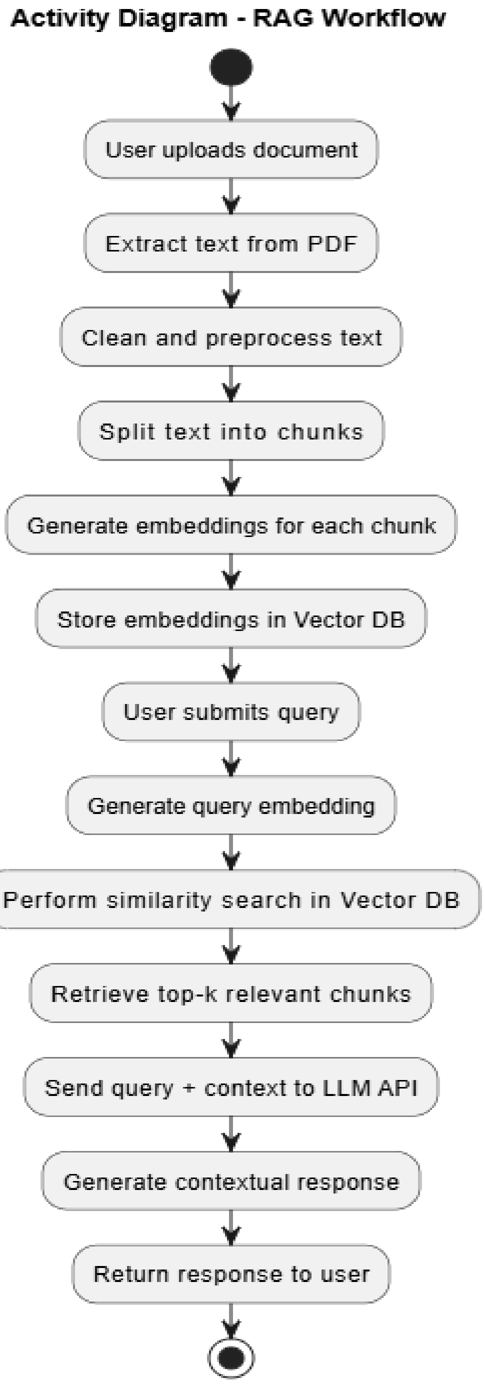
Method	Endpoint	Description	Auth	Response
GET	/api/auth/me	Return authenticated user profile and email	Required	200 user object
POST	/api/upload	Upload PDF, extract, chunk, embed, store in ChromaDB	Optional	200 + filename
POST	/api/chat	Send query, retrieve chunks, generate AI response	Optional	200 + answer + sources
GET	/api/chat/history	Fetch authenticated user's full message history	Optional	200 messages[ ]
DELETE	/api/chat/history	Clear all chat messages for authenticated user	Optional	200 success
GET	/health	Railway liveness probe / health check endpoint	None	200 OK

UML Diagrams:-

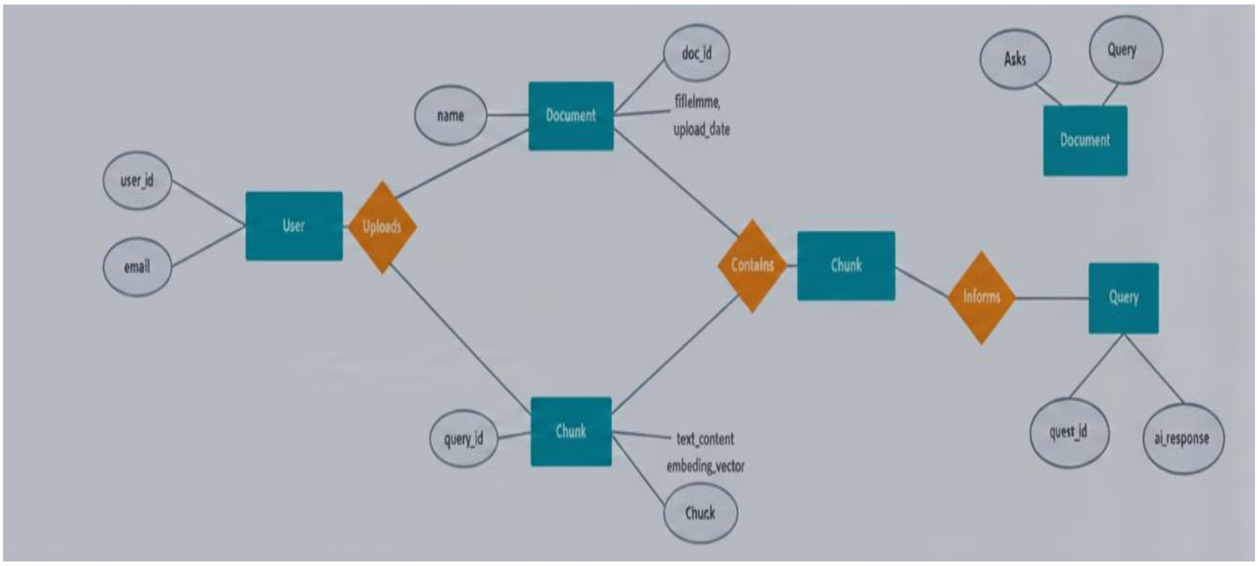
Sequence Diagram



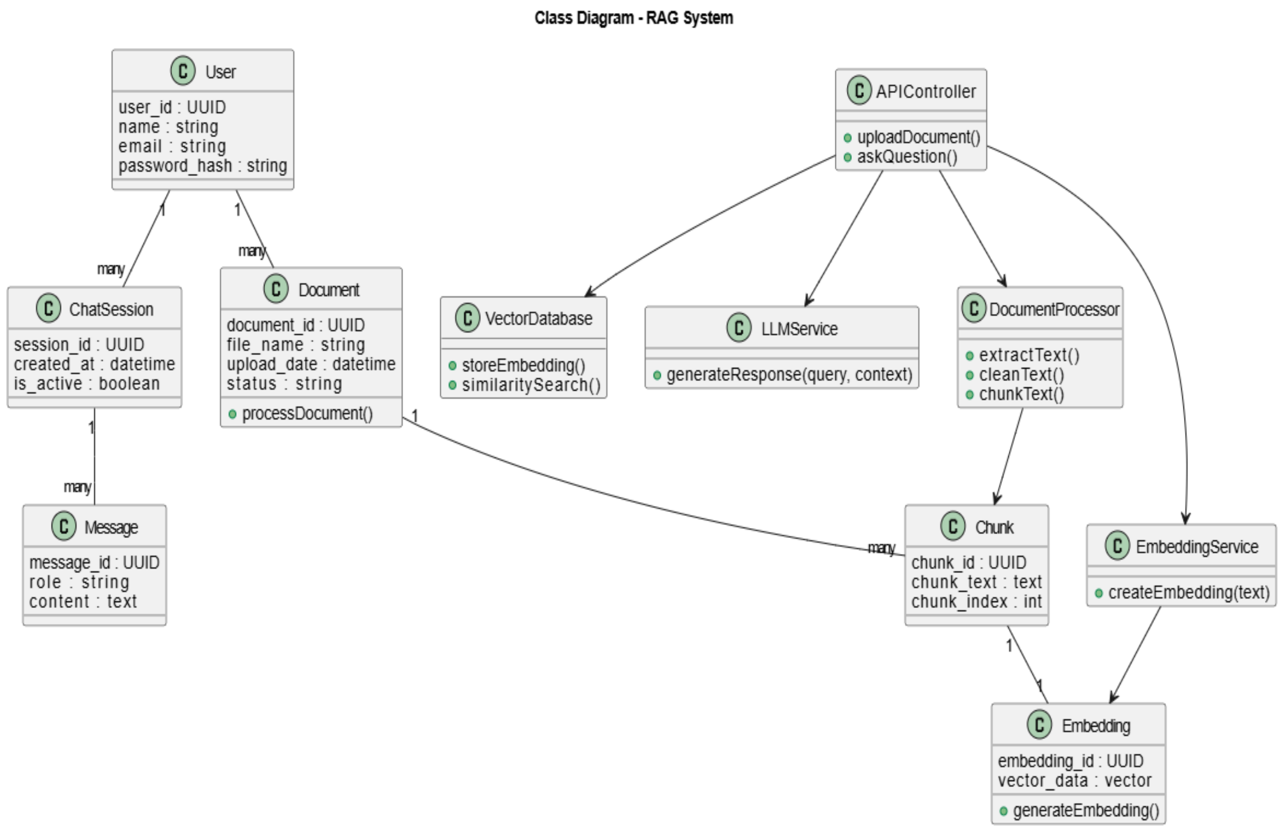
Activity Diagram



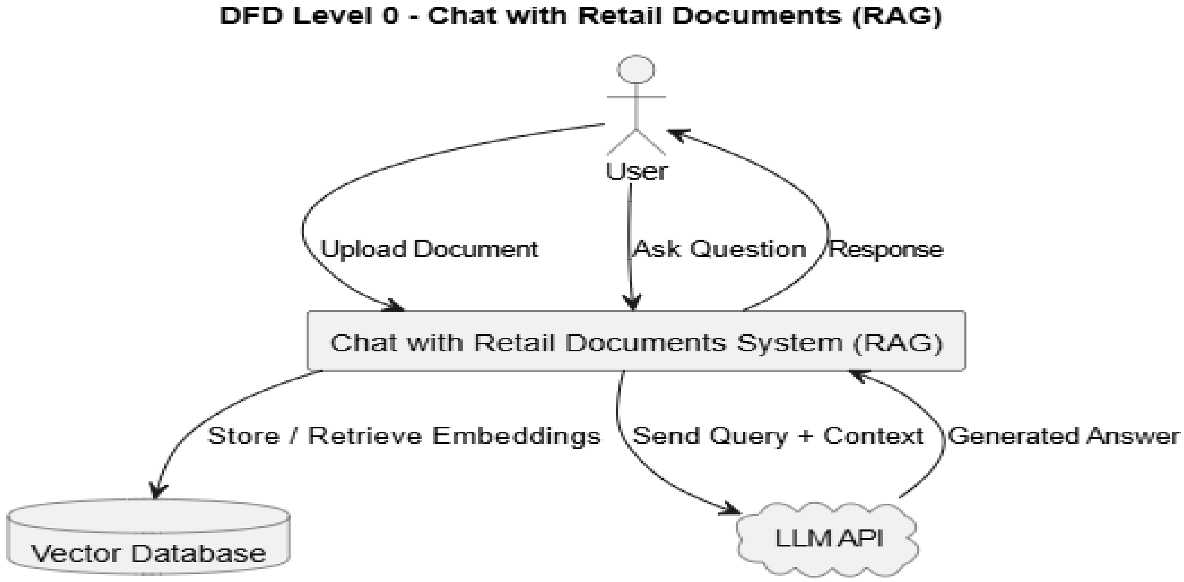
Entity Relationship Diagram



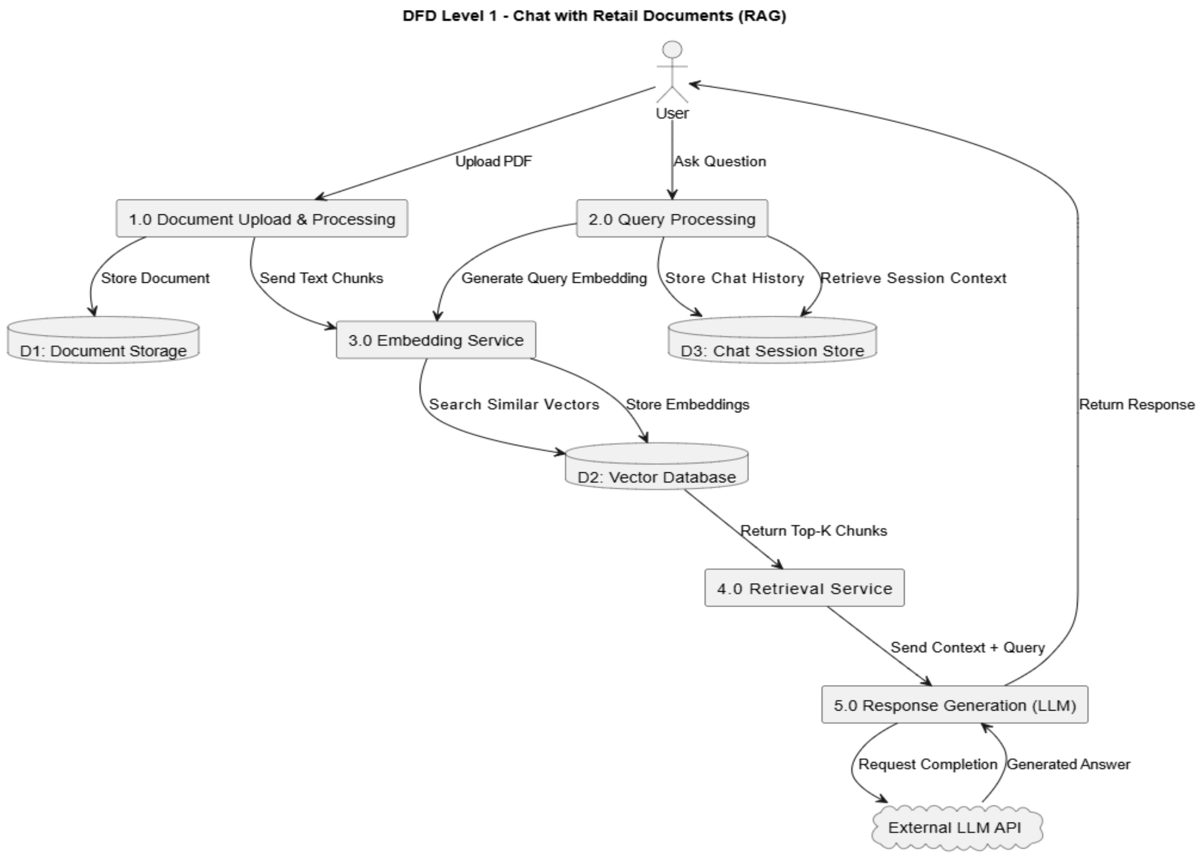
Class Diagram



Data Flow Diagram - Level 0



Data Flow Diagram – Level 1



3. Data Design

3.1 Data Model

ChromaDB — Vector Store (per chunk)

Field	Type	Description
chunk_id	String (UUID)	Unique identifier for each text chunk
session_id	String (UUID)	Links chunk to user's chat session — used as retrieval filter
document_name	String	Original uploaded PDF filename — returned as source citation
text_content	String	Raw text of the chunk (up to 1000 characters)
embedding	Float[768]	768-dimensional vector from gemini-embedding-001

Firestore — Chat History (per message)

Field	Type	Description
user_id	String (Firebase UID)	Firebase user identifier — scopes history to the user
session_id	String (UUID)	Links message to a specific chat session
role	Enum (user/assistant)	Identifies whether message is from user or AI
content	String	Full text content of the message
timestamp	Timestamp	Server-side creation time for ordering messages

Frontend State — Session (in-memory)

Field	Type		Description
session_id	String (UUID)		Generated on New Chat; passed with every API request
messages	Array		Local array of {role, content} objects for current chat
documents	Array		List of uploaded document names in current session
pastSessions	Array		List of previous sessions for sidebar history display

3.2 Data Access Mechanism

- Embeddings stored and queried via ChromaDB Cloud REST API through the LangChain-Chroma integration
- Retrieval uses cosine similarity with a metadata filter: where({session_id: <id>}), returning top-3 results
- Chat history read and written via Firebase Admin SDK using Firestore collection queries filtered by user_id
- All database access occurs only from the FastAPI backend — frontend has no direct database access

3.3 Data Retention Policies

- ChromaDB embeddings persist indefinitely per session until manually cleared (document deletion not yet implemented in UI)
- Firestore chat history retained until user explicitly calls DELETE /api/chat/history
- Guest user data (session vectors in ChromaDB) has no user_id association and may be cleaned up by TTL policy in future
- No PII stored beyond Firebase-managed user email and UID

4. Interfaces

Interface	Protocol	Description
Frontend ↔ Backend	HTTPS REST (Axios + JWT)	All frontend-to-backend communication via JSON REST APIs
Backend ↔ ChromaDB Cloud	HTTPS (LangChain-Chroma SDK)	Vector storage and similarity search via ChromaDB Cloud API
Backend ↔ Gemini API	HTTPS (LangChain Google GenAI)	Embedding generation and LLM response via Google AI API
Backend ↔ Firebase Admin	HTTPS (Firebase Admin SDK)	JWT token verification and Firestore read/write
Frontend ↔ Firebase Auth	HTTPS (Firebase JS SDK)	User authentication, token acquisition, and state management

5. State and Session Management

- Each new chat creates a UUID session_id generated in the React frontend
- session_id is sent with every /api/upload and /api/chat request to scope ChromaDB vectors to the user's session
- Authenticated users: full message history persisted in Firestore, restorable across devices and browser sessions
- Guest users: session exists only in React state (in-memory); history lost on page refresh
- Sidebar displays past sessions and allows one-click restoration of prior chat context
- Firebase JWT tokens are short-lived; the Auth SDK handles automatic token refresh transparently
- No server-side session timeout is currently implemented — marked as future enhancement

6. Caching & Future Optimizations

Caching is not implemented in the current version v1.0. The following optimizations are planned:

- Query result caching — cache top-k retrieval results for repeated identical queries to reduce ChromaDB calls
- Embedding caching — cache embedding vectors for frequently uploaded document chunks to reduce Gemini API costs
- Response caching — cache LLM responses for identical (context, query) pairs to reduce latency
- CDN caching — static frontend assets are already cached via Vercel's global CDN

Current average response time is under 3 seconds without caching, which meets the target for v1.0.

7. Deployment Architecture

Service	Platform	Details
Frontend	Vercel	Auto-deploys on git push to main; preview URL per PR; SPA fallback via vercel.json
Backend	Railway	Python 3.11 auto-detected; start command via Procfile; root directory: /backend
Vector Database	ChromaDB Cloud	Managed cloud instance; accessed via API key, tenant, and database credentials
Authentication	Firebase Auth	Google-managed; JWT tokens issued to frontend and verified by backend middleware
Chat History	Firestore	Google-managed NoSQL; accessed via Firebase Admin SDK with service account JSON
LLM + Embeddings	Google AI API	Gemini 2.5 Flash (responses) + gemini-embedding-001 (embeddings)
CI/CD	GitHub	Both Railway and Vercel auto-deploy on every push to main branch

CORS Strategy

A key deployment challenge was handling CORS for Vercel preview deployments. Each deployment generates a unique URL. The backend uses FastAPI's allow_origin_regex to allow all preview URLs automatically:

```
allow_origins=["https://retail-document-chatbot.vercel.app"]  # Production
allow_origin_regex=r"https://retail-document-chatbot(-[a-z0-9]+)*-awwniket47s-projects\.vercel\.app"
```

Environment Variables

Sensitive credentials are never stored in the repository. All secrets are configured as environment variables on Railway (backend) and Vercel (frontend). The Firebase service account JSON is passed as a single-line string in FIREBASE_CREDENTIALS_JSON.

8. Non-Functional Requirements

8.1 Security Aspects

Security Requirement	Implementation
JWT Authentication	Firebase Auth issues tokens; FastAPI middleware verifies on every protected request
Guest Mode Security	Optional auth middleware allows guest access without exposing protected user data
Credential Management	All API keys and secrets stored as environment variables; never committed to Git
Firebase Credentials	Service account JSON passed as env var on Railway; excluded from .gitignore
CORS Policy	Restricted to known production + preview Vercel origins; all other origins blocked
Data Encryption in Transit	All communication over HTTPS (Vercel, Railway, Firebase, ChromaDB, Gemini)
Session Isolation	ChromaDB metadata filter (session_id) prevents cross-user document access
File Upload Validation	Backend validates uploaded file is a valid PDF before processing

8.2 Performance Aspects

Metric	Target	Achieved
End-to-end query response time	< 3 seconds	~2.5–3s observed in production
PDF upload + indexing time	< 10 seconds	~5–8s for typical 5–10 page PDF
Frontend initial load time	< 2 seconds	< 1.5s via Vercel CDN
ChromaDB retrieval latency	< 500ms	~200–400ms for top-3 search
Concurrent users	10+ concurrent	Railway hobby tier supports this
Availability	99%+	Managed platforms: Vercel + Railway

9. Known Limitations & Future Enhancements

The current version has the following known limitations that are planned for future releases:

- No server-side session timeout is currently implemented.
- No caching layer for repeated queries; future versions may add Redis or similar.
- ChromaDB embeddings persist indefinitely per session; manual cleanup is required.
- Future enhancements include multi-document comparison, export functionality, and advanced analytics.

10. References

The system design is based on:

- Retrieval-Augmented Generation (RAG) Architecture Concepts
- Vector Database Documentation (FAISS / Pinecone / Chroma)
- LLM API Documentation
- Python Backend Framework Documentation
- REST API Design Standards