Healthcare Knowledge Assistant

RAG System - Backend Implementation

Submitted by: Md Rakibul Haque
Date: 06 November 2025
Assignment: Sr. LLM / Backend Engineer - 3 Hour Assignment

AI Usage Disclaimer

This submission was created without the use of AI code generation tools (Copilot, ChatGPT, Gemini, etc.) for the source code implementation. All code was written manually.

AI-Generated Content Disclosure:

The test documents in the test_documents/ directory (sample_en.txt, sample_ja.txt, heart_disease_en.txt) were generated using ChatGPT for testing purposes. These files are used solely for demonstration and testing of the RAG system's document ingestion and retrieval capabilities.

Citation:

Test documents generated by ChatGPT (OpenAI). Used for system testing and demonstration purposes only.

Table of Contents

1. Project Overview

README.md

Healthcare Knowledge Assistant - RAG System

A RAG-powered backend system for retrieving medical guidelines and research summaries in English and Japanese. Built with FastAPI, LangChain, FAISS, and sentence-transformers.

Features

- Multilingual support: English and Japanese document ingestion and querying
- Document ingestion with automatic language detection
- Vector search using FAISS for semantic document retrieval
- Mock LLM response generation with bilingual output
- Optional translation between English and Japanese
- API key authentication
- Docker support
- CI/CD pipeline with GitHub Actions

Setup

Prerequisites

- Python 3.11+
- pip
- Docker (optional)

Installation

1. Clone the repository:
```bash
git clone
cd acme-RAG
```

2. Create virtual environment:
```bash
python -m venv venv
source venv/bin/activate # On Windows: venv\Scripts\activate
```

3. Install dependencies:
```bash
pip install -r requirements.txt
```

4. Configure environment variables:
```bash
cp .env.example .env
# Edit .env to set your API key (optional - defaults are provided)
```

5. Run the application:
```bash
uvicorn app.main:app --reload --host 0.0.0.0 --port 8000
```

The API will be available at `http://localhost:8000`

Docker Setup

Build and run:
```bash
docker build -t healthcare-rag:latest .
docker run -d -p 8000:8000 \
-e API_KEY=acme-ai-secret-key-2025 \
-v $(pwd)/data:/app/data \
--name healthcare-rag \
healthcare-rag:latest
```

API Endpoints

Health Check
GET `/health`

Ingest Documents
POST `/ingest`
- Accepts .txt files in English or Japanese
- Headers: `X-API-Key: `
- Request: Multipart form data with file uploads

Retrieve Documents
POST `/retrieve`
- Returns top-k relevant documents with similarity scores
- Headers: `X-API-Key: `
- Request body: `{"query": "your query", "top_k": 3}`

Generate Response
POST `/generate`
- Combines retrieved docs + query into mock LLM response
- Headers: `X-API-Key: `
- Request body: `{"query": "your query", "output_language": "en", "top_k": 3}`

API Documentation

Interactive API documentation available at:
- Swagger UI: `http://localhost:8000/docs`
- ReDoc: `http://localhost:8000/redoc`

Design Notes

Scalability

The current implementation uses FAISS with a flat index for exact similarity search. For production at scale, consider:

1. **Index Optimization**: Upgrade to FAISS IndexIVF or HNSW for faster approximate search
2. **Distributed Storage**: Move FAISS index to cloud storage (S3, Azure Blob) for multi-instance deployments
3. **Caching Layer**: Implement Redis for caching frequently accessed queries
4. **Load Balancing**: Use nginx or cloud load balancers for horizontal scaling
5. **Database Backend**: Replace in-memory metadata storage with PostgreSQL or MongoDB
6. **Async Processing**: Use Celery for background document ingestion

Modularity

The architecture follows clear separation of concerns:

1. **Service Layer**: Independent services (embedding, FAISS, language, translation) that can be swapped or extended
2. **Router Layer**: Isolated API endpoints for easy extension
3. **Middleware**: Centralized authentication that can be extended
4. **Configuration**: Environment variables for easy configuration

Future improvements could include dependency injection, plugin system for embedding models, and strategy pattern for different vector stores.

Future Improvements

1. **LLM Integration**: Replace mock response generation with actual LLM API
2. **Advanced Chunking**: Implement semantic chunking with overlap
3. **Reranking**: Add cross-encoder reranking for better relevance
4. **Query Expansion**: Implement query expansion for better retrieval
5. **Multi-format Support**: Extend beyond .txt to PDF, DOCX, Markdown
6. **Metadata Filtering**: Support filtering by document metadata
7. **User Management**: Add user authentication and document ownership
8. **Analytics**: Track query patterns and system performance
9. **Monitoring**: Add logging, metrics (Prometheus), and error tracking
10. **Batch Processing**: Support batch ingestion for large document sets

Security Considerations

1. **API Key Management**: Use secure key management services in production
2. **Rate Limiting**: Implement rate limiting to prevent abuse
3. **Input Validation**: Enhanced validation for file uploads (size limits, content scanning)
4. **HTTPS**: Always use HTTPS in production
5. **CORS**: Configure CORS appropriately for your frontend
6. **Data Privacy**: Implement data encryption at rest and in transit
7. **Audit Logging**: Log all API access for security auditing

License

This project is created for the Acme AI assignment.

2. Source Code Files

app/main.py

```
from fastapi import FastAPI
from app.routers import ingest, retrieve, generate

app = FastAPI(
    title="Healthcare Knowledge Assistant API",
    description="RAG-powered assistant for retrieving medical guidelines and research summaries in
English and Japanese",
    version="1.0.0"
)

app.include_router(ingest.router, prefix="/ingest", tags=["ingestion"])
app.include_router(retrieve.router, prefix="/retrieve", tags=["retrieval"])
app.include_router(generate.router, prefix="/generate", tags=["generation"])


@app.get("/")
async def root():
    return {
        "message": "Healthcare Knowledge Assistant API",
        "status": "operational",
        "version": "1.0.0",
        "docs": "/docs"
    }


@app.get("/health")
async def health():
    return {"status": "healthy"}
```

app/routers/ingest.py

```
from fastapi import APIRouter, UploadFile, File, HTTPException, Depends
from typing import List
from app.models.responses import IngestResponse, IngestedDocument
from app.services.faiss_service import faiss_service
from app.services.language_service import language_service
from app.services.logging_service import logger
from app.middleware.auth import verify_api_key
from langchain_text_splitters import RecursiveCharacterTextSplitter

router = APIRouter()


@router.post("/", response_model=IngestResponse)
async def ingest_documents(
    files: List[UploadFile] = File(...),
    chunk_size: int = 500,
    api_key: str = Depends(verify_api_key)
):
    if not files:
        raise HTTPException(status_code=400, detail="No files provided")

    ingested_docs = []
    errors = []
```

```
    for file in files:
        try:
            if not file.filename or not file.filename.endswith('.txt'):
                errors.append(f"{file.filename}: Only .txt files are supported")
                continue

            content = await file.read()
            text = content.decode('utf-8')

            if not text.strip():
                errors.append(f"{file.filename}: Empty file")
                continue

            try:
                language = language_service.detect_language(text)
                logger.info(f"Detected language '{language}' for {file.filename}")
            except Exception as e:
                errors.append(f"{file.filename}: Language detection failed - {str(e)}")
                continue

            text_splitter = RecursiveCharacterTextSplitter(
                chunk_size=chunk_size,
                chunk_overlap=50
            )
            documents = text_splitter.create_documents(
                [text],
                metadatas=[{"language": language, "filename": file.filename}]
            )

            if documents:
                await faiss_service.add_documents_from_langchain(documents)
                ingested_docs.append(IngestedDocument(
                    filename=file.filename,
                    language=language,
                    chunks=len(documents),
                    size=len(text)
                ))

        except UnicodeDecodeError:
            errors.append(f"{file.filename}: Unable to decode file. Please ensure it's UTF-8
encoded.")
        except Exception as e:
            logger.error(f"Error processing {file.filename}: {e}")
            errors.append(f"{file.filename}: Error processing file - {str(e)}")

    total_docs = faiss_service.get_stats().get("total_documents", 0)

    return IngestResponse(
        status="success",
        ingested=len(ingested_docs),
        documents=ingested_docs,
        errors=errors if errors else None,
        total_documents_in_index=total_docs
    )
```

app/routers/retrieve.py

```
from fastapi import APIRouter, HTTPException, Depends
from app.models.requests import RetrieveRequest
from app.models.responses import RetrieveResponse, RetrievedDocument
from app.services.faiss_service import faiss_service
from app.middleware.auth import verify_api_key
```

```
router = APIRouter()


@router.post("/", response_model=RetrieveResponse)
async def retrieve_documents(
    request: RetrieveRequest,
    api_key: str = Depends(verify_api_key)
):
    if not request.query or not request.query.strip():
        raise HTTPException(status_code=400, detail="Query cannot be empty")

    results = await faiss_service.search(request.query, request.top_k or 3)

    retrieved_docs = [
        RetrievedDocument(
            id=metadata["id"],
            text=metadata["text"],
            language=metadata["language"],
            filename=metadata["filename"],
            similarity_score=round(similarity, 4)
        )
        for metadata, similarity in results
    ]

    return RetrieveResponse(
        query=request.query,
        results=retrieved_docs,
        total_results=len(retrieved_docs)
    )
```

app/routers/generate.py

```
from fastapi import APIRouter, HTTPException, Depends
from app.models.requests import GenerateRequest
from app.models.responses import GenerateResponse
from app.services.faiss_service import faiss_service
from app.services.language_service import language_service
from app.middleware.auth import verify_api_key
from app.utils.rag_utils import refine_query, format_retrieved_documents, generate_response

router = APIRouter()


@router.post("/", response_model=GenerateResponse)
async def generate_rag_response(
    request: GenerateRequest,
    api_key: str = Depends(verify_api_key)
):
    if not request.query or not request.query.strip():
        raise HTTPException(status_code=400, detail="Query cannot be empty")

    query_language = language_service.detect_language(request.query)

    output_language = request.output_language
    if not output_language:
        query_lower = request.query.lower()
        if "answer in english" in query_lower or "respond in english" in query_lower or "reply in
english" in query_lower:
            output_language = "en"
        elif "■■■" in query_lower or "■■■■■■" in query_lower:
            output_language = "en"
        elif "answer in japanese" in query_lower or "respond in japanese" in query_lower or "■■■■"
in query_lower:
```

```
            output_language = "ja"
        else:
            output_language = query_language

    if output_language not in language_service.SUPPORTED_LANGUAGES:
        raise HTTPException(
            status_code=400,
            detail=f"Unsupported output_language: {output_language}. Supported: en, ja"
        )

    refined_query = await refine_query(request.query, query_language)
    retrieved_results = await faiss_service.search(refined_query, request.top_k or 3)

    if not retrieved_results:
        raise HTTPException(
            status_code=404,
            detail="No relevant documents found. Please ingest documents first."
        )

    context_text = await format_retrieved_documents(retrieved_results)
    context_docs = [
        {
            "id": metadata["id"],
            "text": metadata["text"][:200] + "..." if len(metadata["text"]) &gt; 200 else metadata[
"text"],
            "language": metadata["language"],
            "filename": metadata["filename"],
            "similarity_score": round(similarity, 4)
        }
        for metadata, similarity in retrieved_results
    ]

    response = await generate_response(
        query=request.query,
        context=context_text,
        query_language=query_language,
        output_language=output_language
    )

    debug_info = {
        "refined_query": refined_query,
        "num_context_chunks": len(retrieved_results),
        "query_language": query_language,
        "output_language": output_language
    }

    return GenerateResponse(
        query=request.query,
        refined_query=refined_query,
        response=response,
        language=output_language,
        retrieved_context=context_docs,
        num_context_docs=len(context_docs),
        debug_info=debug_info
    )
```

app/services/faiss_service.py

```
import os
from typing import List, Tuple, Optional
import asyncio
from langchain_community.vectorstores import FAISS
from langchain_community.embeddings import HuggingFaceEmbeddings
```

```
from langchain_core.documents import Document
from app.services.logging_service import logger

STORAGE_DIR = "data"
FAISS_STORAGE_DIR = os.path.join(STORAGE_DIR, "faiss_index")


class FAISSService:
    def __init__(self):
        self.vectorstore: Optional[FAISS] = None
        self._index_loaded = False
        self._ensure_storage_dir()

    def _ensure_storage_dir(self):
        os.makedirs(STORAGE_DIR, exist_ok=True)
        os.makedirs(FAISS_STORAGE_DIR, exist_ok=True)

    def _get_embeddings(self):
        return HuggingFaceEmbeddings(
            model_name="paraphrase-multilingual-MiniLM-L12-v2",
            model_kwargs={'device': 'cpu'}
        )

    def _load_index(self):
        if self._index_loaded:
            return

        if os.path.exists(FAISS_STORAGE_DIR) and os.listdir(FAISS_STORAGE_DIR):
            try:
                embeddings = self._get_embeddings()
                self.vectorstore = FAISS.load_local(
                    FAISS_STORAGE_DIR,
                    embeddings,
                    allow_dangerous_deserialization=True
                )
                logger.info(f"Loaded FAISS index with {len(self.vectorstore.docstore._dict)}
documents")
                self._index_loaded = True
            except Exception as e:
                logger.warning(f"Error loading index: {e}. Creating new index.")
                self._create_new_index()
                self._index_loaded = True
        else:
            self._create_new_index()
            self._index_loaded = True

    def _create_new_index(self):
        self.vectorstore = None

    async def add_documents_from_langchain(self, documents: List[Document]):
        if not documents:
            return

        if not self._index_loaded:
            self._load_index()

        for doc in documents:
            doc.metadata["length"] = len(doc.page_content)

        loop = asyncio.get_event_loop()
        embeddings = self._get_embeddings()

        if self.vectorstore is None or len(self.vectorstore.docstore._dict) == 0:
            self.vectorstore = await loop.run_in_executor(
                None, FAISS.from_documents, documents, embeddings
            )
        else:
            await loop.run_in_executor(
                None, self.vectorstore.add_documents, documents
```

```
            )

        await loop.run_in_executor(None, self._save_index)
        logger.info(f"Added {len(documents)} documents to index. Total: {len(self.vectorstore.
docstore._dict)}")

    async def search(self, query: str, top_k: int = 3) -&gt; List[Tuple[dict, float]]:
        if not self._index_loaded:
            self._load_index()

        if self.vectorstore is None or len(self.vectorstore.docstore._dict) == 0:
            return []

        loop = asyncio.get_event_loop()
        docs_with_scores = await loop.run_in_executor(
            None, self.vectorstore.similarity_search_with_score, query, top_k
        )

        return [
            (
                {
                    "id": hash(doc.page_content) % 1000000,
                    "text": doc.page_content,
                    "language": doc.metadata.get("language", "en"),
                    "filename": doc.metadata.get("filename", "unknown"),
                    "length": len(doc.page_content)
                },
                1.0 / (1.0 + float(score))
            )
            for doc, score in docs_with_scores
        ]

    def _save_index(self):
        try:
            if self.vectorstore:
                self.vectorstore.save_local(FAISS_STORAGE_DIR)
        except Exception as e:
            logger.error(f"Error saving index: {e}")

    def get_stats(self) -&gt; dict:
        if not self._index_loaded:
            self._load_index()

        total_docs = len(self.vectorstore.docstore._dict) if self.vectorstore else 0

        return {
            "total_documents": total_docs,
            "dimension": 384,
            "index_type": "FAISS (LangChain)"
        }


faiss_service = FAISSService()
```

app/services/language_service.py

```
from langdetect import detect, LangDetectException


class LanguageService:
    SUPPORTED_LANGUAGES = {"en", "ja"}

    @staticmethod
```

```
    def detect_language(text: str) -&gt; str:
        if not text or not text.strip():
            raise ValueError("Empty text cannot be processed")

        try:
            detected_lang = detect(text)
            if detected_lang == "en":
                return "en"
            elif detected_lang == "ja":
                return "ja"
            else:
                return "en"
        except LangDetectException:
            return "en"

    @staticmethod
    def is_supported(language: str) -&gt; bool:
        return language in LanguageService.SUPPORTED_LANGUAGES


language_service = LanguageService()
```

app/services/translation_service.py

```
from googletrans import Translator
from typing import Optional
from app.services.logging_service import logger


class TranslationService:
    def __init__(self):
        try:
            self.translator = Translator()
        except Exception as e:
            logger.warning(f"googletrans initialization failed: {e}")
            self.translator = None

    def translate(self, text: str, target_lang: str, source_lang: Optional[str] = None) -&gt; str:
        if not text or not text.strip():
            return text

        if self.translator is None:
            logger.warning("Translator not initialized. Returning original text.")
            return text

        try:
            lang_map = {"en": "en", "ja": "ja"}
            target = lang_map.get(target_lang, "en")

            result = self.translator.translate(
                text,
                dest=target,
                src=source_lang if source_lang else None
            )

            return result.text
        except Exception as e:
            logger.warning(f"Translation error: {e}. Returning original text.")
            return text


translation_service = TranslationService()
```

app/services/logging_service.py

```
import logging
import sys
from typing import Optional

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.StreamHandler(sys.stdout),
        logging.FileHandler('app.log')
    ]
)

logger = logging.getLogger("healthcare_rag")
logger.setLevel(logging.INFO)


def get_logger(name: Optional[str] = None) -&gt; logging.Logger:
    if name:
        return logging.getLogger(f"healthcare_rag.{name}")
    return logger
```

app/middleware/auth.py

```
from fastapi import HTTPException, Header
from typing import Optional
import os
from dotenv import load_dotenv

load_dotenv()

VALID_API_KEY = os.getenv("API_KEY", "acme-ai-secret-key-2025")


async def verify_api_key(x_api_key: Optional[str] = Header(None, alias="X-API-Key")):
    if not x_api_key:
        raise HTTPException(
            status_code=401,
            detail="API key missing. Please provide X-API-Key header."
        )

    if x_api_key != VALID_API_KEY:
        raise HTTPException(
            status_code=403,
            detail="Invalid API key."
        )

    return x_api_key
```

app/models/requests.py

```
from pydantic import BaseModel, Field
from typing import Optional


class RetrieveRequest(BaseModel):
    query: str = Field(..., min_length=1, description="Search query in English or Japanese")
    top_k: Optional[int] = Field(default=3, ge=1, le=10, description="Number of results to return")


class GenerateRequest(BaseModel):
    query: str = Field(..., min_length=1, description="User query in English or Japanese")
    output_language: Optional[str] = Field(
        default=None,
        description="Output language: 'en' or 'ja'. Defaults to query language"
    )
    top_k: Optional[int] = Field(default=3, ge=1, le=10, description="Number of context chunks to
retrieve")
```

app/models/responses.py

```
from pydantic import BaseModel, Field
from typing import Optional, List, Dict


class IngestedDocument(BaseModel):
    filename: str
    language: str
    chunks: int
    size: int


class IngestResponse(BaseModel):
    status: str
    ingested: int
    documents: List[IngestedDocument]
    errors: Optional[List[str]] = None
    total_documents_in_index: int


class RetrievedDocument(BaseModel):
    id: int
    text: str
    language: str
    filename: str
    similarity_score: float = Field(..., ge=0.0, le=1.0)


class RetrieveResponse(BaseModel):
    query: str
    results: List[RetrievedDocument]
    total_results: int


class GenerateResponse(BaseModel):
    query: str
    refined_query: Optional[str] = None
    response: str
    language: str
    retrieved_context: List[Dict]
    num_context_docs: int
    debug_info: Optional[Dict] = None
```

app/utils/rag_utils.py

```
from typing import List, Dict, Tuple
from app.services.logging_service import logger
from app.services.language_service import language_service

try:
    from app.services.translation_service import translation_service
    if translation_service.translator is None:
        translation_service = None
except Exception:
    translation_service = None


async def refine_query(query: str, language: str = "en") -&gt; str:
    return query.strip()


async def format_retrieved_documents(documents: List[Tuple[Dict, float]]) -&gt; str:
    if not documents:
        return ""

    formatted_parts = []
    for i, (metadata, score) in enumerate(documents, 1):
        text = metadata.get('text', '')
        filename = metadata.get('filename', 'unknown')
        language = metadata.get('language', 'en')

        formatted_parts.append(
            f"[Document {i} - Source: {filename}, Language: {language}, Relevance: {score:.2f}]\n"
            f"{text}\n"
        )

    return "\n".join(formatted_parts)


async def generate_response(
    query: str,
    context: str,
    query_language: str,
    output_language: str
) -&gt; str:
    max_context_length = 2000
    formatted_context = context[:max_context_length]
    if len(context) &gt; max_context_length:
        formatted_context += "...\n[Context truncated]"

    context_summary = formatted_context[:1000]

    if output_language == "en":
        response = f"""Based on the retrieved medical guidelines and research documents, here is
the answer to your question:

{context_summary}

Please note that this information is based on the available documents. For personalized medical
advice, please consult with a healthcare professional."""
    else:
        response = f"""[redacted]

{context_summary}

[redacted]"""
```

```
if output_language != query_language and translation_service:
    try:
        response = translation_service.translate(
            response,
            target_lang=output_language,
            source_lang=query_language
        )
    except Exception as e:
        logger.warning(f"Translation error: {e}. Returning original response.")

return response
```

3. Configuration Files

requirements.txt

```
fastapi==0.104.1
uvicorn[standard]==0.24.0
python-multipart==0.0.6
faiss-cpu==1.12.0
langchain==1.0.3
langchain-community==0.4.1
langchain-core==1.0.3
langchain-text-splitters==1.0.0
sentence-transformers==5.1.2
huggingface-hub==0.36.0
langdetect==1.0.9
googletrans
pydantic==2.12.4
numpy==1.26.4
python-dotenv==1.0.0
requests==2.32.5
```

Dockerfile

```
FROM python:3.11-slim

WORKDIR /app

RUN apt-get update &amp;&amp; apt-get install -y \
    build-essential \
    &amp;&amp; rm -rf /var/lib/apt/lists/*

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY app/ ./app/

RUN mkdir -p /app/data

EXPOSE 8000

ENV PYTHONUNBUFFERED=1
ENV API_KEY=acme-ai-secret-key-2025

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

.github/workflows/ci-cd.yml

```
name: CI/CD Pipeline

on:
  push:
    branches: [ main, master ]
  pull_request:
```

```
    branches: [ main, master ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.11'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Lint with flake8
      run: |
        pip install flake8
        flake8 app --count --select=E9,F63,F7,F82 --show-source --statistics
        flake8 app --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics

    - name: Test API endpoints
      run: |
        python -c "from app.main import app; print('■ App imports successfully')"

    - name: Build Docker image
      run: |
        docker build -t healthcare-rag:test .

    - name: Run Docker container
      run: |
        docker run -d -p 8000:8000 -e API_KEY=test-key --name healthcare-rag-test healthcare-rag:
test
        sleep 5
        curl -f http://localhost:8000/health || exit 1
        docker stop healthcare-rag-test
        docker rm healthcare-rag-test

  deploy:
    needs: test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main' || github.ref == 'refs/heads/master'

    steps:
    - uses: actions/checkout@v3

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v2

    - name: Login to Docker Hub (optional)
      if: ${{ secrets.DOCKER_USERNAME != '' }}
      uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}

    - name: Build and push Docker image
      uses: docker/build-push-action@v4
      with:
        context: .
        push: ${{ secrets.DOCKER_USERNAME != '' }}
        tags: |
          ${{ secrets.DOCKER_USERNAME != '' &amp;amp;&amp;amp; format('{0}/healthcare-rag:latest', secrets.
DOCKER_USERNAME) || 'healthcare-rag:latest' }}
          ${{ secrets.DOCKER_USERNAME != '' &amp;amp;&amp;amp; format('{0}/healthcare-rag:{1}', secrets.
```

```
DOCKER_USERNAME, github.sha) || format('healthcare-rag:{0}', github.sha) }}
```

4. Design Notes

Scalability:

Right now I'm using FAISS with a flat index for exact search. For production, I'd upgrade to IndexIVF or HNSW for faster approximate search, add distributed storage like S3, implement Redis caching, use load balancers, and move metadata to PostgreSQL or MongoDB. Background processing with Celery would help with large document ingestion.

Modularity:

The code is split into separate layers - services handle FAISS, language detection, and translation; routers handle API endpoints; middleware handles auth. Each part can be swapped or extended independently. Configuration uses environment variables so it's easy to change without touching code.

Future Improvements:

1. Replace mock responses with real LLM API
2. Better chunking strategies with overlap
3. Add reranking for better results
4. Query expansion for improved retrieval
5. Support PDF, DOCX, Markdown formats
6. Metadata filtering
7. User management and document ownership
8. Analytics for query patterns
9. Monitoring with Prometheus and error tracking
10. Batch processing for large document sets

5. Security Implementation

API key auth is handled via X-API-Key header. The key is validated in middleware and can be set via environment variables.

For production, I'd add:

1. Proper secrets management (AWS Secrets Manager or HashiCorp Vault)
2. Rate limiting to prevent abuse
3. Better file upload validation (size limits, content scanning)
4. HTTPS everywhere
5. CORS configuration for frontend
6. Data encryption at rest and in transit
7. Audit logging for all API access

6. Deployment Configuration

The app is containerized with Docker and has a CI/CD pipeline using GitHub Actions. The Dockerfile uses Python 3.11-slim and installs dependencies from requirements.txt.

Docker Commands:

docker build -t healthcare-rag:latest .
docker run -d -p 8000:8000 -e API_KEY=acme-ai-secret-key-2025 --name healthcare-rag healthcare-rag:latest

GitHub Actions:

The CI/CD workflow runs on push to main/master. It does linting, tests imports, builds the Docker image, and runs health checks.

7. Testing Documentation

Test Documents:

I included sample files in test_documents for testing:

• sample_en.txt: English diabetes guidelines
• sample_ja.txt: Japanese diabetes guidelines
• heart_disease_en.txt: English heart disease guidelines

Note: These test files were generated using ChatGPT for testing only.

Japanese Language Support:

The system handles Japanese for both document ingestion and queries. Here's the Japanese test document:

Japanese Test Document Content:

[illegible]

2[illegible]

[illegible]
1. [illegible]80-130 mg/dL[illegible]180 mg/dL[illegible]

2. [illegible]

3. [illegible]

4. [illegible]150[illegible]

5. [illegible]A1C[illegible]7%[illegible]

6. [illegible]

7. [illegible]

■■■■■
- ■■
- ■■■

■■■
■■

API Testing:

You can test the API using Swagger UI at http://localhost:8000/docs, curl commands, or Postman.

Test Workflow:

1. Start the app: uvicorn app.main:app --reload
2. Ingest documents via POST /ingest
3. Test retrieval via POST /retrieve (works with English and Japanese)
4. Test generation via POST /generate (bilingual output)

Example Japanese Query:

POST /retrieve
{
"query": "■■■■■■■■■■■■■■■■■■■",
"top_k": 3
}

The system detects Japanese, retrieves relevant docs, and returns results in Japanese.