AI Question-Answering Service

Retrieval-Augmented Generation (RAG) System

Introduction & Demo

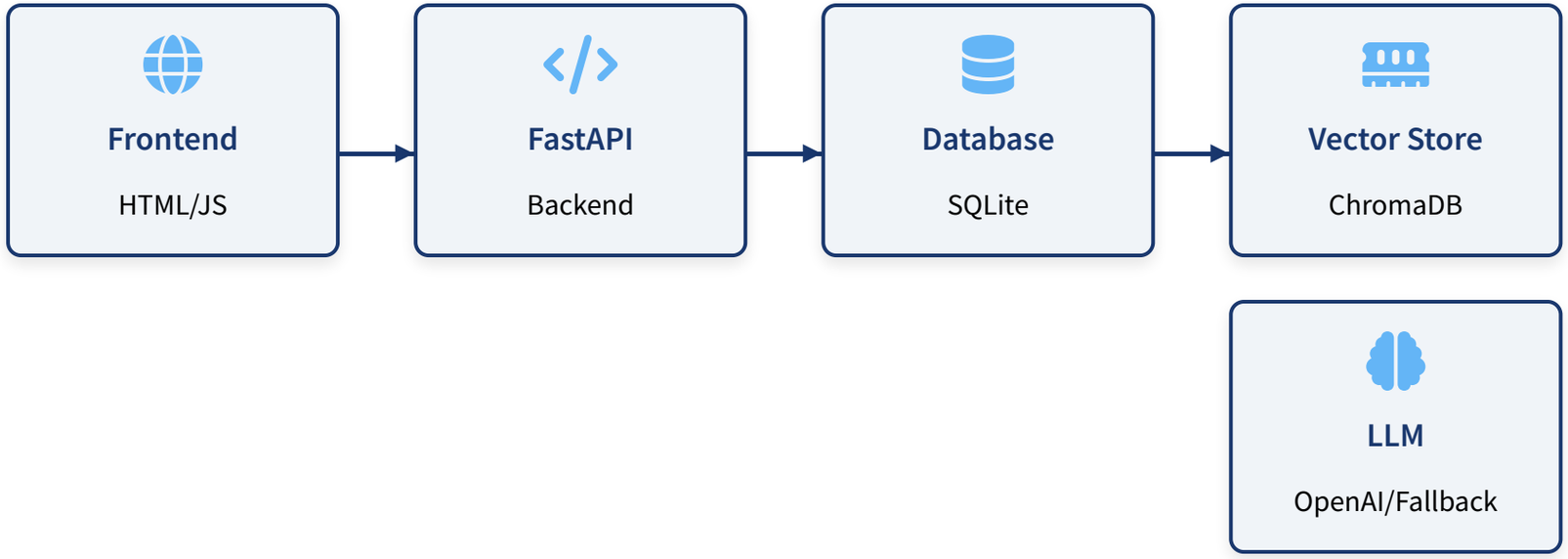
Presentation Structure (5-10 min)

- **Introduction & Demo**(2 min)
- **System Architecture**(2 min)
- **Key Technical Decisions**(2 min)
- **Future Improvements**(2 min)
- **Technical Highlights**(1 min)
- **Q&A**(1 min)

Live Demo Script

1. Homepage: Clean, responsive design
2. Register/Login: JWT authentication
3. Upload Document: PDF/TXT processing
4. Ask Questions: Real-time AI responses
5. Document Management: View & delete
6. Query History: Previous Q&A
7. API Documentation: OpenUI/Swagger

System Architecture



RESTful API

OpenAPI/Swagger auto-docs

JWT Authentication

Secure user management

Vector Database

ChromaDB semantic search

Document Processing

PDF/TXT intelligent chunking

LLM Integration

OpenAI GPT with fallback

Multi-user Support

Complete data isolation

User Isolation Best Practice: Filtering by user_id in every vector search ensures complete data separation between users - critical for multi-tenant RAG systems

Key Technical Decisions

What Worked Well

RAG Architecture

Retrieval-Augmented Generation for accurate, context-aware answers

Multi-user Isolation

Complete data separation between users

Fallback LLM

System works **without OpenAI API key**

Chunking Strategy

Overlapping chunks with sentence-boundary splitting

> Aligns with current best practices for context-aware chunking, preserving context and improving retrieval accuracy

Vector Similarity

Cosine similarity for semantic search

Docker-First

Complete containerization with docker-compose

Trade-offs Made

SQLite vs PostgreSQL

SQLite for simplicity, easy to switch

ChromaDB vs Elasticsearch

ChromaDB for easier setup, Elasticsearch option available

Simple UI vs React

Vanilla JS for zero build complexity

Synchronous Processing

Easier debugging, could add async for scale

What I'd Do Differently

Immediate Improvements (6-8 hour)

Advanced Chunking

1 H

Semantic-aware splitting using sentence embeddings for better context preservation

Streaming Responses

1 H

WebSocket support for real-time answers and improved user experience

Rich File Support

1-2 H

Add support for **DOCX, PPTX, HTML** processing and parsing

Better Error Handling

1 H

More **granular error responses** with user-friendly messages

Rate Limiting

1 H

API throttling and quotas to prevent abuse and ensure fair usage

What I'd Do Differently

Scaling Improvements (1 day)

Long RAG Implementation

4 H

Process **longer retrieval units** (sections or entire documents) instead of small chunks

- Improved retrieval efficiency
- Better context preservation
- Reduced computational costs

Async Processing

3 H

Background document processing with Celery

Caching Layer

2 H

Redis for embeddings and frequent queries

Advanced Search

3 H

Hybrid search (vector + keyword)

Analytics Dashboard

2 H

Usage metrics and insights

What I'd Do Differently

Production Features (2 days)

Self-RAG Implementation

5 H

Self-Reflective Retrieval-Augmented Generation with dynamic retrieval decisions and output evaluation

- Dynamic retrieval decisions
- Relevance evaluation
- Evidence-backed responses
- Self-critique mechanism

Document Versioning

4 H

Track changes over time with comparison

Team Workspaces

3 H

Shared document spaces with permissions

Advanced Security

4 H

Role-based access with audit logs

Monitoring

3 H

Observability stack with metrics & alerts

Technical Implementation Highlights

Code Quality

Complete Test Suite
15+ tests covering auth, upload, QA functionality

Type Hints
Full mypy compatibility for better code reliability

Documentation
Automatic OpenAPI/Swagger docs generation

Clean Architecture
Modular design with separation of concerns

Bonus Features Completed

Multi-user Support
Complete user isolation with secure data separation

Docker Compose
Full containerization with PostgreSQL + Elasticsearch

Elasticsearch Backend
Alternative vector database option for scalability

Web Interface
Complete HTML/JS frontend with responsive design

Production Ready
Environment configs, logging, and security measures

Performance & Security

Performance Stats

Upload Processing

~2-3 seconds

Query Response

~1-2 seconds

Chunk Processing

1000 chars + 200 overlap

Vector Search

Sub-100ms

Database Queries

Optimized indexing

Security Implementations

JWT Token Validation

Best Practice

```
def verify_token(token: str) ->
Optional[str]:
    try:
        payload = jwt.decode(token,
settings.secret_key, algorithms=
[settings.algorithm])
        return payload.get("sub")
    except JWTError:
        return None
```

Password Hashing

Best Practice

```
# Bcrypt password hashing
pwd_context = CryptContext(schemes=
["bcrypt"], deprecated="auto")

def get_password_hash(password):
    return pwd_context.hash(password)
```

Q&A Preparation

Likely Questions & Answers

How does the vector search work?

Sentence-transformers generate 384-dimensional embeddings for each chunk, then perform **cosine similarity** search to find relevant chunks.

What happens if OpenAI API is down?

System includes a **fallback LLM** that provides basic context-based responses using keyword matching and template generation.

How do you handle user data isolation?

Every database query and vector search is **filtered by user_id**, ensuring complete data separation between users.

Can this scale to thousands of users?

Yes - architecture supports **horizontal scaling** with load balancers, database read replicas, and distributed vector stores.

What about security?

JWT authentication, bcrypt password hashing, input validation, file type restrictions, and SQL injection protection.

How accurate are the responses?

Accuracy depends on document quality. **RAG approach** provides source context, making responses more reliable than pure generative models.