RAG API

Structure & Documentation

Technical Specification

December 1, 2025

Contents

1 Document Upload API **2**
1.1 Pydantic Models . . . 2
1.1.1 Upload Request Model . . . 2
1.1.2 Upload Response Model . . . 2

2 RAG Search API **3**
2.1 Pydantic Models . . . 3
2.1.1 Search Request Model . . . 3
2.1.2 Search Response Model . . . 3

3 Chat / Session API **4**
3.1 Pydantic Models . . . 4
3.1.1 Chat Request Model . . . 4
3.1.2 Chat Response Model . . . 4

4 RAG Architecture **5**
4.1 LangChain Framework . . . 5
4.2 Vector Databases . . . 5
4.3 Session History Storage . . . 6

5 Implementation Requirements **7**

1 Document Upload API

1.1 Pydantic Models

1.1.1 Upload Request Model

DocumentMetadata

```python
class DocumentMetadata(BaseModel):
    doc_id: str
    chapter: Optional[str] = None
    section: Optional[str] = None
    tags: Optional[List[str]] = None
```

UploadRequest

```python
class UploadRequest(BaseModel):
    filename: str
    content: str
    metadata: Optional[DocumentMetadata] = None
    chunk_size: int = 500
    chunk_overlap: int = 50
```

Purpose: The `UploadRequest` model defines everything needed to upload a document into the RAG pipeline, including the filename, raw text content, optional metadata, and chunking parameters (`chunk_size`, `chunk_overlap`), enabling the system to properly chunk, embed, and ingest the document into the vector database for retrieval.

1.1.2 Upload Response Model

UploadResponse

```python
class UploadResponse(BaseModel):
    status: str
    total_chunks: int
    doc_id: str
```

Purpose: The `UploadResponse` model provides confirmation of a successful document upload by returning ingestion status, total number of chunks created, and the assigned `doc_id`, allowing clients to verify that the document was processed and stored correctly in the RAG system.

2 RAG Search API

2.1 Pydantic Models

2.1.1 Search Request Model

RAGSearchRequest

```python
class RAGSearchRequest(BaseModel):
    query: str
    top_k: int = 5
    include_sources: bool = True
    metadata_filters: Optional[dict] = None
```

Key Features:

- Represents a user's question with optional filters and retrieval settings
- Specifies how many chunks (`top_k`) to fetch and whether sources should be included
- Used for single-turn RAG queries without session history

2.1.2 Search Response Model

RetrievedChunk

```python
class RetrievedChunk(BaseModel):
    chunk_id: str
    doc_id: str
    text: str
    score: float
    metadata: Optional[dict] = None
```

Purpose: Represents a piece of text fetched from the vector DB during retrieval. Contains the chunk text, similarity score, source document ID, and metadata. Helps the frontend show evidence supporting the LLM's answer.

RAGSearchResponse

```python
class RAGSearchResponse(BaseModel):
    answer: str
    sources: Optional[List[RetrievedChunk]] = None
```

Purpose: Contains the final generated answer and optionally the retrieved source chunks. Used to deliver both the model's response and transparency of evidence. Essential for RAG-based systems that require citation or explainability.

3 Chat / Session API

3.1 Pydantic Models

3.1.1 Chat Request Model

ChatRequest

```python
class ChatRequest(BaseModel):
    session_id: Optional[str] = None
    message: str
    top_k: int = 5
```

Key Features:

- Defines a message in a multi-turn chat session with optional session ID
- If a session exists, previous history is reused; otherwise a new session is created
- Supports RAG answering in a conversational context

3.1.2 Chat Response Model

ChatResponse

```python
class ChatResponse(BaseModel):
    session_id: str
    answer: str
    history: List[dict]
    sources: Optional[List[RetrievedChunk]] = None
```

Purpose: Returns the AI reply along with the session ID and full chat history. Also includes the retrieved evidence chunks from the vector DB. Allows the frontend to maintain consistent multi-turn RAG conversations.

4 RAG Architecture

4.1 LangChain Framework

LangChain is a framework used to build RAG systems easily by providing tools for text splitting, embeddings, vector databases, retrieval, and LLM chains. It reduces the code you need and connects all RAG components together in one workflow.

4.2 Vector Databases

Pinecone

Overview: Pinecone is a managed, serverless vector database hosted only in the cloud (AWS, GCP, Azure) with no self-hosting option at all.

Key Features:

- Ultra-low latency (< 50 ms)
- Automatic scaling to billions of vectors
- Built-in replication, monitoring, and enterprise security (SOC 2, VPC peering)
- Free tier: up to 5 million vectors in serverless
- Production pricing: starting around \$70/month

Best For: Production applications needing real-time performance and zero infrastructure management, such as large-scale recommendation systems or RAG chatbots.

ChromaDB

Overview: ChromaDB is a fully open-source, embeddable vector database you install with a single pip command and run locally on your laptop, on-prem server, or any cloud VM.

Key Features:

- Completely free forever (only pay for cloud server if deployed)
- Zero vendor lock-in with full code control and customizability
- Deploy on AWS EC2, GCP, DigitalOcean, Fly.io, Railway
- Managed Chroma Cloud available (beta)
- Excellent metadata filtering and multi-tenancy support

Best For: Rapid prototyping, internal tools, startups, and environments where data cannot leave your infrastructure.

4.3 Session History Storage

MongoDB

Overview: MongoDB is a document-based NoSQL database that stores chat history as flexible JSON-like documents.

Key Features:

- Easy logging of multi-turn conversations without rigid schema management
- Each session stored as a document with conversation array, metadata, and timestamps
- Scales well and fits variable-length chat histories
- Easy integration with FastAPI using PyMongo or Motor

Best For: Rapid development and storing complex conversational data.

5 Implementation Requirements

Requirements Checklist

All items are quick to provide and required for deployment:

1. LLM API Key + Base URL

We need your preferred LLM provider credentials:

- Options: Gemini, Groq
- Powers both final answer generation in Search/Chat APIs
- Required from day one for testing

2. Vector Database Choice & Credentials

Choose one of the following (we'll set it up for you):
Option 1 – Pinecone (recommended for speed & zero maintenance):

- Pinecone API key
- Index name

Option 2 – ChromaDB:

- Self-host on your AWS account (ECS/EKS)
- Provide server/VM access

3. AWS Account Access

IAM user or role with permissions for:

- Lambda
- API Gateway
- CloudWatch

Note: We'll deploy everything serverless inside your account so you own it 100%.

4. MongoDB Atlas Connection String

- Used to securely store chat session history
- Stores conversation context and sessionId mappings