AI Knowledge Base & Customer Service Copilot

Report

Ayush Tandon
(tandonayush04@gmail.com)

---

1. Problem Statement

Teams often rely on internal documents such as HR policies, handbooks, and FAQs.
However:

- Information is scattered across files
- Searching manually is slow and error-prone
- Employees often receive inconsistent answers

Goal:
Build an AI-powered knowledge base assistant that can **answer user questions using internal documents**, while **strictly grounding responses in source material**.

---

2. High-Level Solution Overview

The system is built using a **Retrieval-Augmented Generation (RAG)** architecture.

Flow:

1. User uploads internal text documents
2. Documents are chunked and embedded
3. Embeddings are stored in a vector database
4. User asks a question
5. Relevant chunks are retrieved
6. LLM generates an answer **only from retrieved context**
7. Answer is returned with **ranked sources and line numbers**

This ensures **accuracy, transparency, and trust**.

---

3. Architecture Overview

Frontend

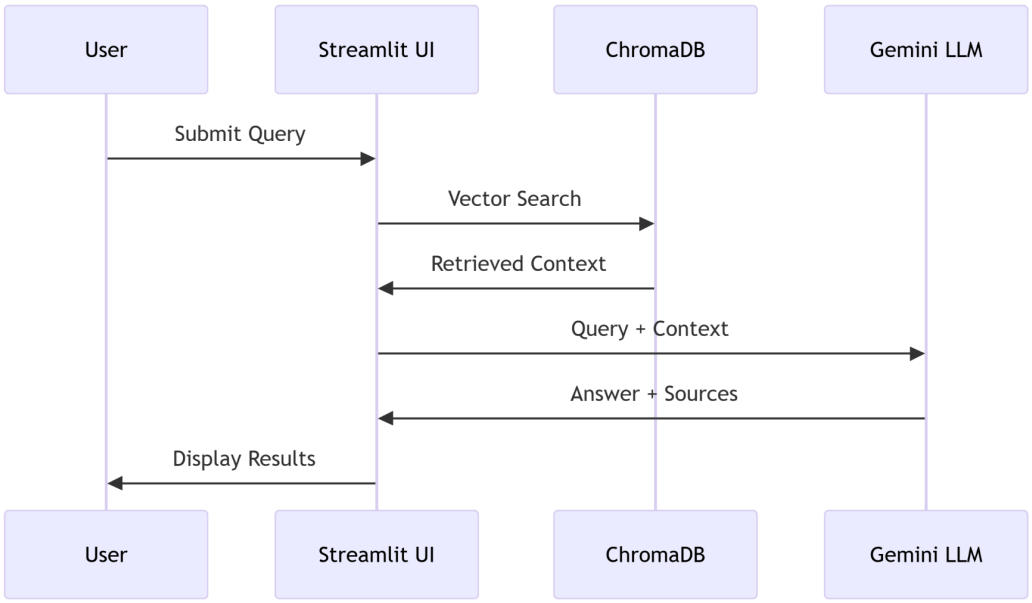
- Streamlit (Web UI)

Backend

- Python
- LangChain (orchestration)
- ChromaDB (vector storage)
- Gemini API (LLM)

Storage

- In-memory / local vector store: ChromaDB with local persistence (stores embeddings in `./chroma_db/`



---

4. Key Technical Decisions

4.1 Why Retrieval-Augmented Generation (RAG)?

Decision:
Used RAG instead of directly prompting the LLM.

Reason:

- Prevents hallucination
- Ensures answers come from documents only
- Allows clear source attribution
- Scales to large document sets

Tradeoff:

- Slightly more complex than direct LLM calls
- Worth it for correctness and explainability

4.2 Why ChromaDB for Vector Storage?

Decision:
Used ChromaDB as the vector database.

Why ChromaDB:

- Lightweight and easy to run locally
- No external service dependency
- Fast similarity search
- Ideal for small–medium document sets

Alternatives Considered:

- FAISS (lower-level, less metadata support)
- Pinecone (external dependency, overkill for prototype)

4.3 Why Sentence-Transformers for Embeddings?

Decision:
Used `sentence-transformers/all-MiniLM-L6-v2`.

Why:

- Fast inference
- High-quality semantic embeddings
- Runs locally (no API cost)
- Stable and well-documented

Tradeoff:

- Slightly less accurate than large hosted models
- Acceptable for internal knowledge search

4.4 Why Gemini API for LLM?

Decision:
Used Google Gemini via `langchain-google-genai`.

Why Gemini:

- Provides API to connect to LLM for free
- Lower latency for short answers
- Cost-effective for prototypes
- Easy integration with LangChain

Additional Controls:

- Rate limiting
- Throttling
- Retry with backoff

This avoids API quota exhaustion.

4.5 Why Streamlit for UI?

Decision:
Used Streamlit for frontend.

Why Streamlit:

- Rapid development
- Python-native
- Easy file upload
- Clean UI with minimal code

Features Implemented:

- File upload (user-provided documents)
- Query input
- Styled answers
- Ranked source display

---

5. Document Processing Strategy

5.1 Chunking

- Documents are split into fixed-size chunks
- Overlap is used to preserve context

Why chunking matters:

- Improves retrieval accuracy
- Prevents token overflow
- Enables precise source citation

5.2 Metadata Tracking

Each chunk stores:

- Source filename
- Line number range

This enables:

- Source ranking
- Transparent citations
- Debuggable responses

---

6. Source Ranking & Citations

How sources are ranked:

- Based on vector similarity score
- Top-K relevant chunks are selected

Why this matters:

- Users can verify answers
- Increases trust
- Prevents black-box behavior

---

7. Rate Limiting & Reliability

Problem Encountered:

- Gemini API quota limits (429 errors)

Solution:

- Implemented request throttling
- Limited max tokens
- Reduced unnecessary calls
- Cached embeddings

Result:

- Stable performance
- Predictable API usage

---

8. Deployment Decisions

8.1 Why Streamlit Cloud?

Decision:
Deployed on Streamlit Cloud.

Why:

- Free tier
- Simple GitHub integration
- No server management
- Ideal for demos

8.2 Dependency Management

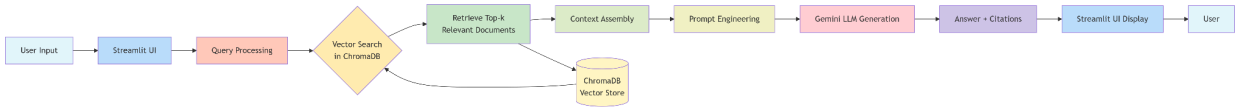
Key Challenges:

- Conflicting package versions
- HuggingFace compatibility issues

Solution:

- Explicit version pinning
- Stable legacy LangChain stack
- Reproducible builds

This demonstrates real-world debugging skills.



---

9. Security & Limitations

Current Limitations:

- No authentication
- In-memory vector store
- Text files only (can be extended to other formats)

Future Improvements:

- Role-based access
- Persistent storage
- Feedback loop for answer quality

---

10. Outcome & Impact

What this system achieves:

- Accurate document-grounded answers
- Transparent citations
- Fast knowledge retrieval
- Reduced manual searching

Business Value:

- Saves employee time
- Reduces misinformation
- Improves internal support efficiency

---

11. Implementation Details & Code Walkthrough

This section explains how the system is implemented end-to-end, mapping directly to the actual codebase.

11.1 Core Components

The application is structured into **logical modules**, each with a single responsibility.

```
├── app.py              # Streamlit UI + orchestration
├── query.py            # Retrieval + LLM answering logic
├── prompts.py          # Prompt templates
├── utils.py            # Chunking, helpers (optional)
├── requirements.txt    # Dependency management
```

Component Responsibilities

- **app.py**
  - Handles UI (file upload, query input, results display)
  - Triggers document indexing
  - Calls query pipeline
- **query.py**
  - Loads vector store
  - Performs similarity search
  - Calls Gemini with retrieved context
  - Returns answer + ranked sources
- **prompts.py**
  - Centralized simple prompt template
  - Ensures document-grounded answers only

This modular design improves **readability, debugging, and extensibility**.

11.2 Chunking Strategy – Deep Dive

Chunking is the most critical design decision in RAG.

Implementation

chunk_size = 1000
chunk_overlap = 200

Why this configuration?

- Preserves semantic continuity across chunks
- Prevents loss of context at boundaries
- Balances retrieval precision vs recall
- Keeps token usage under control

Metadata Stored Per Chunk

Each chunk stores:

- `source` → filename
- `line_start` → starting line number
- `line_end` → ending line number

This enables **exact citations** in the final answer.

---

12. Alternative Approaches Considered

Model	Pros	Cons	Decision
OpenAI embeddings	High accuracy	Paid, latency	Not used
BERT-base	Free, local	Large, slow	Rejected

MiniLM-L6	Fast, light	Slight accuracy drop	**Chosen**

Reason: Best balance between speed, quality, and cost.

---

13. Error Handling & Edge Cases

1. No relevant documents → Clear fallback message
2. API rate limits → Retry with exponential backoff
3. Empty queries → Input validation
4. Unsupported files → User-facing warning
5. Large files → Chunked progressive processing

---

14. Live Deployment

Live URL:

https://ai-knowledge-base-copilot-rag.streamlit.app/

Github Repo:

https://github.com/ayushtan123/ai-knowledge-base-copilot-rag

Sample Queries:

- “What is the leave policy?”
- “How many sick leaves are allowed?”
- “What are working hours?”