Eng_Jap_RAG_Assisstant

Github Repo: https://github.com/1morshed1/eng_japanese_RAG

Usage of AI:

For initial planning and validation I used claude pro, deepseek, qwen, kimi, glm 4.6 and cursor agent, Then for finalizing the plan and designing the initial system claude pro, deepseek and cursor was used. Cursor along with its mcp servers(context7, sequential thinking, exa search) was used all the time for checking on the correctness of my code. Cursor's memory bank was used to always have the full context of the project, and I also setup cursorrules so that the code format remains consistent overall. For debugging I used cursor agent, claude pro and deepseek.
The cursor mcp servers integration helped with my knowledge gap and implementing boilerplate codes. Each script was run through over and over in claude pro, deepseek and cursor , in order to maintain quality. Finally glm 4.6 used via CLine extension was used for file indexing , so that all the AI's which do not have knowledge bank like cursor can have full context to work with each time i open them. Also Claude Desktop along with mcp tools was used for final verification.The mcp tools were integrated from the site: https://smithery.ai/

The mermaid diagrams(flowcharts) were made with the help of file indexing capabilities of glm 4.6 agent mode, cursor agent mode and deepseek. Then visualized via the site: https://mermaid.live/edit

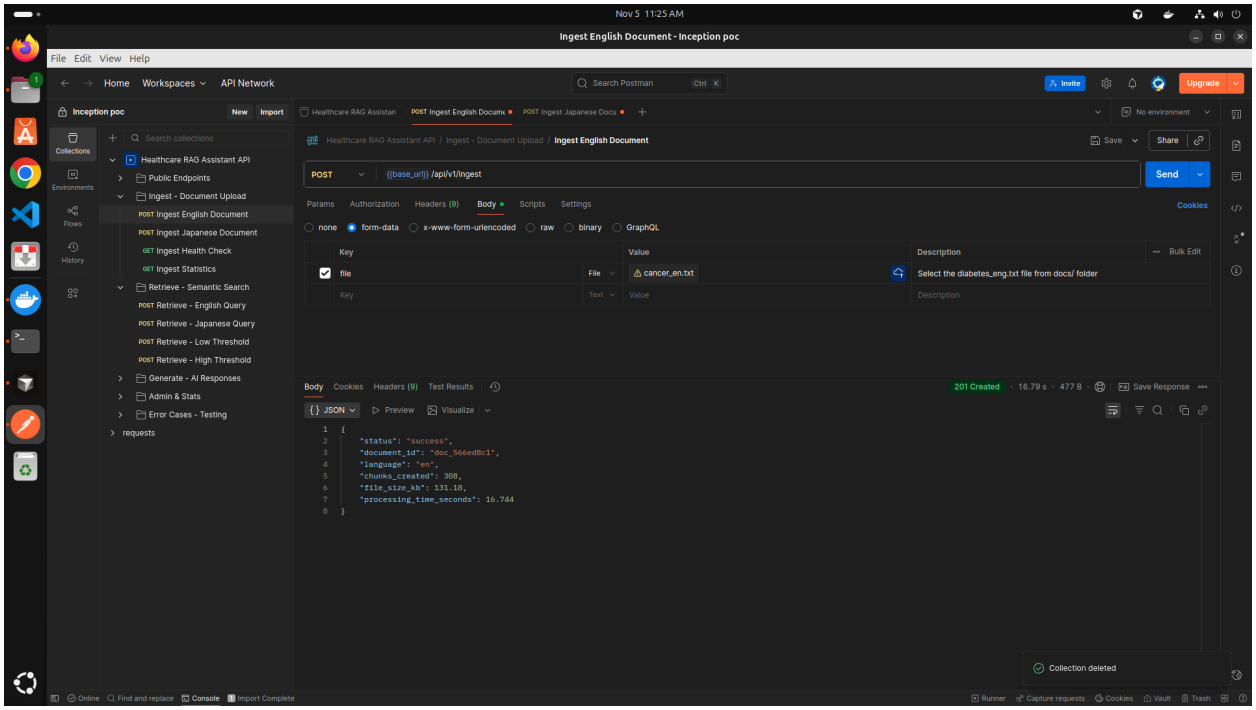
Running the app:

1. Clone the repo
2. Install the dependencies from requirements.txt
3. Setup .env file
4. Run the command uvicorn app.main:app --host 0.0.0.0 --port 8000 --reload
5. Setup api authentication
6. Start calling the api endpoints

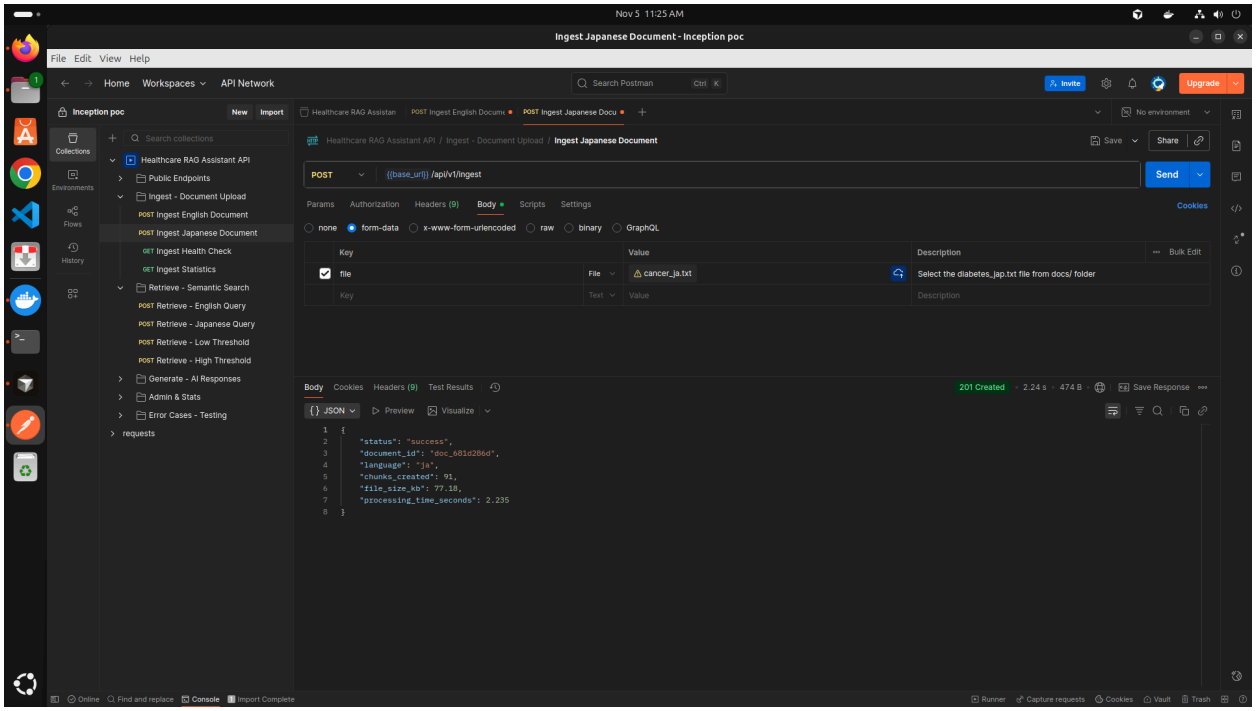
Postman Screenshots:

You can find the json for the postman collection in the github repo.

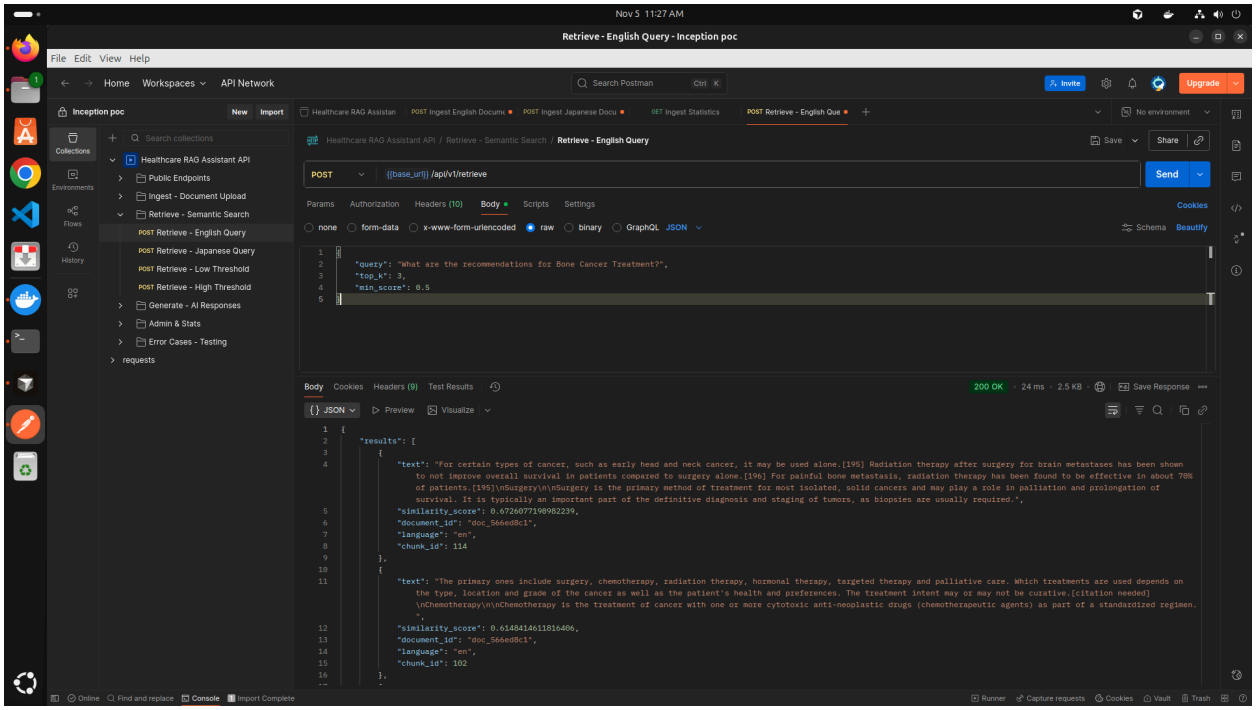
English Doc Ingest Endpoint:



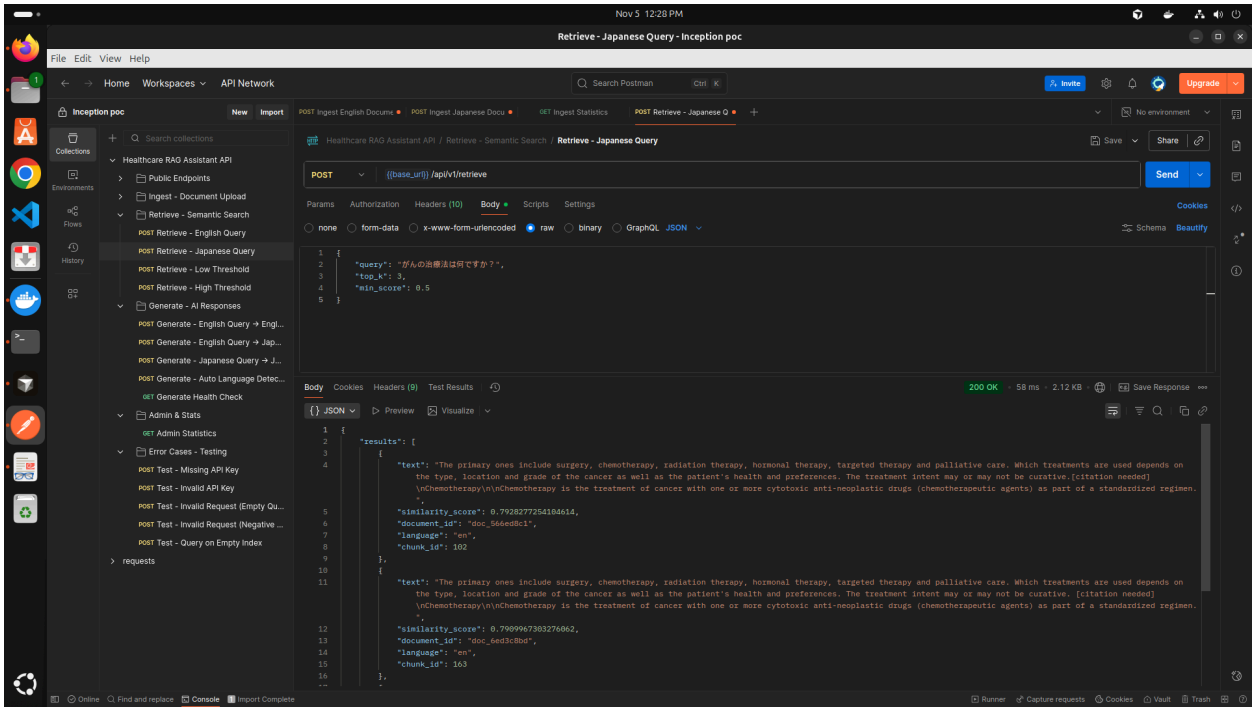
Japanese Doc Ingest Endpoint:



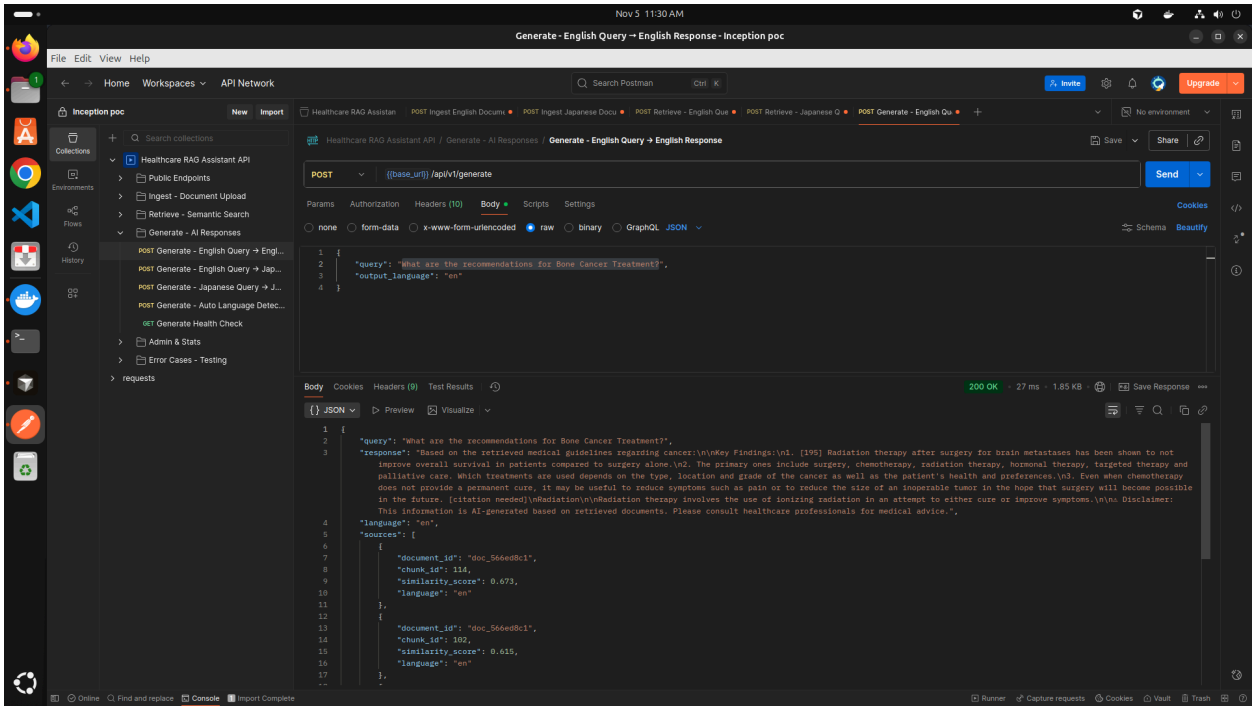
English Query Retrieve Endpoint:



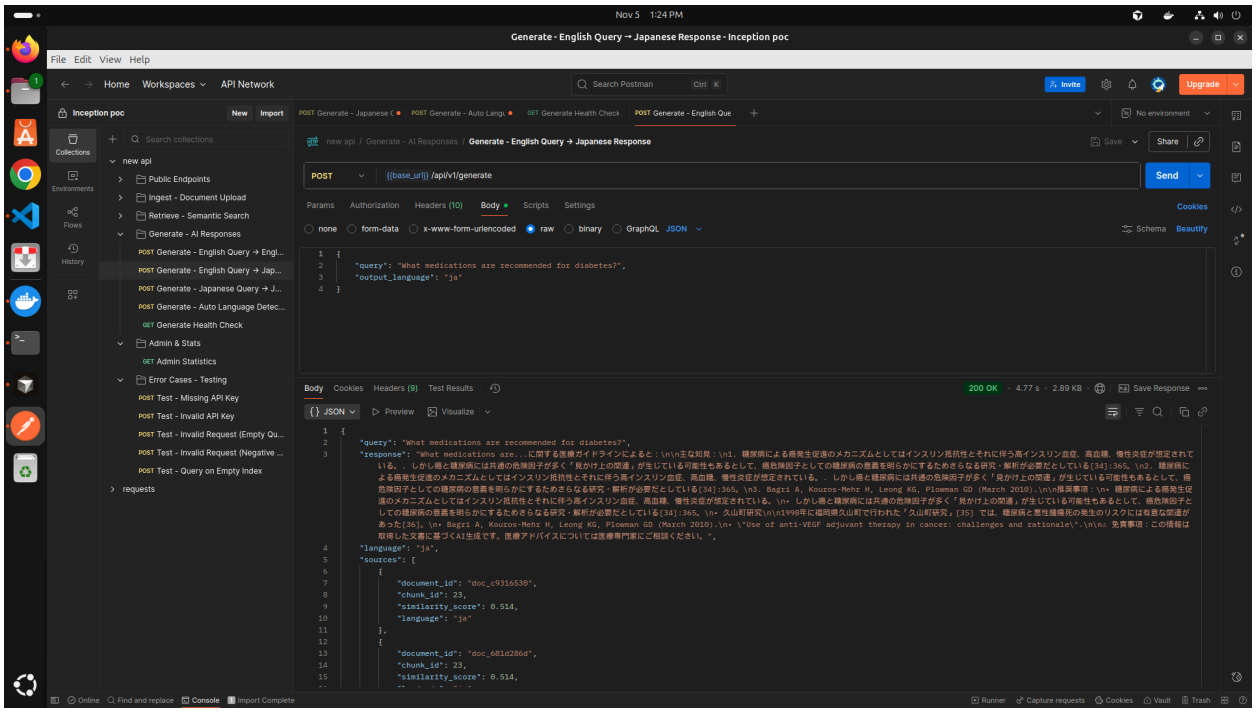
Japanese Query Retrieve Endpoint:



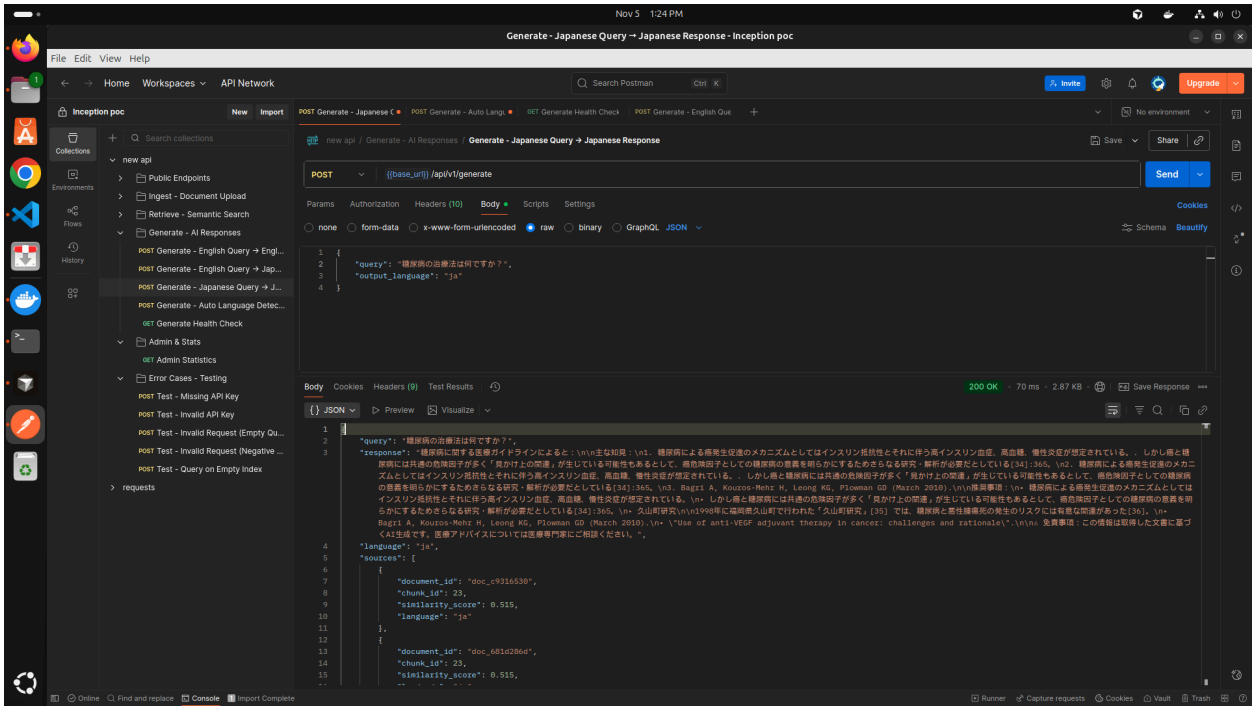
English query to English Response generate endpoint:



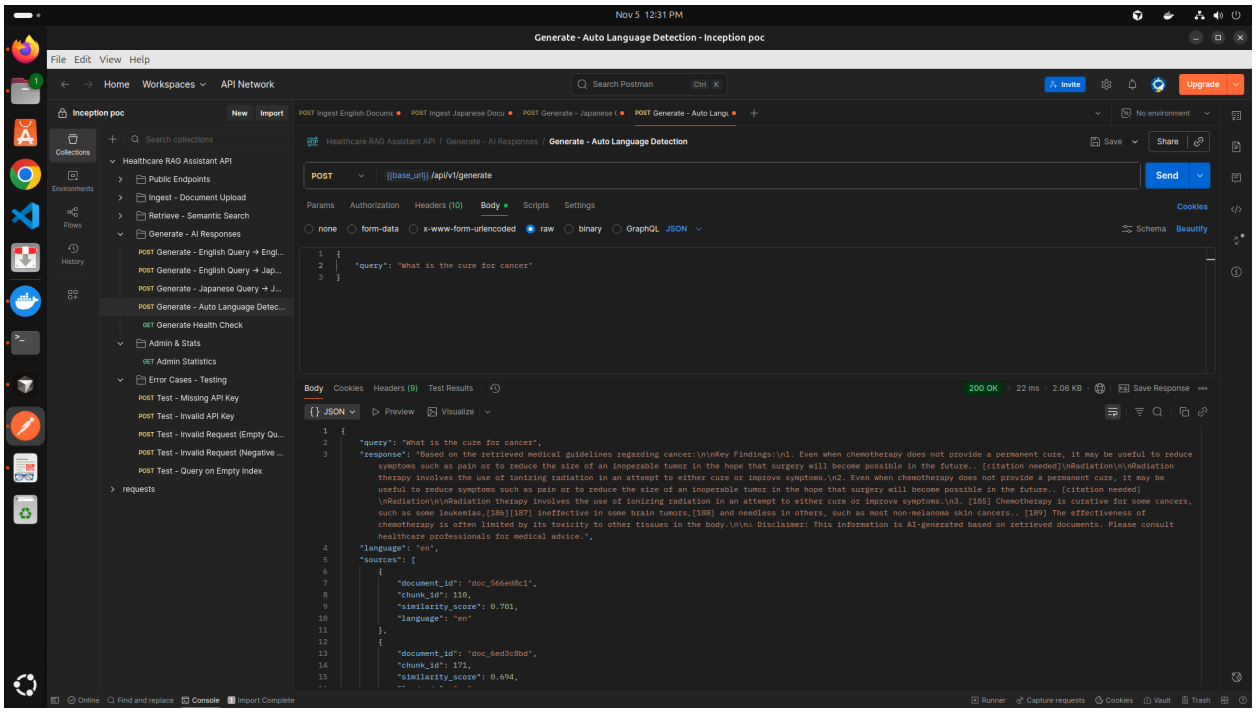
English query japanese response generate endpoint:



Japanese query japanese response generate endpoint:



Auto detect language generate endpoint:



Project Structure

```
eng_japanese_RAG/
├── .dockerignore                    # Docker ignore file
├── .env                             # Environment variables (not tracked
in git)
├── .env.example                     # Example environment variables
template
├── .gitignore                       # Git ignore file
├── docker-compose.yml               # Docker Compose configuration
├── Dockerfile                       # Docker image configuration
├── Healthcare_RAG_Assistant.postman_collection.json  # Postman API
collection
├── README.md                        # Project documentation
├── requirements.txt                 # Python dependencies
├── app/                             # Main application directory
│   ├── __init__.py                  # Package initialization
│   ├── config.py                    # Application configuration
│   ├── main.py                      # FastAPI application entry point
│   │
│   ├── middleware/                  # Middleware components
│   │   ├── __init__.py             # Package initialization
│   │   └── auth.py                 # Authentication middleware
│   │
│   ├── models/                      # Data models and schemas
│   │   ├── __init__.py             # Package initialization
│   │   └── schemas.py              # Pydantic schemas for API
│   │
│   ├── routes/                      # API route handlers
│   │   ├── __init__.py             # Package initialization
│   │   ├── generate.py             # Text generation routes
│   │   ├── ingest.py               # Data ingestion routes
│   │   └── retrieve.py             # Information retrieval routes
│   │
│   ├── services/                    # Business logic services
│   │   ├── __init__.py             # Package initialization
│   │   ├── embedding_service.py    # Text embedding service
│   │   ├── faiss_service.py        # FAISS vector database service
│   │   ├── llm_service.py          # Large Language Model service
│   │   └── translation_service.py # Translation service
│   │
```

```
│   └── utils/                      # Utility functions
│       ├── __init__.py            # Package initialization
│       ├── language_detector.py   # Language detection utilities
│       └── logger.py              # Logging configuration
│
├── data/                          # Data storage directory
│   └── (vector database files, documents, etc.)
│
├── logs/                          # Application logs directory
│   └── (application log files)
│
└── screenshots/                    # Project screenshots (22 files)
```

Project Configurations

Embedding Model: paraphrase-multilingual-MiniLM-L12-v2

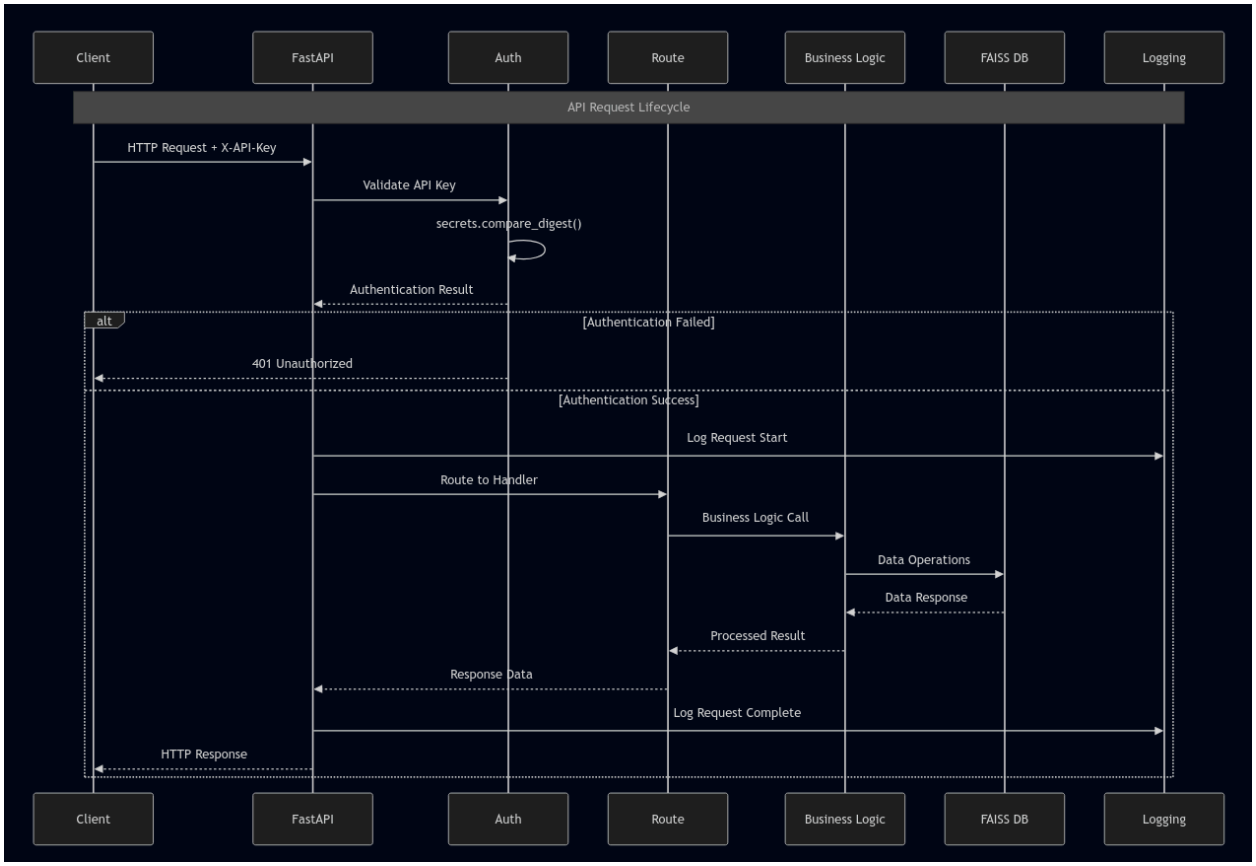
Embedding Dim: 384

Translation Models:
Eng to Jap: Helsinki-NLP/opus-mt-en-jap
Jap to Eng: Helsinki-NLP/opus-mt-jap-en

Tokenization:
For English: NLTK punkt tokenizer
For Japanese: Used Regex pattern

FAISS Indexing: IndexFlatIP

API Lifecycle:



Initiation: The process starts with an HTTP Request coming from the Client, which includes an X-API-Key header for authentication.

Authentication:

The request enters the Auth component.

The system Validates the API Key using secrets.compare_digest(), a secure method for comparing keys.

The result is a decision point:

Authentication Failed: The flow immediately jumps to sending a 401 Unauthorized HTTP Response back to the client.

Authentication Success: The flow continues to the next stage.

Request Logging & Routing:

Upon successful auth, the Logging component records the Request Start.

The FastAPI framework then Routes the request to the appropriate handler function.

Business Logic Execution:

The Route handler calls the Business Logic.

The Business Logic, in turn, performs Data Operations with the FAISS DB (a vector database often used for similarity search).

Response Formation:

The FAISS DB returns a Data Response.

The Business Logic processes this into a Processed Result.

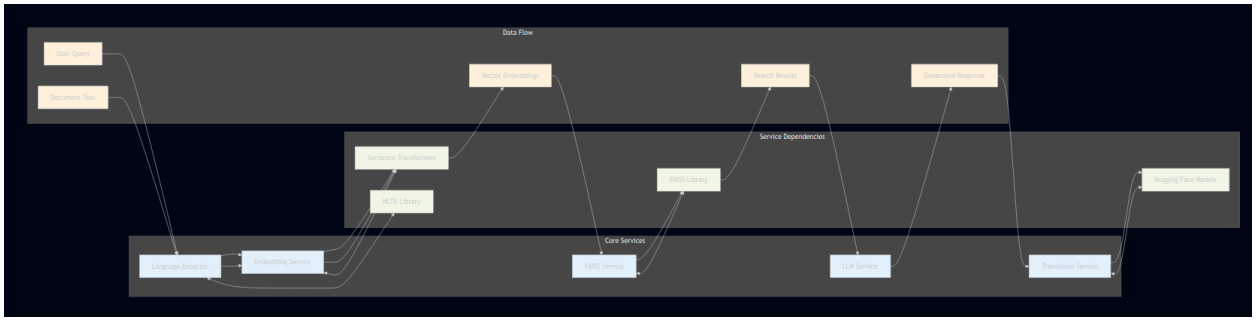
This result is formatted as Response Data.

Completion:

The Logging component records that the Request is Complete.

Finally, an HTTP Response containing the data is sent back to the Client.

Service Interactions:



The system has five main services:

Embedding Service - turns text into numbers

FAISS Service - searches through those numbers

LLM Service - generates answers

Translation Service - translates between languages

Language Detector - identifies what language text is in

Here's how it works:

A user question comes in and gets checked for language

The text is converted to numbers (embeddings)

The system searches for similar information using FAISS

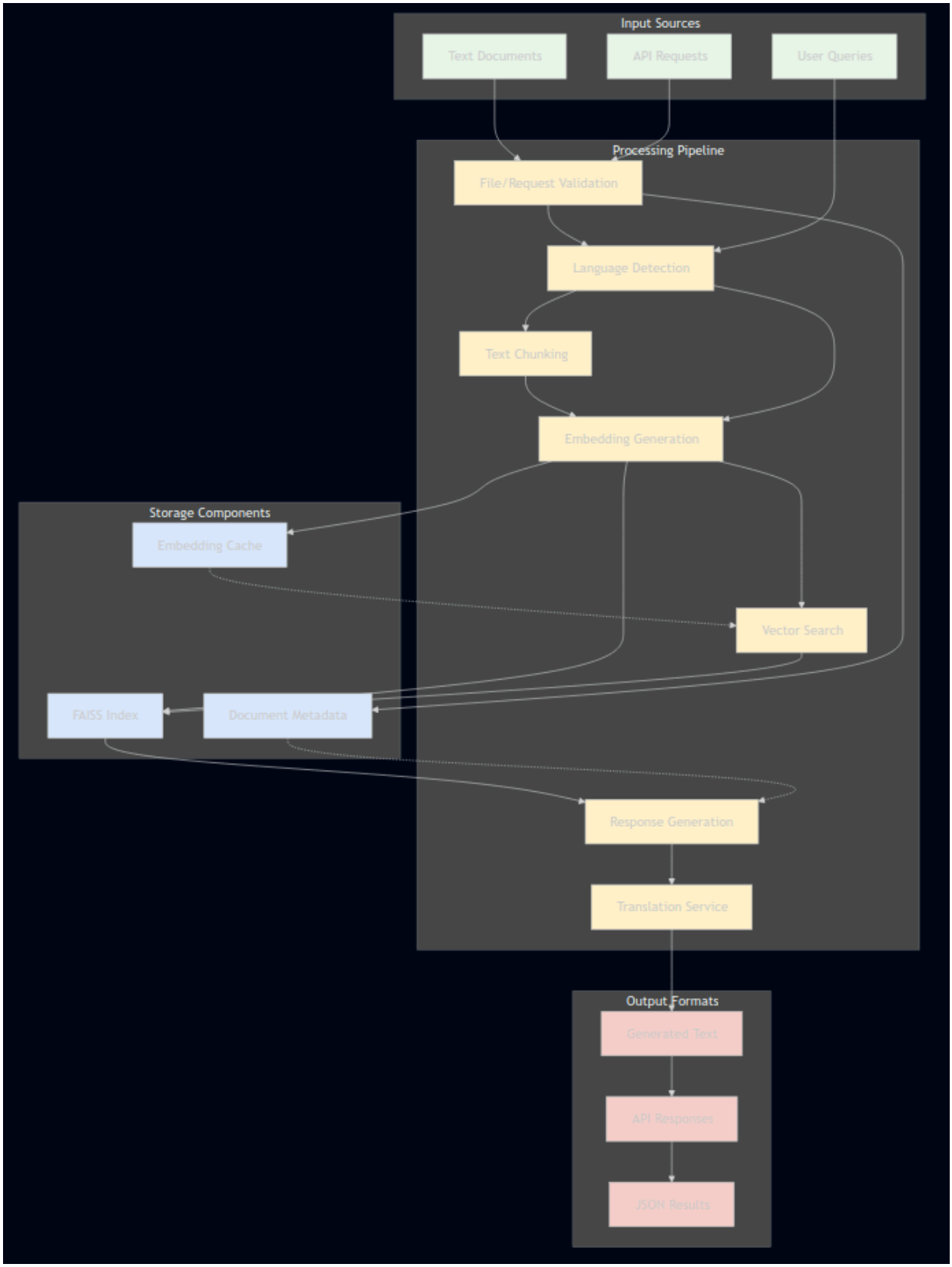
The LLM service creates an answer based on the search results

If needed, the answer gets translated

The final response goes back to the user

The system uses popular AI libraries like Sentence Transformers, Hugging Face, FAISS, and NLTK to power these services.

Data Flow Diagram:


Input Sources
Text Documents
API Requests
User Queries
Processing Pipeline
File/Request Validation
Language Detection
Text Chunking
Embedding Generation
Storage Components
Embedding Cache
Vector Search
FAISS Index
Document Metadata
Response Generation
Translation Service
Output Formats
Generated Text
API Responses
JSON Results

The system has four main parts:

Input Sources - Where information comes from

- Text Documents (files to search through)
- User Queries (questions people ask)
- API Requests (programmatic requests)

Processing Pipeline - The steps that handle the work

- Validation - checks if files and requests are valid
- Language Detection - figures out what language text is in
- Text Chunking - breaks large documents into smaller pieces
- Embedding Generation - converts text into number vectors
- Vector Search - finds similar content using math
- Response Generation - creates answers using AI
- Translation - converts between languages

Storage Components - Where data is saved

- FAISS Index - stores vectors for fast searching
- Document Metadata - keeps information about documents
- Embedding Cache - stores pre-computed vectors to save time

Output Formats - What the system produces

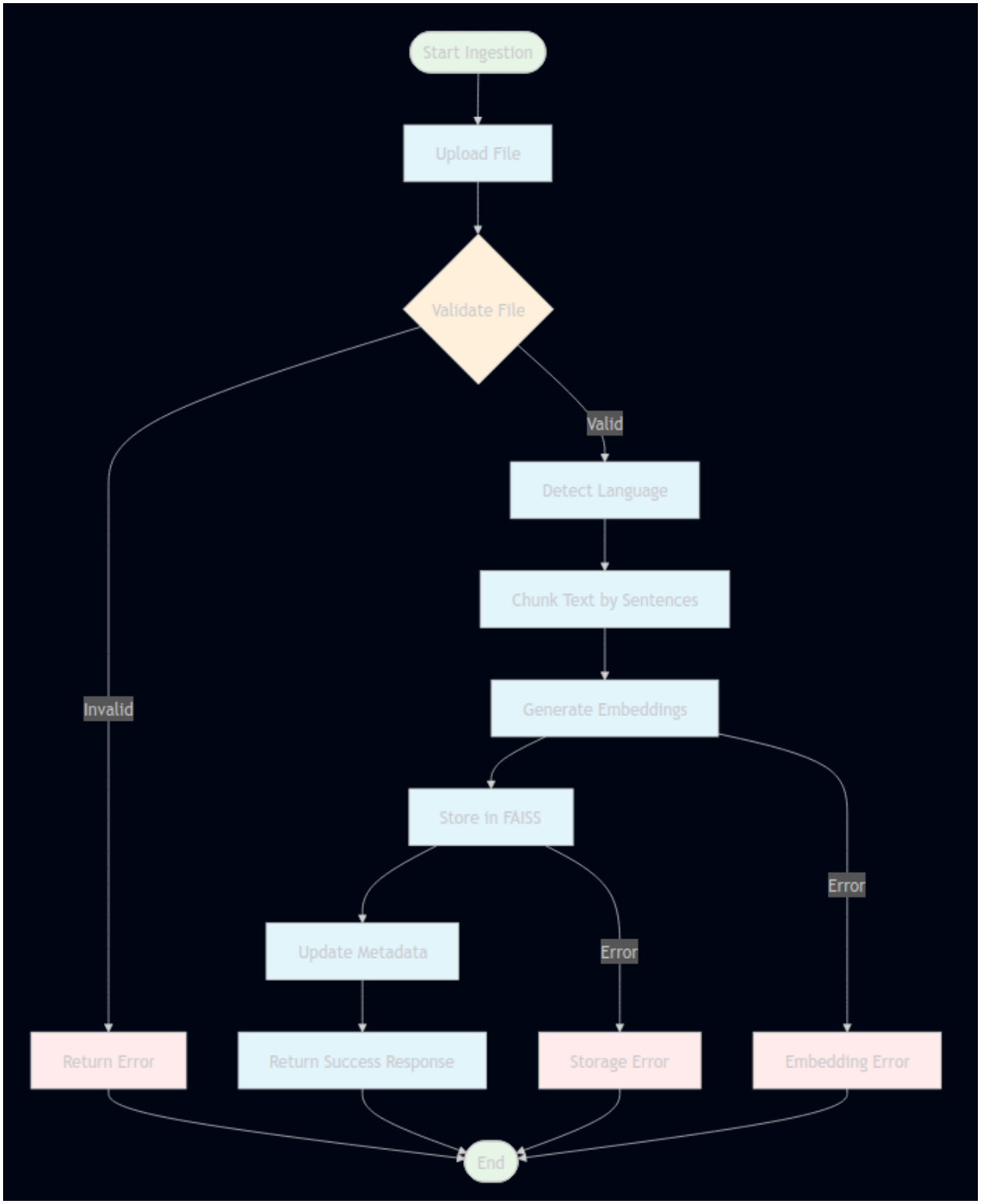
- API Responses - data for other programs
- JSON Results - structured data format
- Generated Text - AI-created answers

How it works:

For adding documents:
Text files get validated, have their language detected, are split into chunks, converted to vectors, and stored in the database.

For answering questions:
User questions get processed through language detection and vector conversion, then the system searches for similar content and generates answers, with optional translation.

Document Ingestion Flow:



The process starts when a file needs to be added to the system.

Step 1: Upload and Validate

First, a file is uploaded

The system checks if the file is valid

If invalid, it returns an error and stops

If valid, it continues to the next step

Step 2: Process the Content

Detect what language the text is in

Split the text into smaller chunks by sentences

Convert these text chunks into number vectors (embeddings)

Step 3: Store and Finish

Save the vectors in the FAISS search database

Update the document metadata (information about the document)

Return a success message to confirm everything worked

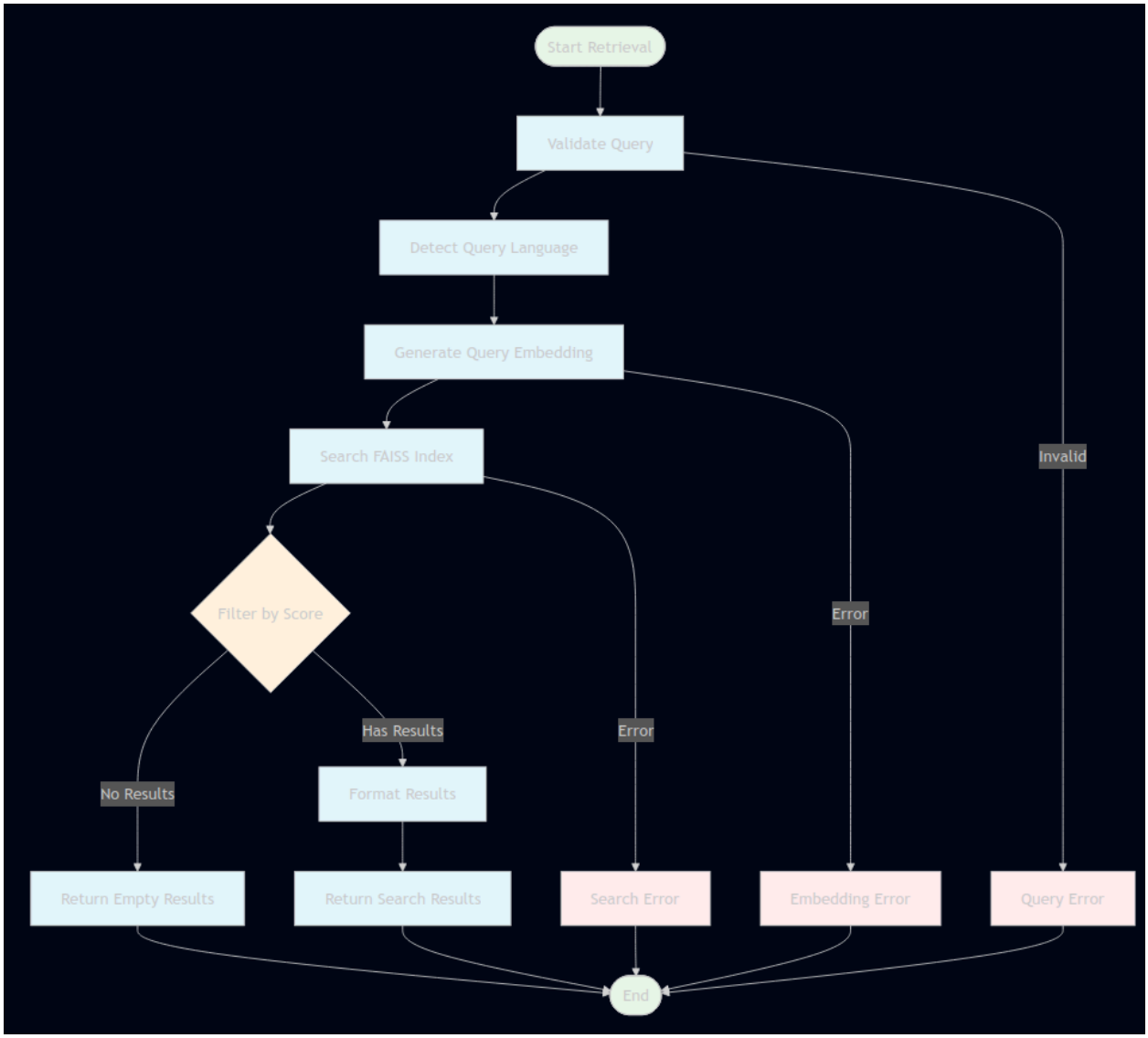
Error Handling:
If anything goes wrong during embedding generation or storage, the system catches the error and stops the process.

The process ends after either:

Successfully storing the document

Encountering any error along the way

Document Retrieval Flow:



The process goes like this:

Start - Begin the document ingestion process

Upload File - The document file is uploaded

Validate File - System checks if the file is valid

If invalid: Return error and end

If valid: Continue to next step

Process the document:

Detect what language the text is in

Split the text into smaller sentence chunks

Convert text chunks into number vectors (embeddings)

Store these vectors in the FAISS search database

Update document information in the metadata

Final step - Return success message

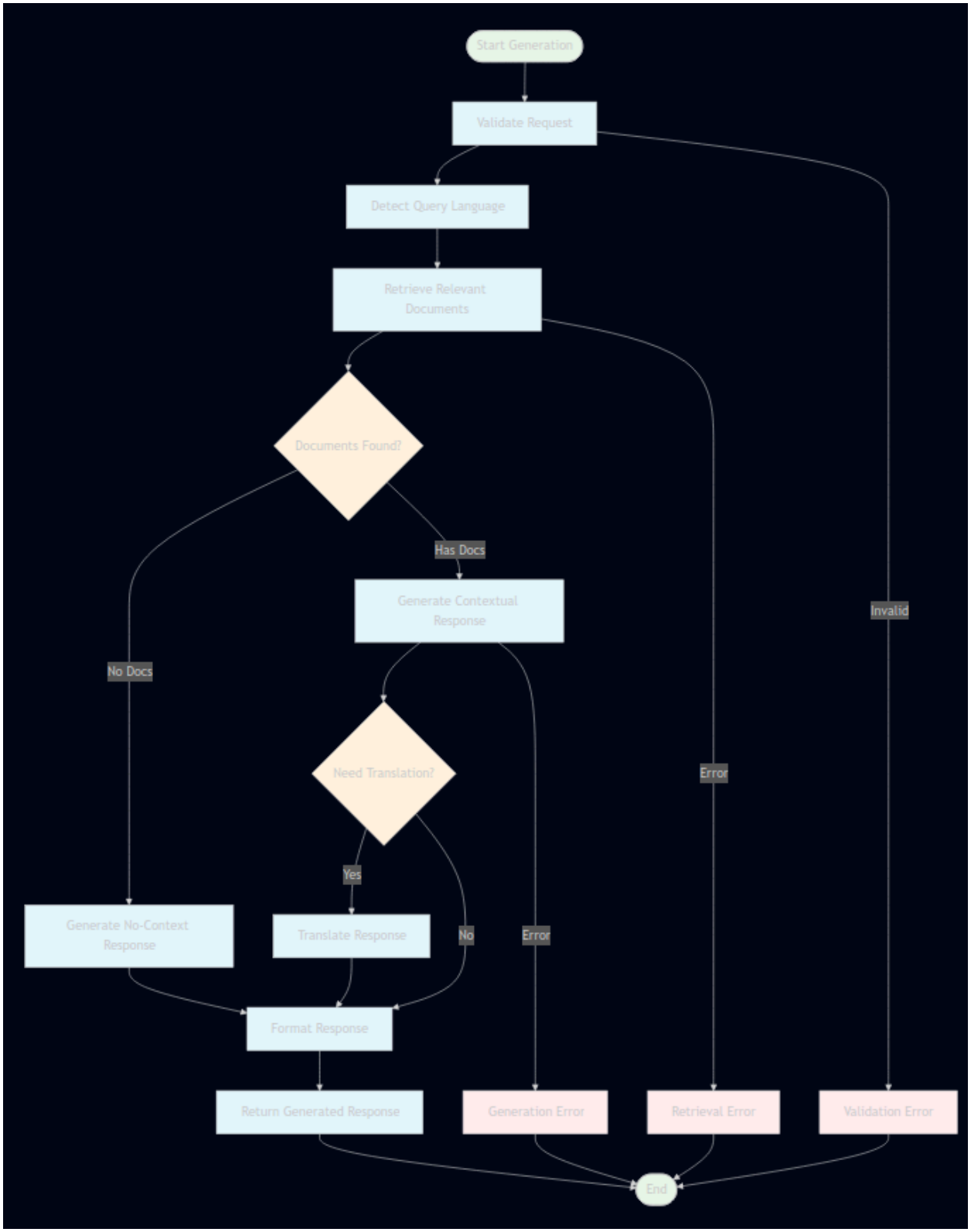
Error handling: If anything fails during embedding generation or storage, the system catches the error and stops the process.

The process ends when either:

The document is successfully added

Any error occurs along the way

Response Generation Flow:



The process goes like this:

Start - Begin processing a user's question

Validate Request - Check if the request is valid

If invalid: Return error and stop

Detect Language - Figure out what language the question is in

Search - Look for relevant documents in the database

Check Results - See if any documents were found

If no documents found: Generate a general answer without context

If documents found: Create an answer using the found information

Translation Check - Decide if the answer needs translation

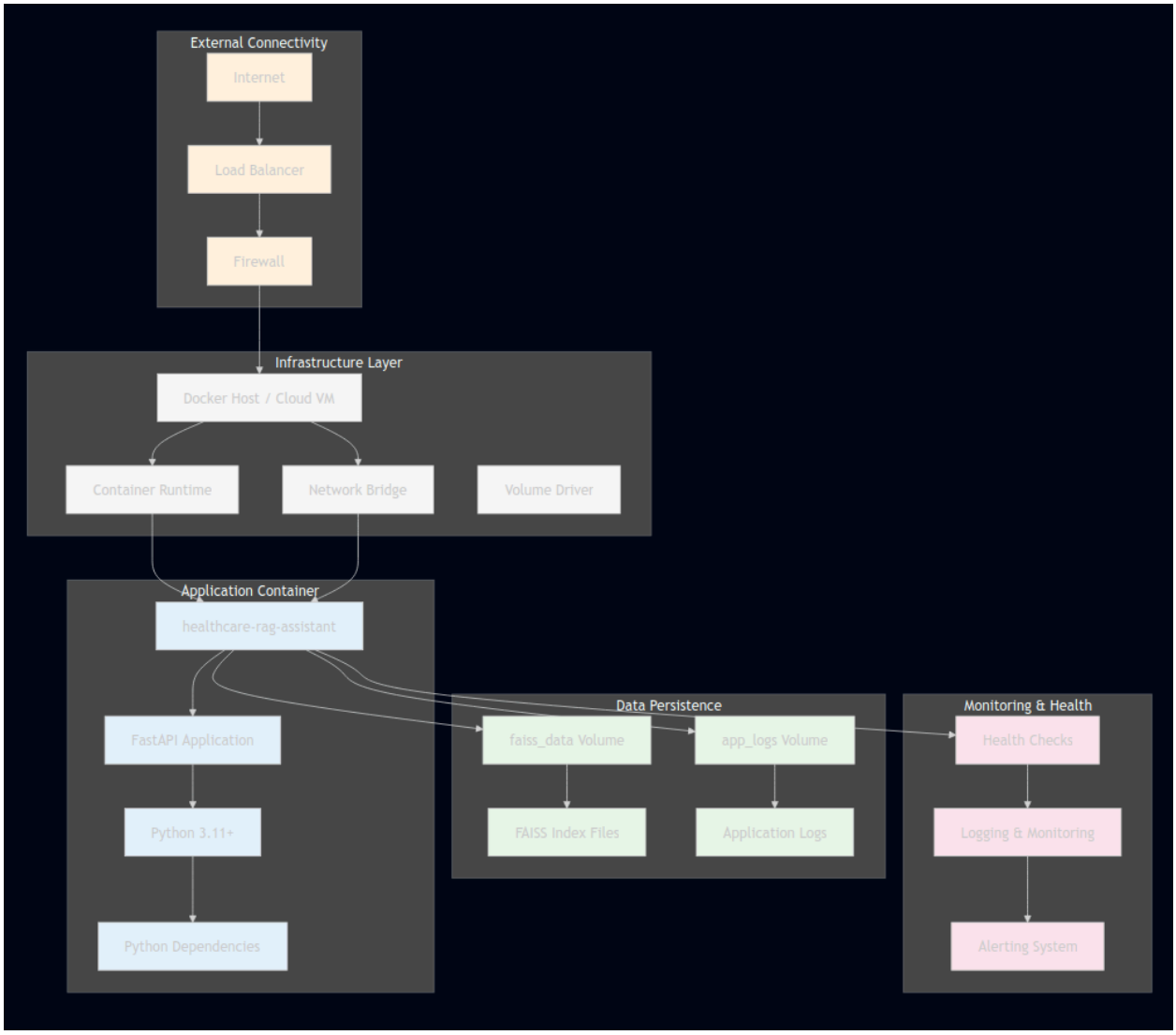
If translation needed: Translate the response

If no translation needed: Keep the original response

Final Steps - Format the response properly and return it to the user

Error handling: If anything fails during search or answer generation, the system catches the error and stops.

Deployment Architecture:



The System Infrastructure:

- Runs on a Docker host or cloud virtual machine
- Uses container technology to package the application
- Has network and storage systems set up

The Application:

- A container called "healthcare-rag-assistant" running a FastAPI application
- Uses Python 3.11 with all necessary libraries and dependencies

Data Storage:

FAISS Volume: Stores the search index files for fast document retrieval

Logs Volume: Stores application logs and activity records

Network Connections:

Internet traffic comes through a load balancer

Passes through a firewall for security

Reaches the Docker host and finally the application container

Monitoring & Health:

Health checks continuously verify the application is working

Monitoring systems track performance and logs

Alerting system notifies if problems occur

How it all connects:
Users on the internet send requests that flow through the load balancer, firewall, and network to reach the application container. The application uses the FAISS data to answer questions and saves logs to the storage volumes. Health monitoring runs constantly to ensure everything works properly.

Security Architecture:

Security Layers:
The system has four main security areas:

Network Security

Application Security

Data Security

- Authentication Security

Network Security:

- Uses HTTPS encryption for all connections
- Implements CORS policies to control access
- Has rate limiting to prevent abuse
- Uses firewall rules to block unauthorized access

Authentication & Authorization:

- Requires API keys for access
- Uses secure constant-time comparison for key checking
- Validates all API keys
- Tracks requests with unique IDs

Input Validation:

- Checks file types to prevent malicious uploads
- Enforces size limits
- Validates text encoding
- Checks content for safety

Response Security:

- Adds security headers to all responses
- Sanitizes data before sending
- Uses secure error handling that doesn't leak information
- Logs security events

Data Protection:

Encrypts stored data

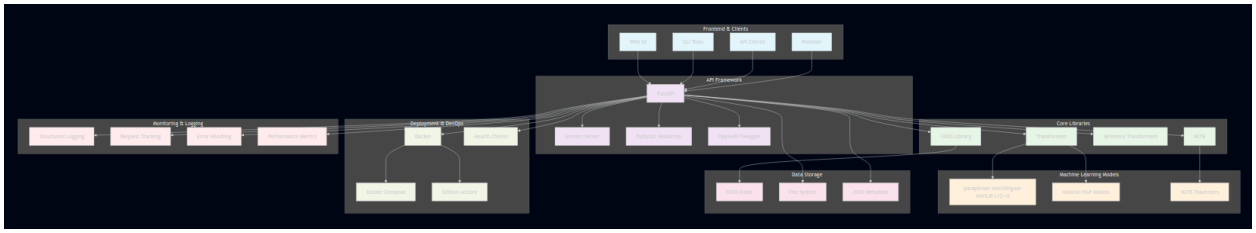
Controls who can access what data

Securely backs up data

Keeps audit logs of all access

How it works together:
The system checks security at every step - from network connections to user authentication, input validation, and data protection. All security measures work together to protect the healthcare data and ensure only authorized users can access the system.

Technology Stack:



Frontend & Clients:

Web UI for browser access

Command line tools for developers

API clients for programmatic access

Postman for testing

API Framework:

FastAPI as the main web framework

Uvicorn as the web server

Pydantic for data validation

OpenAPI/Swagger for documentation

Core Libraries:

- Transformers for AI models
- FAISS Library for vector search
- Sentence Transformers for text embeddings
- NLTK for text processing

Machine Learning Models:

- A multilingual embedding model for converting text to vectors
- Helsinki-NLP models for translation
- NLTK tokenizers for breaking down text

Data Storage:

- FAISS Index for vector storage
- File system for general storage
- JSON files for metadata

Deployment & DevOps:

- Docker for containerization
- Docker Compose for multi-container setup
- GitHub Actions for automation
- Health checks for monitoring

Monitoring & Logging:

- Structured logging for organized records
- Request tracking for debugging
- Error handling for reliability
- Performance metrics for optimization

How everything connects:
All clients connect to the FastAPI application, which uses the AI libraries and models to process requests. The system stores data in FAISS and files, runs in Docker containers, and is monitored with comprehensive logging. The entire setup is automated with GitHub Actions.

Now codes for individuals files will be shared below:

app/main.py:

```python
# app/main.py
import time
import uuid
import logging
import nltk
from fastapi import FastAPI, Request, Depends
from fastapi.middleware.cors import CORSMiddleware

from app.config import settings
from app.middleware.auth import api_key_auth
from app.routes import ingest, retrieve, generate
from app.utils.logger import setup_logging
from app.services.faiss_service import faiss_service
from app.services.embedding_service import embedding_service

# Setup logging
setup_logging()
logger = logging.getLogger(__name__)

# Track startup time
start_time = time.time()

# Create FastAPI app
app = FastAPI(
    title="Healthcare RAG Assistant",
    description="Bilingual medical knowledge assistant for clinicians",
    version="1.0.0"
)

# Security headers middleware
```

```python
@app.middleware("http")
async def add_security_headers(request: Request, call_next):
    response = await call_next(request)
    response.headers["X-Content-Type-Options"] = "nosniff"
    response.headers["X-Frame-Options"] = "DENY"
    response.headers["X-XSS-Protection"] = "1; mode=block"
    response.headers["Strict-Transport-Security"] = "max-age=31536000;
includeSubDomains"
    return response

# Request logging middleware
@app.middleware("http")
async def log_requests(request: Request, call_next):
    request_id = str(uuid.uuid4())
    request.state.request_id = request_id

    start = time.time()

    logger.info(
        "Request started",
        extra={
            "request_id": request_id,
            "method": request.method,
            "path": request.url.path,
            "client_ip": request.client.host if request.client else
"unknown"
        }
    )

    response = await call_next(request)

    duration = time.time() - start

    logger.info(
        "Request completed",
        extra={
            "request_id": request_id,
            "status_code": response.status_code,
            "duration": duration
        }
```

```
    )

    response.headers["X-Request-ID"] = request_id
    return response

# CORS middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=settings.CORS_ORIGINS.split(",") if hasattr(settings,
'CORS_ORIGINS')
                  else ["http://localhost:3000", "http://localhost:8000"],
    allow_credentials=True,
    allow_methods=["GET", "POST"],
    allow_headers=["X-API-Key", "Content-Type"],
)

# Startup event
@app.on_event("startup")
async def startup_event():
    """Initialize services on startup"""
    logger.info("Starting Healthcare RAG Assistant...")

    # Download NLTK data if not present
    try:
        nltk.data.find('tokenizers/punkt_tab')
        logger.info("NLTK punkt_tab already available")
    except LookupError:
        try:
            logger.info("Downloading NLTK punkt_tab tokenizer...")
            nltk.download('punkt_tab', quiet=True)
            logger.info("NLTK punkt_tab downloaded successfully")
        except Exception as e:
            logger.warning(f"Failed to download NLTK punkt_tab (will use
fallback): {e}")

    logger.info("FAISS index loaded automatically")
    logger.info("Services ready")

# Shutdown event
@app.on_event("shutdown")
```

```python
async def shutdown_event():
    """Cleanup on shutdown"""
    logger.info("Shutting down...")
    faiss_service._save_index()
    logger.info("Shutdown complete")

# Include routes
app.include_router(ingest.router, prefix="/api/v1", tags=["ingest"])
app.include_router(retrieve.router, prefix="/api/v1", tags=["retrieve"])
app.include_router(generate.router, prefix="/api/v1", tags=["generate"])

@app.get("/")
async def root():
    """Root endpoint"""
    return {
        "message": "Healthcare RAG Assistant API",
        "version": "1.0.0",
        "docs": "/docs"
    }

@app.get("/health")
async def health_check():
    """Enhanced health check endpoint"""
    try:
        return {
            "status": "healthy",
            "faiss_index_size": faiss_service.index.ntotal if
faiss_service.index else 0,
            "total_documents": faiss_service.doc_counter,
            "model_loaded": embedding_service.model is not None,
            "uptime_seconds": int(time.time() - start_time)
        }
    except Exception as e:
        logger.error(f"Health check failed: {e}")
        return {
            "status": "unhealthy",
            "error": str(e)
        }

@app.get("/admin/stats", dependencies=[Depends(api_key_auth)])
```

```python
async def get_admin_stats():
    """System statistics (requires authentication)"""
    import os

    index_size_mb = 0
    if faiss_service.index_path.exists():
        index_size_mb = os.path.getsize(faiss_service.index_path) / (1024 *
1024)

    return {
        "faiss": {
            "total_vectors": faiss_service.index.ntotal if
faiss_service.index else 0,
            "total_documents": faiss_service.doc_counter,
            "embedding_dimension": faiss_service.embedding_dim,
            "index_size_mb": round(index_size_mb, 2)
        },
        "uptime_seconds": int(time.time() - start_time)
    }
```

This is the main application file for Healthcare Search Assistant API built with FastAPI.

What the application does:

- Provides a bilingual medical knowledge assistant for doctors and clinicians
- Allows uploading, searching, and generating answers from medical documents

Key features:

Security & Setup:

- Uses API key authentication
- Adds security headers to prevent attacks
- Sets up CORS to control which websites can access the API
- Downloads necessary language processing data on startup

Logging & Monitoring:

- Assigns a unique ID to every request for tracking
- Logs all requests with timing information
- Provides health checks to monitor system status

Routes & Endpoints:

- / - Basic API information
- /health - System health check (shows index size, uptime, model status)
- /admin/stats - Detailed statistics (requires API key)
- Three main API routes: /ingest, /retrieve, /generate

How it works:

- When the server starts, it loads the search index and language tools
- Every request gets logged and assigned a tracking ID
- Security headers are added to all responses
- The health check monitors if the search system is working properly
- On shutdown, it automatically saves the search index

Main components:

- FastAPI web framework
- FAISS for document search
- Embedding service for text processing
- NLTK for language analysis

This is the core server that handles all document processing, searching, and question-answering functionality for the healthcare application.

app/config.py:

```python
# app/config.py
"""
Application configuration management using Pydantic Settings.

Environment variables are loaded from .env file or system environment.
All configuration is validated at startup.
"""

from pydantic_settings import BaseSettings, SettingsConfigDict
from pydantic import Field, field_validator
from typing import Optional
import os


class Settings(BaseSettings):
    """
    Application settings with validation.

    Environment Variables:
        API_KEYS: Comma-separated API keys (required)
        LOG_LEVEL: Logging level (default: INFO)
        MAX_FILE_SIZE_MB: Maximum file upload size (default: 10)
        EMBEDDING_MODEL: Sentence transformer model name
        ENV: Environment (development/production/testing)
    """

    model_config = SettingsConfigDict(
        env_file=".env",
        env_file_encoding="utf-8",
        case_sensitive=True,
        extra="ignore"
    )


    # API Configuration

    API_KEYS: str = Field(
        default="",
        description="Comma-separated list of valid API keys"
```

```
    )
    HOST: str = Field(default="0.0.0.0", description="API host")
    PORT: int = Field(default=8000, ge=1024, le=65535, description="API
port")
    ENV: str = Field(default="development", description="Environment")
    CORS_ORIGINS: str = Field(
        default="http://localhost:3000,http://localhost:8000",
        description="Comma-separated CORS origins"
    )


    # Logging Configuration

    LOG_LEVEL: str = Field(
        default="INFO",
        description="Logging level (DEBUG, INFO, WARNING, ERROR, CRITICAL)"
    )


    # Model Configuration

    EMBEDDING_MODEL: str = Field(
        default="paraphrase-multilingual-MiniLM-L12-v2",
        description="Sentence transformer model for embeddings"
    )
    TRANSLATION_MODEL_EN_JA: str = Field(
        default="Helsinki-NLP/opus-mt-en-ja",
        description="English to Japanese translation model"
    )
    TRANSLATION_MODEL_JA_EN: str = Field(
        default="Helsinki-NLP/opus-mt-ja-en",
        description="Japanese to English translation model"
    )


    # FAISS Configuration

    FAISS_INDEX_DIR: str = Field(
        default="data/faiss_index",
        description="Directory for FAISS index storage"
```

```
    )
    EMBEDDING_DIM: int = Field(
        default=384,
        description="Embedding dimension (must match model)"
    )


    # File Upload Limits

    MAX_FILE_SIZE_MB: int = Field(
        default=10,
        ge=1,
        le=100,
        description="Maximum file upload size in MB"
    )
    MAX_CHUNK_SIZE: int = Field(
        default=500,
        ge=100,
        le=2000,
        description="Target chunk size in characters"
    )
    MAX_CHUNK_OVERLAP: int = Field(
        default=1,
        ge=0,
        le=5,
        description="Number of sentences to overlap between chunks"
    )
    ALLOWED_EXTENSIONS: str = Field(
        default=".txt",
        description="Comma-separated allowed file extensions"
    )


    # Validators

    @field_validator('API_KEYS', mode='before')
    @classmethod
    def validate_api_keys(cls, v):
        """Ensure API keys are provided"""
        if not v or v.strip() == "":
```

```
            raise ValueError(
                "API_KEYS must be set in environment variables. "
                "Example: API_KEYS=key1,key2,key3"
            )
        return v

    @field_validator('LOG_LEVEL')
    @classmethod
    def validate_log_level(cls, v):
        """Validate log level"""
        valid_levels = ["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"]
        if v.upper() not in valid_levels:
            raise ValueError(f"LOG_LEVEL must be one of {valid_levels}")
        return v.upper()

    @field_validator('ENV')
    @classmethod
    def validate_env(cls, v):
        """Validate environment"""
        valid_envs = ["development", "production", "testing"]
        if v.lower() not in valid_envs:
            raise ValueError(f"ENV must be one of {valid_envs}")
        return v.lower()

    @field_validator('EMBEDDING_DIM')
    @classmethod
    def validate_embedding_dim(cls, v):
        """Validate embedding dimension"""
        valid_dims = [128, 256, 384, 512, 768, 1024]
        if v not in valid_dims:
            raise ValueError(
                f"EMBEDDING_DIM must be one of {valid_dims}. "
                f"Must match your embedding model's output dimension."
            )
        return v

    @field_validator('FAISS_INDEX_DIR', mode='after')
    @classmethod
    def create_index_dir(cls, v):
        """Ensure FAISS index directory exists"""
```

```
        os.makedirs(v, exist_ok=True)
        return v


    # Helper Methods

    def get_api_keys_list(self) -> list[str]:
        """
        Parse API_KEYS string into list.

        Returns:
            List of API key strings
        """
        return [key.strip() for key in self.API_KEYS.split(",") if
key.strip()]

    def get_cors_origins_list(self) -> list[str]:
        """
        Parse CORS_ORIGINS string into list.

        Returns:
            List of allowed CORS origin URLs
        """
        return [origin.strip() for origin in self.CORS_ORIGINS.split(",")
if origin.strip()]

    def get_allowed_extensions_set(self) -> set[str]:
        """
        Parse ALLOWED_EXTENSIONS string into set.

        Returns:
            Set of allowed file extensions
        """
        return {ext.strip() for ext in self.ALLOWED_EXTENSIONS.split(",")
if ext.strip()}

    @property
    def max_file_size_bytes(self) -> int:
        """
        Convert MAX_FILE_SIZE_MB to bytes.
```

```
        Returns:
            Maximum file size in bytes
        """
        return self.MAX_FILE_SIZE_MB * 1024 * 1024

    @property
    def is_production(self) -> bool:
        """Check if running in production"""
        return self.ENV == "production"

    @property
    def is_development(self) -> bool:
        """Check if running in development"""
        return self.ENV == "development"


# Create global settings instance
settings = Settings()


# Print configuration summary (only in development)
if settings.is_development:
    print("\n" + "="*50)
    print("Healthcare RAG Assistant - Configuration Loaded")
    print("="*50)
    print(f"Environment: {settings.ENV}")
    print(f"Host: {settings.HOST}:{settings.PORT}")
    print(f"Log Level: {settings.LOG_LEVEL}")
    print(f"API Keys Configured: {len(settings.get_api_keys_list())}")
    print(f"Max File Size: {settings.MAX_FILE_SIZE_MB}MB")
    print(f"FAISS Index Dir: {settings.FAISS_INDEX_DIR}")
    print(f"Embedding Model: {settings.EMBEDDING_MODEL}")
    print(f"Embedding Dimension: {settings.EMBEDDING_DIM}")
    print("="*50 + "\n")
```

This is the configuration file for the Healthcare Search Assistant. It manages all the settings and options for the application.

What it does:

- Loads settings from environment variables or a .env file
- Validates all configuration values
- Provides default values for missing settings

Main configuration sections:

API Settings:

- API keys (required - must be set)
- Host and port where the server runs
- Environment (development/production/testing)
- CORS origins (which websites can access the API)

Logging & Files:

- Log level (how detailed the logs should be)
- Maximum file upload size (default 10MB)
- Allowed file extensions (only .txt files by default)

AI Model Settings:

- Embedding model for converting text to vectors
- Translation models for English-Japanese translation
- Embedding dimension (384 - must match the model)

Search Index Settings:

- Where to store the FAISS search index
- How to chunk documents (500 characters with 1 sentence overlap)

Safety Features:

- Validates that API keys are provided

Checks that log levels are valid

Ensures the FAISS directory exists

Validates embedding dimensions match the model

Helper Methods:

Converts API keys string to a list

Converts file size from MB to bytes

Checks if running in production or development

Parses allowed file extensions

Startup Output:
In development mode, it prints a summary showing all the loaded configuration.

This file ensures the application has all the necessary settings to run properly and securely.

app/middleware/auth.py:

```python
# app/middleware/auth.py
"""
API Key authentication middleware for FastAPI.

Provides secure API key validation using constant-time comparison
and comprehensive logging for security monitoring.
"""

import logging
import secrets
from typing import Optional

from fastapi import Header, HTTPException, Request
from app.config import settings

logger = logging.getLogger(__name__)
```

```python
async def api_key_auth(
    request: Request,
    x_api_key: Optional[str] = Header(None, alias="X-API-Key")
) -> str:
    """
    API Key authentication dependency.

    Validates the X-API-Key header against configured API keys using
    constant-time comparison to prevent timing attacks.

    Args:
        request: FastAPI request object for logging and tracking
        x_api_key: API key from X-API-Key header

    Returns:
        str: The validated API key

    Raises:
        HTTPException: 401 if API key is missing or invalid

    Example:
        >>> @app.get("/protected", dependencies=[Depends(api_key_auth)])
        >>> async def protected_endpoint():
        >>>     return {"message": "Success"}

    Security:
        - Uses secrets.compare_digest() for constant-time comparison
        - Logs authentication attempts without exposing keys
        - Includes request tracking via request_id
    """
    # Get request ID for tracking
    request_id = getattr(request.state, 'request_id', 'unknown')

    # Get client IP for logging
    client_ip = request.client.host if request.client else "unknown"

    # Log authentication attempt
    logger.info(
        "Authentication attempt",
        extra={
```

```
            "request_id": request_id,
            "path": request.url.path,
            "method": request.method,
            "client_ip": client_ip,
            "has_api_key": x_api_key is not None
        }
    )

    # Check if API key is provided
    if not x_api_key:
        logger.warning(
            "Missing API key",
            extra={
                "request_id": request_id,
                "client_ip": client_ip,
                "path": request.url.path
            }
        )
        raise HTTPException(
            status_code=401,
            detail="Missing API key. Include 'X-API-Key' header in your
request.",
            headers={"WWW-Authenticate": "ApiKey"}
        )

    # Get valid API keys from settings
    valid_keys = settings.get_api_keys_list()

    # Validate API key using constant-time comparison
    # This prevents timing attacks that could leak valid key information
    is_valid = any(
        secrets.compare_digest(x_api_key, valid_key)
        for valid_key in valid_keys
    )

    if not is_valid:
        # Log failed authentication (but don't log the actual key)
        logger.warning(
            "Invalid API key",
            extra={
```

```
                "request_id": request_id,
                "client_ip": client_ip,
                "path": request.url.path,
                "api_key_prefix": x_api_key[:8] + "..." if len(x_api_key) >
8 else "***"
            }
        )
        raise HTTPException(
            status_code=401,
            detail="Invalid API key. Please check your credentials.",
            headers={"WWW-Authenticate": "ApiKey"}
        )

    # Log successful authentication
    logger.info(
        "Authentication successful",
        extra={
            "request_id": request_id,
            "client_ip": client_ip,
            "path": request.url.path
        }
    )

    return x_api_key


class APIKeyAuth:
    """
    Alternative class-based authentication dependency.

    Useful if you need to maintain state or customize behavior per
instance.

    Example:
        >>> auth = APIKeyAuth()
        >>> @app.get("/protected", dependencies=[Depends(auth)])
        >>> async def protected_endpoint():
        >>>     return {"message": "Success"}
    """
```

```
    def __init__(self):
        """Initialize the API key authenticator."""
        self.valid_keys = set(settings.get_api_keys_list())
        logger.info(f"APIKeyAuth initialized with {len(self.valid_keys)}
valid keys")

    async def __call__(
        self,
        request: Request,
        x_api_key: Optional[str] = Header(None, alias="X-API-Key")
    ) -> str:
        """
        Validate API key.

        This is a wrapper around the api_key_auth function for class-based
usage.
        """
        # Use the same logic as the function
        return await api_key_auth(request, x_api_key)


# Create a global instance for easy importing
api_key_auth_instance = APIKeyAuth()
```

This is the authentication system for the Healthcare Search Assistant API. It handles API key security.

What it does:

Protects API endpoints by requiring valid API keys

Validates keys securely to prevent attacks

Logs all authentication attempts for security monitoring

How it works:

Key Validation:

Looks for an X-API-Key header in incoming requests

Compares the provided key against a list of valid keys from the configuration

Uses secrets.compare_digest() for secure comparison (prevents timing attacks)

Security Features:

Constant-time comparison: Always takes the same time to check keys, preventing attackers from guessing valid keys by timing responses

Secure logging: Logs authentication attempts but never exposes the full API key in logs (only shows first 8 characters for tracking)

Request tracking: Links all authentication events to request IDs for debugging

Error Handling:

If no API key is provided: Returns 401 error "Missing API key"

If invalid API key is provided: Returns 401 error "Invalid API key"

Includes security headers in error responses

Logging:

Logs every authentication attempt with request details

Records successful and failed attempts

Tracks client IP addresses and request paths

Uses request IDs to trace authentication through the system

Two Ways to Use:

Function version: api_key_auth - directly protect endpoints

Class version: APIKeyAuth - for more complex scenarios

This authentication system ensures that only authorized users with valid API keys can access the healthcare application's sensitive endpoints.

app/models/schemas.py:

```
# app/models/schemas.py
"""
```

```
Pydantic models for request/response validation.

All models are automatically validated by FastAPI and generate
OpenAPI documentation with examples.
"""

from pydantic import BaseModel, Field, field_validator, ConfigDict
from typing import Optional, List, Literal
from datetime import datetime



# Shared Validators

def validate_query_field(v: str) -> str:
    """
    Shared validator for query fields.

    Ensures query is not empty or just whitespace.
    """
    if not v or not v.strip():
        raise ValueError('Query cannot be empty or whitespace')
    return v.strip()



# Ingest Models

class IngestResponse(BaseModel):
    """
    Response model for document ingestion.

    Returned after successfully processing and storing a document.
    """
    model_config = ConfigDict(
        json_schema_extra={
            "example": {
                "status": "success",
                "document_id": "doc_a1b2c3d4",
                "language": "en",
```

```
                "chunks_created": 15,
                "file_size_kb": 42.3,
                "processing_time_seconds": 2.4
            }
        }
    )

    status: str = Field(
        ...,
        description="Status of the ingestion operation",
        examples=["success"]
    )
    document_id: str = Field(
        ...,
        description="Unique identifier for the ingested document",
        examples=["doc_a1b2c3d4"]
    )
    language: str = Field(
        ...,
        description="Detected language of the document (en or ja)",
        examples=["en", "ja"]
    )
    chunks_created: int = Field(
        ...,
        description="Number of text chunks created",
        ge=0,
        examples=[15]
    )
    file_size_kb: float = Field(
        ...,
        description="Size of the uploaded file in kilobytes",
        ge=0,
        examples=[42.3]
    )
    processing_time_seconds: float = Field(
        ...,
        description="Time taken to process the document",
        ge=0,
        examples=[2.4]
    )
```

```python
# Retrieve Models

class RetrieveRequest(BaseModel):
    """
    Request model for document retrieval.

    Searches the vector database for documents semantically similar
    to the provided query.
    """
    model_config = ConfigDict(
        str_strip_whitespace=True,
        json_schema_extra={
            "example": {
                "query": "What are the latest recommendations for Type 2
diabetes management?",
                "top_k": 3,
                "min_score": 0.5
            }
        }
    )

    query: str = Field(
        ...,
        min_length=1,
        max_length=500,
        description="Search query in English or Japanese",
        examples=["What are the diabetes management guidelines?"]
    )
    top_k: int = Field(
        default=3,
        ge=1,
        le=10,
        description="Number of results to return",
        examples=[3]
    )
    min_score: float = Field(
        default=0.5,
```

```
        ge=0.0,
        le=1.0,
        description="Minimum similarity score threshold (0.0-1.0)",
        examples=[0.5]
    )

    @field_validator('query')
    @classmethod
    def query_not_empty(cls, v: str) -> str:
        """Validate query is not empty"""
        return validate_query_field(v)


class SearchResult(BaseModel):
    """
    Individual search result from vector database.

    Represents a single document chunk with similarity score.
    """
    model_config = ConfigDict(
        json_schema_extra={
            "example": {
                "text": "Type 2 diabetes management requires...",
                "similarity_score": 0.87,
                "document_id": "doc_a1b2c3d4",
                "language": "en",
                "chunk_id": 5
            }
        }
    )

    text: str = Field(
        ...,
        description="Content of the document chunk"
    )
    similarity_score: float = Field(
        ...,
        ge=0.0,
        le=1.0,
        description="Cosine similarity score (0.0-1.0)"
```

```
    )
    document_id: str = Field(
        ...,
        description="ID of the source document"
    )
    language: str = Field(
        ...,
        description="Language of the document (en or ja)"
    )
    chunk_id: int = Field(
        ...,
        ge=0,
        description="Chunk index within the document"
    )


class RetrieveResponse(BaseModel):
    """
    Response model for document retrieval.

    Contains search results and metadata about the query.
    """
    model_config = ConfigDict(
        json_schema_extra={
            "example": {
                "results": [
                    {
                        "text": "Type 2 diabetes management requires...",
                        "similarity_score": 0.87,
                        "document_id": "doc_a1b2c3d4",
                        "language": "en",
                        "chunk_id": 5
                    }
                ],
                "query_language": "en",
                "results_found": 3
            }
        }
    )
```

```
    results: List[SearchResult] = Field(
        ...,
        description="List of search results ordered by similarity"
    )
    query_language: str = Field(
        ...,
        description="Detected language of the query (en or ja)"
    )
    results_found: int = Field(
        ...,
        ge=0,
        description="Total number of results found"
    )



# Generate Models

class GenerateRequest(BaseModel):
    """
    Request model for response generation.

    Generates a contextual response based on retrieved documents.
    """
    model_config = ConfigDict(
        str_strip_whitespace=True,
        json_schema_extra={
            "example": {
                "query": "What are the dietary recommendations for
diabetes?",
                "output_language": "ja"
            }
        }
    )

    query: str = Field(
        ...,
        min_length=1,
        max_length=500,
        description="Question to answer based on retrieved documents",
```

```
        examples=["What are the dietary recommendations?"]
    )
    output_language: Optional[Literal["en", "ja"]] = Field(
        default=None,
        description="Desired output language (en or ja). If not specified,
uses query language."
    )

    @field_validator('query')
    @classmethod
    def query_not_empty(cls, v: str) -> str:
        """Validate query is not empty"""
        return validate_query_field(v)


class Source(BaseModel):
    """
    Source document reference in generated response.
    """
    model_config = ConfigDict(
        json_schema_extra={
            "example": {
                "document_id": "doc_a1b2c3d4",
                "chunk_id": 5,
                "similarity_score": 0.87,
                "language": "en"
            }
        }
    )

    document_id: str = Field(..., description="Source document ID")
    chunk_id: int = Field(..., ge=0, description="Chunk index")
    similarity_score: float = Field(..., ge=0.0, le=1.0,
description="Relevance score")
    language: str = Field(..., description="Language of source (en or ja)")


class GenerateResponse(BaseModel):
    """
    Response model for generated answers.
```

```
    Contains the AI-generated response with source citations.
    """
    model_config = ConfigDict(
        json_schema_extra={
            "example": {
                "query": "What are the dietary recommendations?",
                "response": "Based on the retrieved medical guidelines
regarding diabetes management...",
                "language": "en",
                "sources": [
                    {
                        "document_id": "doc_a1b2c3d4",
                        "chunk_id": 5,
                        "similarity_score": 0.87
                    }
                ],
                "generation_time_seconds": 1.2
            }
        }
    )

    query: str = Field(
        ...,
        description="Original query that was answered"
    )
    response: str = Field(
        ...,
        description="Generated response based on retrieved documents"
    )
    language: str = Field(
        ...,
        description="Language of the response (en or ja)"
    )
    sources: List[Source] = Field(
        ...,
        description="Source documents used to generate the response"
    )
    generation_time_seconds: float = Field(
        ...,
```

```
        ge=0,
        description="Time taken to generate the response"
    )



# Common Response Models

class HealthResponse(BaseModel):
    """Health check response model."""
    model_config = ConfigDict(
        json_schema_extra={
            "example": {
                "status": "healthy",
                "faiss_index_size": 1523,
                "total_documents": 12,
                "model_loaded": True,
                "uptime_seconds": 3600
            }
        }
    )

    status: str = Field(..., description="Health status")
    faiss_index_size: int = Field(..., ge=0, description="Number of vectors
in FAISS")
    total_documents: int = Field(..., ge=0, description="Number of
documents ingested")
    model_loaded: bool = Field(..., description="Whether embedding model is
loaded")
    uptime_seconds: int = Field(..., ge=0, description="Server uptime in
seconds")


class ErrorResponse(BaseModel):
    """Standard error response model."""
    model_config = ConfigDict(
        json_schema_extra={
            "example": {
                "error": "Invalid file format",
                "detail": "Only .txt files are supported",
```

```
                "status_code": 400
            }
        }
    )

    error: str = Field(..., description="Error type")
    detail: str = Field(..., description="Detailed error message")
    status_code: int = Field(..., description="HTTP status code")


class StatsResponse(BaseModel):
    """Admin statistics response model."""
    model_config = ConfigDict(
        json_schema_extra={
            "example": {
                "faiss": {
                    "total_vectors": 1523,
                    "total_documents": 12,
                    "embedding_dimension": 384,
                    "index_size_mb": 8.4
                },
                "uptime_seconds": 3600
            }
        }
    )

    class FAISSStats(BaseModel):
        """FAISS index statistics."""
        total_vectors: int
        total_documents: int
        embedding_dimension: int
        index_size_mb: float

    faiss: FAISSStats
    uptime_seconds: int
```

This file defines all the data structures (models) that the Healthcare Search Assistant API uses for requests and responses.

What it does:

Defines the exact format for all API inputs and outputs

Automatically validates data

Provides examples for API documentation

Ensures consistent data structure throughout the application

Main Models:

1. Document Upload (Ingest)

Request: File upload (handled separately)

Response: Shows upload success with details like:

Document ID

Detected language (English/Japanese)

Number of text chunks created

File size and processing time

2. Document Search (Retrieve)

Request: Search query with options:

Query text (required)

Number of results to return (1-10)

Minimum similarity score (0.0-1.0)

Response: Search results with:

List of matching document chunks

Similarity scores

Source document information

Query language detected

3. Answer Generation (Generate)

Request: Question to answer with optional:

Output language (English/Japanese)

Response: AI-generated answer with:

The generated response

Source citations showing where information came from

Generation time

Response language

4. Support Models

Health Check: System status, index size, uptime

Error Response: Standard error format

Admin Statistics: Detailed system metrics

Key Features:

Validation: Ensures queries aren't empty, numbers are within ranges

Documentation: Includes examples for API docs

Security: Validates all inputs before processing

Consistency: Standard formats across all endpoints

Examples:

If you search for "diabetes treatment", the API expects a specific request format and returns results in a predictable structure

If you upload a document, you get back a consistent response showing what was processed

All errors follow the same format

These models act as contracts between the frontend and backend, ensuring everyone knows exactly what data to send and what to expect back.

app/routes/generate.py:

```python
# app/routes/generate.py
"""
Generate endpoint for creating contextual responses using RAG.

Provides natural language responses based on retrieved medical documents
with support for bilingual queries and responses.
"""

import logging
import time
import uuid
from typing import List, Dict, Any
from fastapi import APIRouter, Depends, HTTPException, status
from asyncio import timeout as async_timeout

from app.middleware.auth import api_key_auth
from app.models.schemas import GenerateRequest, GenerateResponse
from app.services.embedding_service import embedding_service
from app.services.faiss_service import faiss_service
from app.services.llm_service import llm_service
from app.services.translation_service import translation_service
from app.utils.language_detector import detect_language
from app.config import settings

logger = logging.getLogger(__name__)

router = APIRouter()

# Configuration
GENERATION_TOP_K = getattr(settings, 'GENERATION_TOP_K', 5)
GENERATION_MIN_SCORE = getattr(settings, 'GENERATION_MIN_SCORE', 0.3)
GENERATION_TIMEOUT = getattr(settings, 'GENERATION_TIMEOUT', 30)


@router.post(
```

```
    "/generate",
    response_model=GenerateResponse,
    status_code=status.HTTP_200_OK,
    summary="Generate Response",
    description=(
        "Generate contextual medical responses using retrieved documents. "
        "Supports both English and Japanese queries with optional output "
        "language specification. Uses RAG (Retrieval-Augmented Generation)
"
        "to provide accurate, source-backed responses."
    ),
    responses={
        200: {
            "description": "Response generated successfully",
            "model": GenerateResponse
        },
        400: {
            "description": "Invalid request"
        },
        401: {
            "description": "Invalid API key"
        },
        429: {
            "description": "Too many requests"
        },
        500: {
            "description": "Internal server error"
        },
        503: {
            "description": "Service temporarily unavailable"
        }
    }
)
async def generate_response(
    request: GenerateRequest,
    api_key: str = Depends(api_key_auth)
) -> GenerateResponse:
    """
    Generate a contextual response using retrieved documents.
```

```
    This endpoint:
    1. Detects the query language
    2. Translates query to English if needed (for better retrieval)
    3. Generates embeddings and retrieves relevant documents
    4. Generates a structured response using retrieved context
    5. Translates response to requested output language if needed

    Args:
        request: GenerateRequest containing query and optional output
language
        api_key: API key for authentication (injected by dependency)

    Returns:
        GenerateResponse with generated response and source information

    Raises:
        HTTPException: Various status codes for different error conditions

    Example:
```python
        POST /generate
        {
            "query": "What are the diabetes management guidelines?",
            "output_language": "ja"
        }
```
    """
    # Generate request ID for tracking
    request_id = str(uuid.uuid4())[:8]
    start_time = time.time()

    logger.info(
        "Response generation request received",
        extra={
            "request_id": request_id,
            "query_preview": request.query[:100],
            "output_language": request.output_language,
            "api_key": api_key[:8] + "..."
        }
)
```

```python
try:
    # Apply timeout to entire operation
    async with async_timeout(GENERATION_TIMEOUT):
        return await _process_generation_request(
            request, request_id, start_time
        )

except TimeoutError:
    logger.error(
        "Response generation timed out",
        extra={
            "request_id": request_id,
            "timeout": GENERATION_TIMEOUT
        }
    )
    raise HTTPException(
        status_code=status.HTTP_504_GATEWAY_TIMEOUT,
        detail=f"Request timed out after {GENERATION_TIMEOUT} seconds"
    )

except HTTPException:
    # Re-raise HTTP exceptions
    raise

except Exception as e:
    logger.error(
        "Unexpected error in response generation",
        extra={
            "request_id": request_id,
            "error": str(e),
            "error_type": type(e).__name__
        },
        exc_info=True
    )
    raise HTTPException(
        status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
        detail="An unexpected error occurred. Please try again."
    )
```

```python
async def _process_generation_request(
    request: GenerateRequest,
    request_id: str,
    start_time: float
) -> GenerateResponse:
    """
    Process the generation request with detailed error handling.

    Args:
        request: The generate request
        request_id: Unique request identifier
        start_time: Request start timestamp

    Returns:
        GenerateResponse with generated content
    """
    # Step 1: Detect query language
    try:
        query_language = detect_language(request.query)
        logger.debug(
            "Query language detected",
            extra={
                "request_id": request_id,
                "detected_language": query_language
            }
        )
    except Exception as e:
        logger.warning(
            f"Language detection failed, defaulting to English: {e}",
            extra={"request_id": request_id}
        )
        query_language = "en"

    # Step 2: Determine output language
    output_language = request.output_language or query_language

    # Validate output language
    if output_language not in ["en", "ja"]:
        raise HTTPException(
```

```
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=f"Unsupported output language: {output_language}"
        )

    # Step 3: Use original query for retrieval
    query_for_retrieval = request.query

    logger.debug(
        "Using original query for retrieval",
        extra={
            "request_id": request_id,
            "query_language": query_language,
            "query_preview": request.query[:50]
        }
    )

    # Step 4: Generate query embedding
    try:
        query_embedding =
embedding_service.encode_single(query_for_retrieval)
        logger.debug(
            "Query embedding generated",
            extra={
                "request_id": request_id,
                "embedding_shape": query_embedding.shape
            }
        )
    except Exception as e:
        logger.error(
            "Embedding generation failed",
            extra={
                "request_id": request_id,
                "error": str(e)
            },
            exc_info=True
        )
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Failed to process query. Please try again."
        )
```

```
    # Step 5: Retrieve relevant documents
    try:
        retrieved_docs = faiss_service.search(
            query_embedding=query_embedding,
            top_k=GENERATION_TOP_K,
            min_score=GENERATION_MIN_SCORE
        )

        logger.debug(
            "Documents retrieved",
            extra={
                "request_id": request_id,
                "documents_found": len(retrieved_docs),
                "top_score": retrieved_docs[0].get('similarity_score', 0)
if retrieved_docs else 0
            }
        )

    except Exception as e:
        logger.error(
            "Document retrieval failed",
            extra={
                "request_id": request_id,
                "error": str(e)
            },
            exc_info=True
        )
        raise HTTPException(
            status_code=status.HTTP_503_SERVICE_UNAVAILABLE,
            detail="Document retrieval service temporarily unavailable."
        )

    # Step 6: Validate retrieved documents
    if not retrieved_docs:
        logger.warning(
            "No documents retrieved for query",
            extra={"request_id": request_id}
        )
        # LLM service will handle no results with appropriate message
```

```
    else:
        # Check quality of results
        high_quality_count = sum(
            1 for doc in retrieved_docs
            if doc.get('similarity_score', 0) >= 0.5
        )

        if high_quality_count == 0:
            logger.warning(
                "No high-quality documents found",
                extra={
                    "request_id": request_id,
                    "best_score": retrieved_docs[0].get('similarity_score',
0)
                }
            )

    # Step 7: Generate response using LLM service
    try:
        llm_response = llm_service.generate_response(
            query=request.query,  # Use original query for context
            retrieved_docs=retrieved_docs,
            language=output_language
        )

        logger.debug(
            "LLM response generated",
            extra={
                "request_id": request_id,
                "response_length": len(llm_response["response"]),
                "sources_count": len(llm_response["sources"])
            }
        )

    except Exception as e:
        logger.error(
            "LLM response generation failed",
            extra={
                "request_id": request_id,
                "error": str(e)
```

```
            },
            exc_info=True
        )
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Failed to generate response. Please try again."
        )

    # Step 8: Prepare final response
    generation_time = time.time() - start_time

    response = GenerateResponse(
        query=request.query,
        response=llm_response["response"],
        language=output_language,
        sources=llm_response["sources"],
        generation_time_seconds=round(generation_time, 3)
    )

    logger.info(
        "Response generation completed successfully",
        extra={
            "request_id": request_id,
            "query_language": query_language,
            "output_language": output_language,
            "documents_used": len(retrieved_docs),
            "generation_time": round(generation_time, 3),
            "response_length": len(response.response)
        }
    )

    return response


@router.get(
    "/generate/health",
    summary="Health Check",
    description="Check if the generate endpoint and its dependencies are
healthy"
)
```

```python
async def health_check() -> Dict[str, Any]:
    """
    Health check for generate endpoint.

    Verifies that all required services (embedding, FAISS, LLM) are
operational.

    Returns:
        Dictionary with health status of all services
    """
    health_status = {
        "endpoint": "generate",
        "status": "healthy",
        "services": {}
    }

    # Check embedding service
    try:
        embedding_health = embedding_service.health_check()
        health_status["services"]["embedding"] = embedding_health
    except Exception as e:
        health_status["services"]["embedding"] = {
            "status": "unhealthy",
            "error": str(e)
        }
        health_status["status"] = "degraded"

    # Check FAISS service
    try:
        faiss_health = faiss_service.health_check()
        health_status["services"]["faiss"] = faiss_health
    except Exception as e:
        health_status["services"]["faiss"] = {
            "status": "unhealthy",
            "error": str(e)
        }
        health_status["status"] = "degraded"

    # Check LLM service
    try:
```

```
        llm_health = llm_service.health_check()
        health_status["services"]["llm"] = llm_health
    except Exception as e:
        health_status["services"]["llm"] = {
            "status": "unhealthy",
            "error": str(e)
        }
        health_status["status"] = "degraded"

    # Check translation service
    try:
        translation_health = translation_service.health_check()
        health_status["services"]["translation"] = translation_health
    except Exception as e:
        health_status["services"]["translation"] = {
            "status": "unhealthy",
            "error": str(e)
        }
        health_status["status"] = "degraded"

    return health_status
```

This is the answer generation endpoint for the Healthcare Search Assistant. It creates AI-powered responses to medical questions using retrieved documents.

What it does:

- Takes a user's medical question and generates an intelligent answer
- Searches through medical documents to find relevant information
- Creates a well-structured response with source citations
- Supports both English and Japanese questions and answers

How it works step by step:

- Receive Question
  - User sends a question like "What are diabetes treatment guidelines?"
  - Optionally specifies output language (English/Japanese)

- Process the Request
  - Detect the language of the question
  - Determine what language to answer in
  - Convert the question to numerical vectors (embeddings)
- Search for Information
  - Search the document database for relevant medical information
  - Find the most similar document chunks
  - Filter by quality scores
- Generate Answer
  - Use AI to create a response based on the found documents
  - Include citations showing which documents were used
  - Format the response properly
- Return Result
  - Send back the generated answer with sources
  - Include processing time and metadata

Key Features:

Error Handling:

- 30-second timeout to prevent hanging requests
- Handles missing documents gracefully
- Validates all inputs
- Comprehensive error logging

Bilingual Support:

Automatically detects question language

Can answer in either English or Japanese

Uses original language for best search results

Monitoring:

Tracks every request with unique IDs

Logs processing times and success rates

Health check endpoint to verify all services are working

Security:

Requires API key authentication

Validates all inputs

Timeout protection

This endpoint combines document search with AI generation to provide accurate, source-backed medical information to healthcare professionals.

app/routes/ingest.py:

```
# app/routes/ingest.py
"""
Document ingestion endpoint for uploading and processing text documents.

Handles file validation, text chunking, embedding generation, and storage
in the vector database with support for English and Japanese documents.
"""

import time
import uuid
import re
import hashlib
import logging
from pathlib import Path
```

```python
from typing import List, Dict, Any
from asyncio import timeout as async_timeout

from fastapi import APIRouter, Depends, UploadFile, File, HTTPException,
status
import nltk
from nltk.tokenize import sent_tokenize

from app.middleware.auth import api_key_auth
from app.models.schemas import IngestResponse
from app.services.embedding_service import embedding_service
from app.services.faiss_service import faiss_service
from app.utils.language_detector import detect_language
from app.config import settings

logger = logging.getLogger(__name__)
router = APIRouter()

# Ensure NLTK data is available
try:
    nltk.data.find('tokenizers/punkt')
except LookupError:
    try:
        nltk.download('punkt', quiet=True)
        logger.info("Downloaded NLTK punkt tokenizer")
    except Exception as e:
        logger.warning(f"Failed to download NLTK data: {e}")

# Configuration
MAX_FILE_SIZE_BYTES = settings.MAX_FILE_SIZE_MB * 1024 * 1024
MAX_CHUNK_SIZE = getattr(settings, 'MAX_CHUNK_SIZE', 500)
ALLOWED_EXTENSIONS = {'.txt'}
INGEST_TIMEOUT = 60  # seconds


def sanitize_filename(filename: str) -> str:
    """
    Sanitize filename to prevent security issues.

    Args:
```

```
        filename: Original filename

    Returns:
        Sanitized filename
    """
    # Get just the filename, no path components
    filename = Path(filename).name

    # Remove any dangerous characters
    filename = re.sub(r'[^\w\s.-]', '', filename)

    # Limit length
    if len(filename) > 100:
        name, ext = filename.rsplit('.', 1) if '.' in filename else
(filename, '')
        filename = name[:95] + '.' + ext if ext else name[:100]

    return filename or "unnamed.txt"


def calculate_content_hash(content: str) -> str:
    """
    Calculate SHA-256 hash of content for duplicate detection.

    Args:
        content: Text content

    Returns:
        Hex digest of content hash
    """
    return hashlib.sha256(content.encode('utf-8')).hexdigest()


def chunk_text_by_sentences(
    text: str,
    language: str = "en",
    max_chunk_size: int = 500,
    overlap_sentences: int = 1
) -> List[str]:
    """
```

```
    Split text into chunks by sentences, maintaining semantic boundaries.

    Uses language-aware sentence splitting for English and Japanese.

    Args:
        text: Input text to chunk
        language: Language of text ('en' or 'ja')
        max_chunk_size: Target maximum characters per chunk
        overlap_sentences: Number of sentences to overlap between chunks

    Returns:
        List of text chunks

    Example:
        >>> chunks = chunk_text_by_sentences("Text here.", "en", 500)
        >>> len(chunks)
        1
    """
    # Split into sentences based on language
    if language == "ja":
        # Japanese sentence endings: 。！？
        sentences = re.split(r'(?<=[。！？])\s*', text)
        sentences = [s.strip() for s in sentences if s.strip()]
    else:
        # English - use NLTK
        try:
            sentences = sent_tokenize(text)
        except Exception as e:
            logger.warning(f"NLTK tokenization failed, using fallback:
{e}")
            # Fallback to simple splitting
            sentences = re.split(r'(?<=[.!?])\s+', text)

    if not sentences:
        return []

    chunks = []
    current_chunk = []
    current_size = 0
```

```
    for sentence in sentences:
        sentence_size = len(sentence)

        # If adding this sentence exceeds limit and we have content
        if current_size + sentence_size > max_chunk_size and current_chunk:
            # Save current chunk
            chunks.append(' '.join(current_chunk))

            # Start new chunk with overlap
            if overlap_sentences > 0 and len(current_chunk) >
overlap_sentences:
                overlap = current_chunk[-overlap_sentences:]
            else:
                overlap = []

            current_chunk = overlap + [sentence]
            current_size = sum(len(s) for s in current_chunk)
        else:
            current_chunk.append(sentence)
            current_size += sentence_size

    # Add final chunk if not empty
    if current_chunk:
        chunks.append(' '.join(current_chunk))

    return chunks


async def validate_upload_file(file: UploadFile) -> tuple[str, str]:
    """
    Validate uploaded file and return its content and sanitized filename.

    Args:
        file: Uploaded file

    Returns:
        Tuple of (file content as string, sanitized filename)

    Raises:
        HTTPException: If validation fails
```

```
    """
    # Sanitize filename
    sanitized_name = sanitize_filename(file.filename or "unnamed.txt")

    # Check file extension
    file_ext = Path(sanitized_name).suffix.lower()
    if file_ext not in ALLOWED_EXTENSIONS:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=f"Invalid file type. Only {',
'.join(ALLOWED_EXTENSIONS)} files are allowed."
        )

    # Read file content
    try:
        content = await file.read()
    except Exception as e:
        logger.error(f"Failed to read uploaded file: {e}")
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Failed to read uploaded file."
        )

    # Check file size
    if len(content) == 0:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="File is empty."
        )

    if len(content) > MAX_FILE_SIZE_BYTES:
        max_size_mb = MAX_FILE_SIZE_BYTES / (1024 * 1024)
        raise HTTPException(
            status_code=status.HTTP_413_REQUEST_ENTITY_TOO_LARGE,
            detail=f"File too large. Maximum size is {max_size_mb:.1f}MB."
        )

    # Decode and validate encoding
    try:
        text = content.decode('utf-8')
```

```
    except UnicodeDecodeError:
        try:
            # Try other common encodings
            text = content.decode('latin-1')
            logger.warning(f"File decoded as latin-1: {sanitized_name}")
        except Exception:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="File must be UTF-8 or Latin-1 encoded text."
            )

    # Validate content
    if not text.strip():
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="File contains only whitespace."
        )

    # Reset file pointer for potential reuse
    await file.seek(0)

    return text, sanitized_name


@router.post(
    "/ingest",
    response_model=IngestResponse,
    status_code=status.HTTP_201_CREATED,
    summary="Ingest Document",
    description=(
        "Upload and process a text document for storage in the vector
database. "
        "Supports English and Japanese documents. The document is
automatically "
        "chunked, embedded, and indexed for retrieval."
    ),
    responses={
        201: {
            "description": "Document ingested successfully",
            "model": IngestResponse
```

```
        },
        400: {
            "description": "Invalid file or request"
        },
        401: {
            "description": "Invalid API key"
        },
        413: {
            "description": "File too large"
        },
        429: {
            "description": "Too many requests"
        },
        500: {
            "description": "Internal server error"
        }
    }
)
async def ingest_document(
    file: UploadFile = File(..., description="Text file to ingest (.txt
only)"),
    api_key: str = Depends(api_key_auth)
) -> IngestResponse:
    """
    Ingest a document by processing, chunking, and storing it in the vector
database.


    Args:
        file: Text file to ingest (.txt format)
        api_key: API key for authentication (injected by dependency)

    Returns:
        IngestResponse with processing results including document ID

    Raises:
        HTTPException: Various status codes for different error conditions

    Example:
```bash
```
```

```
        curl -X POST "http://localhost:8000/ingest" \
          -H "X-API-Key: your-api-key" \
          -F "file=@document.txt"
```
    """
    # Generate request ID for tracking
    request_id = str(uuid.uuid4())[:8]
    start_time = time.time()

    logger.info(
        "Document ingestion request received",
        extra={
            "request_id": request_id,
            "file_name": file.filename,
            "content_type": file.content_type,
            "api_key": api_key[:8] + "..." if api_key else None
        }
)

    try:
        # Apply timeout to entire operation
        async with async_timeout(INGEST_TIMEOUT):
            return await _process_ingestion(file, request_id, start_time)

    except TimeoutError:
        logger.error(
            "Document ingestion timed out",
            extra={
                "request_id": request_id,
                "timeout": INGEST_TIMEOUT
            }
)
        raise HTTPException(
            status_code=status.HTTP_504_GATEWAY_TIMEOUT,
            detail=f"Document processing timed out after {INGEST_TIMEOUT} seconds"
)

    except HTTPException:
        # Re-raise HTTP exceptions

```
        raise

    except Exception as e:
        logger.error(
            "Unexpected error in document ingestion",
            extra={
                "request_id": request_id,
                "file_name": file.filename,
                "error": str(e),
                "error_type": type(e).__name__
            },
            exc_info=True
        )
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="An unexpected error occurred while processing the
document."
        )


async def _process_ingestion(
    file: UploadFile,
    request_id: str,
    start_time: float
) -> IngestResponse:
    """
    Process document ingestion with detailed error handling.

    Args:
        file: Uploaded file
        request_id: Unique request identifier
        start_time: Request start timestamp

    Returns:
        IngestResponse with processing results
    """
    # Step 1: Validate and read file
    try:
        text_content, sanitized_filename = await validate_upload_file(file)
        file_size_kb = len(text_content.encode('utf-8')) / 1024
```

```
        logger.debug(
            "File validated successfully",
            extra={
                "request_id": request_id,
                "file_name": sanitized_filename,
                "size_kb": round(file_size_kb, 2)
            }
        )
    except HTTPException:
        raise
    except Exception as e:
        logger.error(
            "File validation failed",
            extra={"request_id": request_id, "error": str(e)},
            exc_info=True
        )
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="File validation failed. Please check file format and
try again."
        )

    # Step 2: Detect language
    try:
        language = detect_language(text_content)
        logger.debug(
            "Language detected",
            extra={
                "request_id": request_id,
                "language": language
            }
        )
    except Exception as e:
        logger.warning(
            f"Language detection failed, defaulting to English: {e}",
            extra={"request_id": request_id}
        )
        language = "en"
```

```
    # Step 3: Calculate content hash (for duplicate detection)
    content_hash = calculate_content_hash(text_content)

    # Step 4: Chunk text
    try:
        chunks = chunk_text_by_sentences(
            text_content,
            language=language,
            max_chunk_size=MAX_CHUNK_SIZE
        )

        if not chunks:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="No meaningful chunks could be extracted from the
document."
            )

        # Validate chunks
        valid_chunks = [chunk for chunk in chunks if 10 <= len(chunk) <=
2000]

        if not valid_chunks:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="Document contains no valid text chunks."
            )

        if len(valid_chunks) < len(chunks):
            logger.warning(
                f"Filtered out {len(chunks) - len(valid_chunks)} invalid
chunks",
                extra={"request_id": request_id}
            )

        chunks = valid_chunks

        logger.debug(
            "Text chunked successfully",
            extra={
```

```
                "request_id": request_id,
                "chunk_count": len(chunks),
                "avg_chunk_size": sum(len(c) for c in chunks) / len(chunks)
            }
        )

    except HTTPException:
        raise
    except Exception as e:
        logger.error(
            "Text chunking failed",
            extra={"request_id": request_id, "error": str(e)},
            exc_info=True
        )
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Failed to process document text."
        )

    # Step 5: Generate embeddings
    try:
        embeddings = embedding_service.encode(
            chunks,
            batch_size=32,
            show_progress_bar=False
        )

        logger.debug(
            "Embeddings generated",
            extra={
                "request_id": request_id,
                "embedding_count": len(embeddings),
                "embedding_shape": embeddings.shape
            }
        )

    except Exception as e:
        logger.error(
            "Embedding generation failed",
            extra={"request_id": request_id, "error": str(e)},
```

```
            exc_info=True
        )
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Failed to generate document embeddings."
        )

    # Step 6: Generate document ID
    document_id = faiss_service.generate_document_id()

    # Step 7: Add to FAISS index
    try:
        chunks_created = faiss_service.add_document(
            document_id=document_id,
            chunks=chunks,
            embeddings=embeddings,
            language=language,
            metadata={
                "file_name": sanitized_filename,
                "original_size_kb": round(file_size_kb, 2),
                "chunk_count": len(chunks),
                "content_hash": content_hash,
                "ingested_at": time.time()
            }
        )

        logger.debug(
            "Document added to FAISS index",
            extra={
                "request_id": request_id,
                "document_id": document_id,
                "chunks_created": chunks_created
            }
        )

    except Exception as e:
        logger.error(
            "Failed to add document to FAISS index",
            extra={
                "request_id": request_id,
```

```
                "document_id": document_id,
                "error": str(e)
            },
            exc_info=True
        )
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Failed to store document in database."
        )

    # Step 8: Prepare response
    processing_time = time.time() - start_time

    response = IngestResponse(
        status="success",
        document_id=document_id,
        language=language,
        chunks_created=chunks_created,
        file_size_kb=round(file_size_kb, 2),
        processing_time_seconds=round(processing_time, 3)
    )

    logger.info(
        "Document ingestion completed successfully",
        extra={
            "request_id": request_id,
            "document_id": document_id,
            "language": language,
            "chunks_created": chunks_created,
            "processing_time": round(processing_time, 3),
            "file_name": sanitized_filename
        }
    )

    return response


@router.get(
    "/ingest/health",
    summary="Health Check",
```

```
    description="Check if the ingest endpoint and its dependencies are
healthy"
)
async def health_check() -> Dict[str, Any]:
    """
    Health check for ingest endpoint.

    Verifies that all required services are operational.

    Returns:
        Dictionary with health status of all services
    """
    health_status = {
        "endpoint": "ingest",
        "status": "healthy",
        "services": {}
    }

    # Check embedding service
    try:
        embedding_health = embedding_service.health_check()
        health_status["services"]["embedding"] = embedding_health
    except Exception as e:
        health_status["services"]["embedding"] = {
            "status": "unhealthy",
            "error": str(e)
        }
        health_status["status"] = "degraded"

    # Check FAISS service
    try:
        faiss_health = faiss_service.health_check()
        health_status["services"]["faiss"] = faiss_health
    except Exception as e:
        health_status["services"]["faiss"] = {
            "status": "unhealthy",
            "error": str(e)
        }
        health_status["status"] = "degraded"
```

```
    return health_status


@router.get(
    "/ingest/stats",
    summary="Ingestion Statistics",
    description="Get statistics about ingested documents"
)
async def get_stats(
    api_key: str = Depends(api_key_auth)
) -> Dict[str, Any]:
    """
    Get statistics about ingested documents.

    Returns:
        Dictionary with ingestion statistics
    """
    try:
        stats = faiss_service.get_statistics()
        return {
            "status": "success",
            "statistics": stats
        }
    except Exception as e:
        logger.error(f"Failed to get statistics: {e}")
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Failed to retrieve statistics"
        )
```

This is the document upload and processing endpoint for the Healthcare Search Assistant. It handles adding new medical documents to the search system.

What it does:

- Accepts text file uploads
- Processes and prepares documents for searching
- Stores documents in the vector database

Step-by-step process:

- File Validation
  - Checks file type (only .txt files allowed)
  - Verifies file size (max 10MB by default)
  - Validates text encoding (UTF-8 or Latin-1)
  - Ensures file isn't empty
  - Sanitizes filename for security
- Content Processing
  - Detects document language (English or Japanese)
  - Calculates content hash to detect duplicates
  - Splits text into meaningful chunks by sentences
  - Uses language-specific splitting rules:
    - English: Uses NLTK sentence tokenizer
    - Japanese: Splits at 。！？ characters
- AI Processing
  - Converts text chunks into numerical vectors (embeddings)
  - Uses the embedding model to create semantic representations
- Database Storage
  - Generates unique document ID
  - Stores vectors in FAISS search index
  - Saves metadata (filename, size, language, chunk count)
- Response

- Returns success with document ID and processing details
- Includes number of chunks created and processing time

Key Features:

Security & Validation:

- API key authentication required
- File size limits to prevent abuse
- Filename sanitization
- Encoding validation
- Duplicate detection via content hashing

Error Handling:

- 60-second timeout for processing
- Comprehensive error messages
- Graceful handling of corrupt files
- Detailed logging with request IDs

Bilingual Support:

- Automatic language detection
- Language-specific text processing
- Supports both English and Japanese documents

Monitoring:

- Health check endpoint
- Statistics endpoint (requires auth)
- Detailed processing logs

Performance tracking

This endpoint transforms raw text documents into searchable content that the system can use to answer medical questions. It's the foundation that populates the knowledge base.

app/routes/retrieve.py:

```python
# app/routes/retrieve.py

"""
Document retrieval endpoint for semantic search.

Searches the vector database for documents semantically similar to queries
with support for English and Japanese.
"""

import time
import uuid
import logging
from typing import Dict, Any
from asyncio import timeout as async_timeout

from fastapi import APIRouter, Depends, HTTPException, status

from app.middleware.auth import api_key_auth
from app.models.schemas import RetrieveRequest, RetrieveResponse,
SearchResult
from app.services.embedding_service import embedding_service
from app.services.faiss_service import faiss_service
from app.utils.language_detector import detect_language
from app.config import settings

logger = logging.getLogger(__name__)
router = APIRouter()

# Configuration
RETRIEVE_TIMEOUT = getattr(settings, 'RETRIEVE_TIMEOUT', 10)
DEFAULT_TOP_K = getattr(settings, 'DEFAULT_TOP_K', 3)
MAX_TOP_K = getattr(settings, 'MAX_TOP_K', 100)
```

```python
@router.post(
    "/retrieve",
    response_model=RetrieveResponse,
    status_code=status.HTTP_200_OK,
    summary="Retrieve Documents",
    description=(
        "Search for relevant documents based on semantic similarity. "
        "Supports bilingual queries (English and Japanese) with "
        "configurable result count and similarity threshold."
    ),
    responses={
        200: {
            "description": "Documents retrieved successfully",
            "model": RetrieveResponse
        },
        400: {
            "description": "Invalid request parameters"
        },
        401: {
            "description": "Invalid API key"
        },
        429: {
            "description": "Too many requests"
        },
        500: {
            "description": "Internal server error"
        }
    }
)
async def retrieve_documents(
    request: RetrieveRequest,
    api_key: str = Depends(api_key_auth)
) -> RetrieveResponse:
    """
    Retrieve relevant document chunks based on semantic similarity.

    Process:
    1. Validates query parameters
    2. Detects query language
```

```
    3. Generates query embedding
    4. Searches FAISS index for similar documents
    5. Filters results by similarity threshold
    6. Returns ranked results with metadata

    Args:
        request: RetrieveRequest containing query and search parameters
        api_key: API key for authentication (injected)

    Returns:
        RetrieveResponse with ranked search results

    Example:
```bash
        curl -X POST "http://localhost:8000/retrieve" \
          -H "X-API-Key: your-api-key" \
          -H "Content-Type: application/json" \
          -d '{"query": "diabetes management", "top_k": 3}'
```
    """
    # Generate request ID for tracking
    request_id = str(uuid.uuid4())[:8]
    start_time = time.time()

    logger.info(
        "Document retrieval request received",
        extra={
            "request_id": request_id,
            "query_preview": request.query[:100],
            "top_k": request.top_k,
            "min_score": request.min_score,
            "api_key": api_key[:8] + "..."
        }
)

    try:
        # Apply timeout to entire operation
        async with async_timeout(RETRIEVE_TIMEOUT):
            return await _process_retrieval(request, request_id,
start_time)
```

```
    except TimeoutError:
        logger.error(
            "Document retrieval timed out",
            extra={
                "request_id": request_id,
                "timeout": RETRIEVE_TIMEOUT
            }
        )
        raise HTTPException(
            status_code=status.HTTP_504_GATEWAY_TIMEOUT,
            detail=f"Request timed out after {RETRIEVE_TIMEOUT} seconds"
        )

    except HTTPException:
        # Re-raise HTTP exceptions
        raise

    except Exception as e:
        logger.error(
            "Unexpected error in document retrieval",
            extra={
                "request_id": request_id,
                "error": str(e),
                "error_type": type(e).__name__
            },
            exc_info=True
        )
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="An unexpected error occurred. Please try again."
        )


async def _process_retrieval(
    request: RetrieveRequest,
    request_id: str,
    start_time: float
) -> RetrieveResponse:
    """
```

```
    Process retrieval request with detailed error handling.

    Args:
        request: The retrieve request
        request_id: Unique request identifier
        start_time: Request start timestamp

    Returns:
        RetrieveResponse with search results
    """
    # Step 1: Validate query
    if not request.query.strip():
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Query cannot be empty or whitespace"
        )

    # Validate top_k
    if request.top_k < 1 or request.top_k > MAX_TOP_K:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=f"top_k must be between 1 and {MAX_TOP_K}"
        )

    # Step 2: Detect query language
    try:
        query_language = detect_language(request.query)
        logger.debug(
            "Query language detected",
            extra={
                "request_id": request_id,
                "language": query_language
            }
        )
    except Exception as e:
        logger.warning(
            f"Language detection failed, defaulting to English: {e}",
            extra={"request_id": request_id}
        )
        query_language = "en"
```

```python
# Step 3: Generate query embedding
try:
    # FIX: Extract first element from array
    query_embedding = embedding_service.encode([request.query])[0]

    logger.debug(
        "Query embedding generated",
        extra={
            "request_id": request_id,
            "embedding_shape": query_embedding.shape
        }
    )

except Exception as e:
    logger.error(
        "Embedding generation failed",
        extra={
            "request_id": request_id,
            "error": str(e)
        },
        exc_info=True
    )
    raise HTTPException(
        status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
        detail="Failed to process query. Please try again."
    )

# Step 4: Search FAISS index
try:
    search_results = faiss_service.search(
        query_embedding=query_embedding,
        top_k=request.top_k,
        min_score=request.min_score
    )

    logger.debug(
        "Search completed",
        extra={
            "request_id": request_id,
```

```
                "results_found": len(search_results),
                "top_score": search_results[0].get('similarity_score', 0)
if search_results else 0
            }
        )

    except Exception as e:
        logger.error(
            "FAISS search failed",
            extra={
                "request_id": request_id,
                "error": str(e)
            },
            exc_info=True
        )
        raise HTTPException(
            status_code=status.HTTP_503_SERVICE_UNAVAILABLE,
            detail="Search service temporarily unavailable."
        )

    # Step 5: Check for empty results
    if not search_results:
        logger.info(
            "No documents found matching query",
            extra={
                "request_id": request_id,
                "query": request.query,
                "min_score": request.min_score
            }
        )
        # Return empty results (not an error)
        return RetrieveResponse(
            results=[],
            query_language=query_language,
            results_found=0
        )

    # Step 6: Convert to response format
    results = []
    for result in search_results:
```

```
        try:
            search_result = SearchResult(
                text=result.get('text', ''),
                similarity_score=result.get('similarity_score', 0.0),
                document_id=result.get('doc_id', 'unknown'),
                language=result.get('language', 'en'),
                chunk_id=result.get('chunk_id', 0)
            )
            results.append(search_result)
        except Exception as e:
            logger.warning(
                f"Failed to parse search result: {e}",
                extra={"request_id": request_id}
            )
            continue

    # Step 7: Prepare response
    retrieval_time = time.time() - start_time

    response = RetrieveResponse(
        results=results,
        query_language=query_language,
        results_found=len(results)
    )

    logger.info(
        "Document retrieval completed successfully",
        extra={
            "request_id": request_id,
            "query_language": query_language,
            "results_found": len(results),
            "retrieval_time": round(retrieval_time, 3)
        }
    )

    return response


@router.get(
    "/retrieve/health",
```

```
    summary="Health Check",
    description="Check if the retrieve endpoint and its dependencies are
healthy"
)
async def health_check() -> Dict[str, Any]:
    """
    Health check for retrieve endpoint.

    Returns:
        Dictionary with health status
    """
    health_status = {
        "endpoint": "retrieve",
        "status": "healthy",
        "services": {}
    }

    # Check embedding service
    try:
        embedding_health = embedding_service.health_check()
        health_status["services"]["embedding"] = embedding_health
    except Exception as e:
        health_status["services"]["embedding"] = {
            "status": "unhealthy",
            "error": str(e)
        }
        health_status["status"] = "degraded"

    # Check FAISS service
    try:
        faiss_health = faiss_service.health_check()
        health_status["services"]["faiss"] = faiss_health
    except Exception as e:
        health_status["services"]["faiss"] = {
            "status": "unhealthy",
            "error": str(e)
        }
        health_status["status"] = "degraded"

    return health_status
```

This is the document search endpoint for the Healthcare Search Assistant. It finds relevant medical documents based on semantic similarity to a user's query.

What it does:

- Takes a search query and finds the most relevant document chunks
- Supports both English and Japanese queries
- Returns ranked results by relevance

Step-by-step process:

- Receive Query
  - User sends a search query like "diabetes treatment guidelines"
  - Can specify number of results (top_k) and minimum similarity score
- Process Query
  - Validates the query isn't empty
  - Detects query language (English or Japanese)
  - Converts the query to numerical vectors (embeddings)
- Search Database
  - Searches the FAISS vector database for similar documents
  - Uses cosine similarity to find closest matches
  - Filters results by minimum score threshold
- Return Results
  - Returns ranked list of matching document chunks
  - Includes similarity scores and source information
  - Shows detected query language and result count

Key Features:

Search Options:

- top_k: Number of results to return (1-100, default 3)
- min_score: Minimum similarity score (0.0-1.0, default 0.5)
- Results are automatically ranked by relevance

Bilingual Support:

- Automatically detects query language
- Works with both English and Japanese queries
- No translation needed for searching

Error Handling:

- 10-second timeout to prevent hanging requests
- Validates all inputs
- Handles empty results gracefully (not an error)
- Comprehensive error logging with request IDs

Performance & Monitoring:

- Tracks search performance and timing
- Health check endpoint to verify services
- Detailed logging of search patterns
- Request ID tracking for debugging

Response Format:

- List of results with text content, similarity scores, document IDs, and chunk information
- Query language detected

Total number of results found

This endpoint provides the core search functionality that powers the question-answering system by finding relevant medical information from the document database.

app/utils/language_detector.py:

```
"""
Language detection utilities for Healthcare RAG Assistant.

Provides robust language detection with fallback mechanisms for
English and Japanese text processing.
"""

import logging
from typing import Optional
from langdetect import detect, detect_langs, DetectorFactory
from langdetect.lang_detect_exception import LangDetectException

# Initialize logger
logger = logging.getLogger(__name__)

# Set seed for consistent results
DetectorFactory.seed = 0

# Supported languages
SUPPORTED_LANGUAGES = {'en', 'ja'}
DEFAULT_LANGUAGE = 'en'

# Minimum confidence threshold for language detection
MIN_CONFIDENCE = 0.7


def detect_language(
    text: str,
    fallback: str = DEFAULT_LANGUAGE,
    min_confidence: float = MIN_CONFIDENCE
) -> str:
    """
    Detect the language of input text with robust fallback mechanisms.
```

```
    Attempts to detect language using langdetect library with confidence
    scoring. Falls back to character-based detection (checking for Japanese
    characters) if primary detection fails or confidence is too low.

    Args:
        text: Text to detect language from
        fallback: Language to return if detection fails (default: 'en')
        min_confidence: Minimum confidence threshold for langdetect
(0.0-1.0)

    Returns:
        str: Language code ('en' or 'ja')

    Raises:
        ValueError: If text is None or empty

    Examples:
        >>> detect_language("Hello, how are you?")
        'en'

        >>> detect_language("こんにちは、元気ですか？")
        'ja'

        >>> detect_language("2型糖尿病の管理")
        'ja'

        >>> detect_language("Type 2 diabetes management")
        'en'

    Notes:
        - Prioritizes Japanese detection for medical terms
        - Uses character-based fallback for short texts
        - Logs warnings when fallback mechanisms are used
    """
    # Validate input
    if text is None:
        raise ValueError("Text cannot be None")

    if not text or not text.strip():
```

```
        logger.warning("Empty text provided for language detection")
        return fallback

    # Clean text for detection
    text_cleaned = text.strip()

    # For very short text, use character-based detection
    if len(text_cleaned) < 10:
        logger.debug(
            "Text too short for reliable detection, using character-based
fallback",
            extra={"text_length": len(text_cleaned)}
        )
        return _detect_by_characters(text_cleaned, fallback)

    try:
        # Try detection with confidence scoring
        detected_langs = detect_langs(text_cleaned)

        if not detected_langs:
            logger.warning("No languages detected")
            return _detect_by_characters(text_cleaned, fallback)

        # Get most probable language
        top_lang = detected_langs[0]
        detected_lang = top_lang.lang
        confidence = top_lang.prob

        logger.debug(
            "Language detected",
            extra={
                "language": detected_lang,
                "confidence": round(confidence, 3),
                "text_preview": text_cleaned[:50]
            }
        )

        # Check if detected language is supported
        if detected_lang not in SUPPORTED_LANGUAGES:
            logger.warning(
```

```
                "Unsupported language detected, using fallback",
                extra={
                    "detected_language": detected_lang,
                    "confidence": round(confidence, 3)
                }
            )
            return _detect_by_characters(text_cleaned, fallback)

        # Check confidence threshold
        if confidence < min_confidence:
            logger.warning(
                "Low confidence detection, using character-based fallback",
                extra={
                    "language": detected_lang,
                    "confidence": round(confidence, 3),
                    "threshold": min_confidence
                }
            )
            return _detect_by_characters(text_cleaned, fallback)

        return detected_lang

    except LangDetectException as e:
        logger.warning(
            "Language detection exception, using fallback",
            extra={
                "error": str(e),
                "text_preview": text_cleaned[:50]
            }
        )
        return _detect_by_characters(text_cleaned, fallback)

    except Exception as e:
        logger.error(
            "Unexpected error in language detection",
            extra={
                "error": str(e),
                "error_type": type(e).__name__
            },
            exc_info=True
```

```
        )
        return fallback


def _detect_by_characters(text: str, fallback: str = DEFAULT_LANGUAGE) ->
str:
    """
    Detect language based on character ranges (fallback method).

    Checks for presence of Japanese characters (hiragana, katakana, kanji).
    If found, returns 'ja', otherwise returns fallback language.

    Args:
        text: Text to analyze
        fallback: Language to return if no Japanese characters found

    Returns:
        str: 'ja' if Japanese characters detected, otherwise fallback

    Notes:
        This is a simple heuristic and may not work for mixed-language
text.
    """
    if _contains_japanese_chars(text):
        logger.debug(
            "Japanese characters detected",
            extra={"text_preview": text[:50]}
        )
        return 'ja'

    logger.debug(
        "No Japanese characters found, using fallback",
        extra={"fallback_language": fallback}
    )
    return fallback


def _contains_japanese_chars(text: str) -> bool:
    """
    Check if text contains Japanese characters.
```

```
    Checks for:
    - Hiragana (U+3040 - U+309F)
    - Katakana (U+30A0 - U+30FF)
    - Kanji/CJK Unified Ideographs (U+4E00 - U+9FFF)
    - Katakana Phonetic Extensions (U+31F0 - U+31FF)

    Args:
        text: Text to check

    Returns:
        bool: True if Japanese characters found, False otherwise

    Examples:
        >>> _contains_japanese_chars("hello")
        False

        >>> _contains_japanese_chars("こんにちは")
        True

        >>> _contains_japanese_chars("糖尿病")
        True
    """
    for char in text:
        code = ord(char)

        # Hiragana
        if 0x3040 <= code <= 0x309F:
            return True

        # Katakana
        if 0x30A0 <= code <= 0x30FF:
            return True

        # Katakana Phonetic Extensions
        if 0x31F0 <= code <= 0x31FF:
            return True

        # Kanji (CJK Unified Ideographs)
        if 0x4E00 <= code <= 0x9FFF:
```

```
            return True

        # CJK Compatibility Ideographs
        if 0xF900 <= code <= 0xFAFF:
            return True

        # Half-width Katakana
        if 0xFF65 <= code <= 0xFF9F:
            return True

    return False


def is_japanese(text: str) -> bool:
    """
    Check if text is primarily Japanese.

    Convenience function that returns True if detected language is
Japanese.

    Args:
        text: Text to check

    Returns:
        bool: True if Japanese, False otherwise

    Example:
        >>> is_japanese("こんにちは")
        True

        >>> is_japanese("Hello")
        False
    """
    return detect_language(text) == 'ja'


def is_english(text: str) -> bool:
    """
    Check if text is primarily English.
```

```
    Convenience function that returns True if detected language is English.

    Args:
        text: Text to check

    Returns:
        bool: True if English, False otherwise

    Example:
        >>> is_english("Hello")
        True

        >>> is_english("こんにちは")
        False
    """
    return detect_language(text) == 'en'


def get_language_name(lang_code: str) -> str:
    """
    Get full language name from code.

    Args:
        lang_code: Two-letter language code ('en' or 'ja')

    Returns:
        str: Full language name

    Example:
        >>> get_language_name('en')
        'English'

        >>> get_language_name('ja')
        'Japanese'
    """
    language_names = {
        'en': 'English',
        'ja': 'Japanese'
    }
    return language_names.get(lang_code, lang_code.upper())
```

This is the language detection utility for the Healthcare Search Assistant. It automatically identifies whether text is in English or Japanese.

What it does:

- Detects if text is English or Japanese
- Uses multiple methods for reliable detection
- Provides fallback options when detection is uncertain

How it works:

Primary Detection Method:

- Uses the langdetect library for AI-powered language detection
- Analyzes text patterns and word frequencies
- Provides confidence scores (0.0-1.0)
- Requires at least 70% confidence to trust the result

Fallback Method (for short or uncertain text):

- Checks for Japanese characters in the text
- Looks for specific character ranges:
  - Hiragana (e.g., こんにちは)
  - Katakana (e.g., アメリカ)
  - Kanji (e.g., 糖尿病)
- If Japanese characters are found, returns 'ja'
- Otherwise, defaults to English

Key Features:

Reliability:

- Handles very short text (under 10 characters) with character-based detection
- Uses confidence thresholds to avoid uncertain results
- Multiple fallback mechanisms for edge cases

Supported Languages:

- English ('en')
- Japanese ('ja')
- All other languages default to English

Error Handling:

- Handles empty or None text inputs
- Logs warnings when fallbacks are used
- Provides detailed debugging information

Helper Functions:

- is_japanese(text) - Returns True if text is Japanese
- is_english(text) - Returns True if text is English
- get_language_name('ja') - Returns "Japanese" (full language name)

Use Cases in the System:

- Determines language of uploaded documents
- Detects language of user queries
- Helps route text to appropriate processing pipelines
- Ensures correct language handling throughout the application

This utility ensures that the system properly handles both English and Japanese medical content without requiring users to specify the language manually.

app/utils/logger.py:

```python
# app/utils/logger.py
"""
Structured logging setup for the Healthcare RAG Assistant.

Provides JSON-formatted logging for production and human-readable
logs for development. Supports custom fields via the 'extra' parameter.

Usage:
   from app.utils.logger import setup_logging
   import logging

   setup_logging()
   logger = logging.getLogger(__name__)

   logger.info(
       "Document processed",
       extra={
           "doc_id": "doc_123",
           "chunks": 15,
           "duration": 2.4
       }
   )
"""

import logging
import sys
import json
import os
from datetime import datetime
from pathlib import Path
from logging.handlers import RotatingFileHandler
from typing import Optional


class JSONFormatter(logging.Formatter):
   """
   Custom JSON formatter for structured logging.

   Formats log records as JSON with timestamp, level, message,
   module, function, line number, and any additional fields
```

```
    passed via the 'extra' parameter in log calls.

    Example:
        logger.info(
            "Request completed",
            extra={
                "request_id": "abc123",
                "duration": 1.5,
                "status_code": 200
            }
        )

    Output:
        {
            "timestamp": "2025-11-04T12:34:56.789Z",
            "level": "INFO",
            "message": "Request completed",
            "module": "main",
            "function": "handle_request",
            "line": 45,
            "request_id": "abc123",
            "duration": 1.5,
            "status_code": 200
        }
    """

    # Standard logging attributes that should not be included as extra
fields
    RESERVED_ATTRS = {
        'name', 'msg', 'args', 'created', 'filename', 'funcName',
        'levelname', 'levelno', 'lineno', 'module', 'msecs', 'message',
        'pathname', 'process', 'processName', 'relativeCreated',
        'thread', 'threadName', 'exc_info', 'exc_text', 'stack_info',
        'taskName'
    }

    def format(self, record: logging.LogRecord) -> str:
        """
        Format the log record as JSON.
```

```
        Args:
            record: LogRecord instance

        Returns:
            JSON string representation of the log record
        """
        # Base log data
        log_data = {
            "timestamp": datetime.utcnow().isoformat() + "Z",
            "level": record.levelname,
            "message": record.getMessage(),
            "module": record.module,
            "function": record.funcName,
            "line": record.lineno
        }

        # Add logger name if not root
        if record.name != "root":
            log_data["logger"] = record.name

        # Add process/thread info for debugging
        if os.getenv("LOG_PROCESS_INFO", "false").lower() == "true":
            log_data["process_id"] = record.process
            log_data["thread_id"] = record.thread

        # Add all custom extra fields
        for key, value in record.__dict__.items():
            if key not in self.RESERVED_ATTRS and not key.startswith('_'):
                try:
                    # Ensure value is JSON serializable
                    json.dumps(value)
                    log_data[key] = value
                except (TypeError, ValueError):
                    # If not serializable, convert to string
                    log_data[key] = str(value)

        # Include exception info if present
        if record.exc_info:
            log_data["exception"] = self.formatException(record.exc_info)
```

```
        # Include stack info if present
        if record.stack_info:
            log_data["stack_info"] = self.formatStack(record.stack_info)

        return json.dumps(log_data, default=str)


def setup_logging(log_level: Optional[str] = None) -> logging.Logger:
    """
    Configure application logging with structured output.

    Sets up both console and file handlers with appropriate formatting
    based on environment (JSON for production, simple for development).

    Args:
        log_level: Logging level (DEBUG, INFO, WARNING, ERROR, CRITICAL).
                   If None, uses value from settings.LOG_LEVEL

    Returns:
        Configured root logger instance

    Raises:
        ValueError: If log_level is invalid

    Example:
        >>> from app.utils.logger import setup_logging
        >>> setup_logging("INFO")
        >>> logger = logging.getLogger(__name__)
        >>> logger.info("Application started")
    """
    # Get logger
    logger = logging.getLogger()

    # Prevent duplicate handlers if called multiple times
    if logger.hasHandlers():
        logger.handlers.clear()

    # Get log level from settings if not provided
    if log_level is None:
        try:
```

```
            from app.config import settings
            log_level = settings.LOG_LEVEL
        except ImportError:
            log_level = "INFO"

    # Validate log level
    log_level = log_level.upper()
    if log_level not in ["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"]:
        raise ValueError(
            f"Invalid log level: {log_level}. "
            "Must be one of: DEBUG, INFO, WARNING, ERROR, CRITICAL"
        )

    # Set log level
    logger.setLevel(getattr(logging, log_level))

    # Create logs directory
    logs_dir = Path("logs")
    try:
        logs_dir.mkdir(exist_ok=True)
    except OSError as e:
        print(f"Warning: Could not create logs directory: {e}",
file=sys.stderr)


    # Console Handler

    console_handler = logging.StreamHandler(sys.stdout)
    console_handler.setLevel(logging.DEBUG)

    # Determine environment
    try:
        from app.config import settings
        env = settings.ENV
    except (ImportError, AttributeError):
        env = os.getenv("ENV", "development")

    # Use JSON format in production, simple format in development
    if env == "production":
        console_handler.setFormatter(JSONFormatter())
```

```
else:
    # Human-readable format for development
    console_handler.setFormatter(
        logging.Formatter(
            fmt='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
            datefmt='%Y-%m-%d %H:%M:%S'
        )
    )

logger.addHandler(console_handler)


# File Handler (with rotation)

try:
    file_handler = RotatingFileHandler(
        'logs/app.log',
        maxBytes=10 * 1024 * 1024,
        backupCount=5,
        encoding='utf-8'
    )
    file_handler.setLevel(logging.DEBUG)

    # Always use JSON format for file logs (easy to parse)
    file_handler.setFormatter(JSONFormatter())

    logger.addHandler(file_handler)
except OSError as e:
    logger.warning(f"Could not set up file logging: {e}")

# ========================================
# Configure third-party loggers
# ========================================
# Reduce noise from third-party libraries
logging.getLogger("urllib3").setLevel(logging.WARNING)
logging.getLogger("filelock").setLevel(logging.WARNING)
logging.getLogger("transformers").setLevel(logging.WARNING)
logging.getLogger("sentence_transformers").setLevel(logging.WARNING)
logging.getLogger("httpx").setLevel(logging.WARNING)
logging.getLogger("httpcore").setLevel(logging.WARNING)
```

```
    # Log startup message
    logger.info(
        "Logging configured",
        extra={
            "log_level": log_level,
            "environment": env,
            "console_format": "json" if env == "production" else "simple",
            "file_logging": True
        }
    )

    return logger


def get_logger(name: str) -> logging.Logger:
    """
    Get a logger instance with the given name.

    Convenience function for getting module-specific loggers.

    Args:
        name: Logger name (typically __name__)

    Returns:
        Logger instance

    Example:
        >>> from app.utils.logger import get_logger
        >>> logger = get_logger(__name__)
        >>> logger.info("Module initialized")
    """
    return logging.getLogger(name)



# Context Managers for Request Logging

from contextlib import contextmanager
import time
```

```python
@contextmanager
def log_execution_time(logger: logging.Logger, operation: str,
**extra_fields):
    """
    Context manager to log execution time of an operation.

    Args:
        logger: Logger instance to use
        operation: Description of the operation
        **extra_fields: Additional fields to include in log

    Example:
        >>> logger = get_logger(__name__)
        >>> with log_execution_time(logger, "document_processing",
doc_id="doc_123"):
        >>>     process_document()
    """
    start_time = time.time()

    logger.info(
        f"{operation} started",
        extra=extra_fields
    )

    try:
        yield
    except Exception as e:
        duration = time.time() - start_time
        logger.error(
            f"{operation} failed",
            extra={
                **extra_fields,
                "duration_seconds": round(duration, 3),
                "error": str(e)
            },
            exc_info=True
        )
        raise
    else:
```

```
        duration = time.time() - start_time
        logger.info(
            f"{operation} completed",
            extra={
                **extra_fields,
                "duration_seconds": round(duration, 3)
            }
        )
```

This is the logging system for the Healthcare Search Assistant. It provides structured, organized logging for monitoring and debugging the application.

What it does:

Creates consistent log formats across the entire application

Supports both development (human-readable) and production (JSON) formats

Handles log rotation to prevent disk space issues

Adds useful context to log messages

Key Features:

Two Log Formats:

Development: Easy-to-read text format with timestamps

Production: JSON format for machine parsing and log analysis tools

Multiple Outputs:

Console: Shows logs in terminal during development

File: Saves logs to logs/app.log with rotation

Max 10MB per file

Keeps 5 backup files

Prevents disk space issues

Automatic Setup:

Detects environment (development/production)

Sets appropriate log level from configuration

Configures third-party libraries to reduce noise

Creates logs directory if needed

Helper Functions:

setup_logging() - Initializes the logging system

get_logger(__name__) - Gets a logger for any module

log_execution_time() - Times how long operations take

Use Cases:

Track API request flow with request IDs

Monitor processing times for performance

Debug errors with full context

Audit document processing steps

Health monitoring and alerting

This logging system provides the visibility needed to monitor the healthcare application's operations, troubleshoot issues, and understand system performance.

app/service/embedding_service.py:

```
# app/services/embedding_service.py

import numpy as np
from typing import List, Union
from sentence_transformers import SentenceTransformer
import logging
from app.config import settings

logger = logging.getLogger(__name__)
```

```python
class EmbeddingService:
    """
    Service for generating multilingual embeddings using
sentence-transformers.
    Handles embedding generation for both English and Japanese text.
    """

    def __init__(self, model_name: str = None):
        """
        Initialize the embedding service.

        Args:
            model_name: Name of the sentence-transformers model to use.
                        Defaults to settings.EMBEDDING_MODEL
        """
        self.model_name = model_name or settings.EMBEDDING_MODEL
        self.model = None
        self.embedding_dim = settings.EMBEDDING_DIM
        self._model_loaded = False

        logger.info(f"EmbeddingService initialized with model:
{self.model_name}")

    def _load_model(self):
        """
        Lazy load the embedding model to avoid slow startup.
        """
        if self._model_loaded:
            return

        try:
            logger.info(f"Loading embedding model: {self.model_name}")
            self.model = SentenceTransformer(self.model_name)
            self._model_loaded = True
            logger.info("Embedding model loaded successfully")

            # Verify embedding dimension
            test_embedding = self.model.encode(["test"])
            actual_dim = test_embedding.shape[1]
```

```
            if actual_dim != self.embedding_dim:
                logger.warning(
                    f"Model embedding dimension ({actual_dim}) doesn't
match "
                    f"configured dimension ({self.embedding_dim}). Updating
config."
                )
                self.embedding_dim = actual_dim

        except Exception as e:
            logger.error(f"Failed to load embedding model: {str(e)}")
            raise RuntimeError(f"Could not load embedding model: {str(e)}")

    def encode(
        self,
        texts: Union[str, List[str]],
        batch_size: int = 32,
        normalize_embeddings: bool = True,
        show_progress_bar: bool = False
    ) -> np.ndarray:
        """
        Generate embeddings for input text(s).

        Args:
            texts: Single text string or list of text strings
            batch_size: Number of texts to process in parallel
            normalize_embeddings: Whether to L2 normalize embeddings
            show_progress_bar: Whether to show progress bar for large
batches

        Returns:
            numpy array of shape (len(texts), embedding_dim) containing
embeddings

        Raises:
            RuntimeError: If model fails to load or encode
            ValueError: If input texts are invalid
        """
        if not texts:
            raise ValueError("Texts cannot be empty")
```

```
        # Convert single text to list
        if isinstance(texts, str):
            texts = [texts]

        if not all(isinstance(text, str) for text in texts):
            raise ValueError("All texts must be strings")

        # Filter out empty strings
        valid_texts = [text for text in texts if text.strip()]
        if not valid_texts:
            raise ValueError("No valid non-empty texts provided")

        if len(valid_texts) != len(texts):
            logger.warning(f"Filtered out {len(texts) - len(valid_texts)}
empty texts")

        try:
            self._load_model()

            logger.debug(
                f"Generating embeddings for {len(valid_texts)} texts, "
                f"batch_size={batch_size}"
            )

            embeddings = self.model.encode(
                valid_texts,
                batch_size=batch_size,
                show_progress_bar=show_progress_bar,
                normalize_embeddings=normalize_embeddings,
                convert_to_numpy=True
            )

            logger.debug(
                f"Generated embeddings shape: {embeddings.shape}, "
                f"dtype: {embeddings.dtype}"
            )

            return embeddings
```

```
        except Exception as e:
            logger.error(
                f"Embedding generation failed for {len(valid_texts)} texts:
{str(e)}",
                extra={"error": str(e), "text_count": len(valid_texts)}
            )
            raise RuntimeError(f"Embedding generation failed: {str(e)}")

    def encode_single(self, text: str) -> np.ndarray:
        """
        Generate embedding for a single text.

        Args:
            text: Input text string

        Returns:
            numpy array of shape (embedding_dim,) containing the embedding
        """
        embeddings = self.encode([text])
        return embeddings[0]

    def get_embedding_dimension(self) -> int:
        """
        Get the embedding dimension of the model.

        Returns:
            Integer representing embedding dimension
        """
        self._load_model()
        return self.embedding_dim

    def get_model_info(self) -> dict:
        """
        Get information about the loaded model.

        Returns:
            Dictionary containing model information
        """
        self._load_model()
```

```
        return {
            "model_name": self.model_name,
            "embedding_dimension": self.embedding_dim,
            "model_loaded": self._model_loaded,
            "max_seq_length": self.model.max_seq_length if self.model else
None
        }

    def health_check(self) -> dict:
        """
        Perform health check on the embedding service.

        Returns:
            Dictionary with health status and metrics
        """
        try:
            self._load_model()

            # Test with a simple embedding
            test_text = "Health check"
            embedding = self.encode_single(test_text)

            return {
                "status": "healthy",
                "model_loaded": self._model_loaded,
                "embedding_dimension": self.embedding_dim,
                "test_embedding_shape": embedding.shape,
                "test_embedding_norm": float(np.linalg.norm(embedding))
            }

        except Exception as e:
            return {
                "status": "unhealthy",
                "model_loaded": self._model_loaded,
                "error": str(e)
            }


# Global instance for easy dependency injection
embedding_service = EmbeddingService()
```

```
# For testing purposes
if __name__ == "__main__":
    # Quick test
    service = EmbeddingService()

    # Test single text
    embedding = service.encode_single("Hello, world!")
    print(f"Single embedding shape: {embedding.shape}")

    # Test multiple texts
    texts = ["Hello world", "こんにちは世界", "Medical diagnosis and
treatment"]
    embeddings = service.encode(texts)
    print(f"Batch embeddings shape: {embeddings.shape}")

    # Test model info
    info = service.get_model_info()
    print(f"Model info: {info}")

    # Test health check
    health = service.health_check()
    print(f"Health check: {health}")
```

This is the embedding service that converts text into numerical vectors for the Healthcare Search Assistant.

What it does:

- Transforms medical text (English or Japanese) into mathematical vectors
- Enables semantic search by converting text to numbers that computers can compare
- Uses AI models to understand the meaning behind words

Key Features:

Model Management:

- Uses the paraphrase-multilingual-MiniLM-L12-v2 model by default

- Supports both English and Japanese text
- Lazy loading - only loads the model when needed (saves startup time)
- Automatically verifies embedding dimensions match configuration

Core Functions:

- encode(texts) - Convert text to vectors
  - Handles single texts or lists of texts
  - Processes in batches for efficiency
  - Filters out empty text automatically
  - Returns numpy arrays of shape [number_of_texts, 384] (384-dimensional vectors)
- encode_single(text) - Convenience for single text
  - Returns a single vector of shape [384]

Technical Details:

- Embedding Dimension: 384 numbers per text (configurable)
- Normalization: Automatically scales vectors to unit length for better similarity comparison
- Batch Processing: Default 32 texts at once for optimal performance
- Error Handling: Comprehensive validation and error reporting

Quality Control:

- Validates all input texts are non-empty strings
- Warns about empty texts being filtered out
- Logs embedding generation details for monitoring
- Health check verifies the service is working properly

Health Monitoring:

health_check() returns service status and test results

get_model_info() shows model details and capabilities

Automatic dimension verification on startup

app/service/faiss.service.py:

```
"""
FAISS service for vector similarity search with persistent storage.

Provides robust vector index management with automatic persistence,
metadata tracking, and support for document chunking.
"""

import faiss
import json
import numpy as np
import time
import uuid
import hashlib
import threading
from pathlib import Path
from typing import List, Dict, Optional, Tuple
import logging
from app.config import settings

logger = logging.getLogger(__name__)


class FAISSService:
    """
    Service for managing FAISS vector index with persistent storage.
    """

    def __init__(self, index_dir: str = None):
        """
        Initialize FAISS service with persistent storage.
```

```
        Args:
            index_dir: Directory for storing FAISS index and metadata
        """
        self.index_dir = Path(index_dir or settings.FAISS_INDEX_DIR)
        self.index_dir.mkdir(parents=True, exist_ok=True)

        # File paths
        self.index_path = self.index_dir / "index.faiss"
        self.metadata_path = self.index_dir / "metadata.json"
        self.config_path = self.index_dir / "config.json"

        # FAISS index and metadata
        self.index = None
        self.metadata = {}
        self.doc_counter = 0
        self.embedding_dim = settings.EMBEDDING_DIM

        # Statistics
        self.total_vectors = 0
        self.total_documents = 0

        # Thread safety
        self._lock = threading.Lock()

        # Load or initialize index
        self._load_or_initialize()

        logger.info(
            "FAISSService initialized",
            extra={
                "total_vectors": self.total_vectors,
                "total_documents": self.total_documents,
                "index_dir": str(self.index_dir)
            }
        )

    def _load_or_initialize(self):
        """Load existing FAISS index or create new one."""
        try:
            if self.index_path.exists() and self.metadata_path.exists():
```

```
                logger.info("Loading existing FAISS index from disk")

                # Load FAISS index
                self.index = faiss.read_index(str(self.index_path))

                # Verify index type and dimension
                if self.index.d != self.embedding_dim:
                    logger.error(
                        f"Index dimension mismatch: expected
{self.embedding_dim}, "
                        f"got {self.index.d}"
                    )
                    raise ValueError("Index dimension mismatch")

                # Load metadata
                with open(self.metadata_path, 'r', encoding='utf-8') as f:
                    self.metadata = json.load(f)

                # Load configuration
                if self.config_path.exists():
                    with open(self.config_path, 'r') as f:
                        config = json.load(f)
                    self.doc_counter = config.get('doc_counter', 0)
                    self.total_vectors = config.get('total_vectors',
self.index.ntotal)
                    self.total_documents = config.get('total_documents', 0)
                else:
                    # Backward compatibility
                    self.total_vectors = self.index.ntotal
                    self._recalculate_document_count()

                logger.info(
                    "FAISS index loaded successfully",
                    extra={
                        "vectors": self.total_vectors,
                        "documents": self.total_documents
                    }
                )

            else:
```

```
                logger.info("No existing index found, creating new FAISS
index")
                self._create_new_index()

        except Exception as e:
            logger.error(
                "Failed to load FAISS index",
                extra={"error": str(e)},
                exc_info=True
            )
            logger.info("Creating new index due to loading failure")
            self._create_new_index()

    def _create_new_index(self):
        """
        Create a new FAISS index.

        Uses IndexFlatIP (inner product) which computes dot product.
        After L2 normalization, dot product equals cosine similarity.
        """
        try:
            # Use IndexFlatIP for cosine similarity (after normalization)
            self.index = faiss.IndexFlatIP(self.embedding_dim)
            self.metadata = {}
            self.doc_counter = 0
            self.total_vectors = 0
            self.total_documents = 0

            # Save initial state
            self._save_index()
            logger.info("Created new FAISS index (IndexFlatIP)")

        except Exception as e:
            logger.error(
                "Failed to create new FAISS index",
                extra={"error": str(e)},
                exc_info=True
            )
            raise RuntimeError(f"Could not create FAISS index: {str(e)}")
```

```python
    def generate_document_id(self) -> str:
        """
        Generate a unique document ID.

        Returns:
            String document ID in format "doc_XXXXXXXX"
        """
        data = f"{self.doc_counter}_{time.time()}_{uuid.uuid4()}"
        hash_part = hashlib.md5(data.encode()).hexdigest()[:8]
        return f"doc_{hash_part}"

    def add_document(
        self,
        document_id: str,
        chunks: List[str],
        embeddings: np.ndarray,
        language: str,
        metadata: Optional[Dict] = None
    ) -> int:
        """
        Add a document to the FAISS index with immediate persistence.

        Args:
            document_id: Unique identifier for the document
            chunks: List of text chunks from the document
            embeddings: numpy array of embeddings for each chunk
            language: Language of the document ('en' or 'ja')
            metadata: Additional metadata for the document

        Returns:
            Number of chunks added

        Raises:
            ValueError: If input validation fails
            RuntimeError: If FAISS operation fails
        """
        # Input validation
        if not document_id or not chunks or len(embeddings) == 0:
            raise ValueError("Document ID, chunks, and embeddings cannot be
empty")
```

```
        # Ensure embeddings is 2D
        if len(embeddings.shape) == 1:
            embeddings = embeddings.reshape(1, -1)

        if len(embeddings.shape) != 2:
            raise ValueError(
                f"Embeddings must be 1D or 2D array, got shape
{embeddings.shape}"
            )

        if len(chunks) != embeddings.shape[0]:
            raise ValueError(
                f"Number of chunks ({len(chunks)}) must match number of
embeddings "
                f"({embeddings.shape[0]})"
            )

        if embeddings.shape[1] != self.embedding_dim:
            raise ValueError(
                f"Embedding dimension {embeddings.shape[1]} does not match
"
                f"index dimension {self.embedding_dim}"
            )

        with self._lock:
            try:
                start_time = time.time()

                # Check if document already exists
                existing_chunks = self.get_document_chunks(document_id)
                is_new_document = len(existing_chunks) == 0

                if not is_new_document:
                    logger.warning(
                        f"Document {document_id} already exists, adding
more chunks",
                        extra={"document_id": document_id}
                    )
```

```
                start_idx = self.index.ntotal

                # Normalize embeddings for cosine similarity
                embeddings = embeddings.astype('float32')
                faiss.normalize_L2(embeddings)

                # Add to FAISS index
                self.index.add(embeddings)

                # Store metadata for each chunk
                for i, (chunk, embedding) in enumerate(zip(chunks,
embeddings)):
                    vector_idx = start_idx + i
                    self.metadata[str(vector_idx)] = {
                        "doc_id": document_id,
                        "chunk_id": i,
                        "text": chunk,
                        "language": language,
                        "vector_index": vector_idx,
                        "timestamp": time.time(),
                        "embedding_norm": float(np.linalg.norm(embedding)),
                        "deleted": False,
                        **({} if metadata is None else metadata)
                    }

                chunks_added = len(chunks)
                self.total_vectors += chunks_added

                if is_new_document:
                    self.total_documents += 1
                    self.doc_counter += 1

                # Persist immediately to prevent data loss
                self._save_index()

                processing_time = time.time() - start_time
                logger.info(
                    "Document added to FAISS index",
                    extra={
                        "document_id": document_id,
```

```
                        "chunks_added": chunks_added,
                        "language": language,
                        "processing_time": round(processing_time, 3),
                        "total_vectors": self.total_vectors,
                        "total_documents": self.total_documents,
                        "is_new": is_new_document
                    }
                )

                return chunks_added

            except Exception as e:
                logger.error(
                    "Failed to add document to FAISS index",
                    extra={
                        "document_id": document_id,
                        "error": str(e)
                    },
                    exc_info=True
                )
                raise RuntimeError(f"Failed to add document to index:
{str(e)}")

    def search(
        self,
        query_embedding: np.ndarray,
        top_k: int = 3,
        min_score: float = 0.5,
        language_filter: Optional[str] = None
    ) -> List[Dict]:
        """
        Search for similar documents in the FAISS index.

        Args:
            query_embedding: Query embedding vector (1D or 2D)
            top_k: Number of top results to return
            min_score: Minimum similarity score (0-1)
            language_filter: Filter results by language ('en' or 'ja')

        Returns:
```

```
            List of search results with metadata and similarity scores

        Example:
            >>> results = service.search(embedding, top_k=3, min_score=0.7)
            >>> for result in results:
            >>>     print(f"{result['text']}:
{result['similarity_score']:.3f}")
        """
        if self.index.ntotal == 0:
            logger.debug("Index is empty, returning no results")
            return []

        if top_k <= 0:
            raise ValueError("top_k must be positive")

        # Ensure query_embedding is 2D
        if len(query_embedding.shape) == 1:
            query_embedding = query_embedding.reshape(1, -1)

        if query_embedding.shape[0] != 1:
            raise ValueError(
                f"Only single query supported, got
{query_embedding.shape[0]} queries. "
                "Use search_batch() for multiple queries."
            )

        if query_embedding.shape[1] != self.embedding_dim:
            raise ValueError(
                f"Query embedding dimension {query_embedding.shape[1]} does
not match "
                f"index dimension {self.embedding_dim}"
            )

        with self._lock:
            try:
                # Normalize query embedding
                query_embedding = query_embedding.astype('float32')
                faiss.normalize_L2(query_embedding)
```

```
                # Search FAISS index (search more than needed for
filtering)
                search_k = min(top_k * 3, self.index.ntotal)
                scores, indices = self.index.search(query_embedding,
search_k)

                results = []
                for score, idx in zip(scores[0], indices[0]):
                    if idx == -1:  # No result
                        continue

                    metadata = self.metadata.get(str(idx))
                    if not metadata:
                        continue

                    # Skip deleted chunks
                    if metadata.get('deleted', False):
                        continue

                    # Apply language filter if specified
                    if language_filter and metadata.get('language') !=
language_filter:
                        continue

                    # Score is already cosine similarity (using
IndexFlatIP)
                    similarity_score = float(score)

                    # Clamp to [0, 1] due to numerical precision
                    similarity_score = max(0.0, min(1.0, similarity_score))

                    if similarity_score >= min_score:
                        results.append({
                            **metadata,
                            "similarity_score": similarity_score
                        })

                # Sort by similarity score and return top_k
                results.sort(key=lambda x: x['similarity_score'],
reverse=True)
```

```
            final_results = results[:top_k]

            logger.debug(
                "FAISS search completed",
                extra={
                    "results_found": len(final_results),
                    "min_score": min_score,
                    "top_k": top_k,
                    "language_filter": language_filter
                }
            )

            return final_results

        except Exception as e:
            logger.error(
                "FAISS search failed",
                extra={"error": str(e)},
                exc_info=True
            )
            raise RuntimeError(f"Search failed: {str(e)}")

def get_document_chunks(self, document_id: str) -> List[Dict]:
    """
    Retrieve all chunks for a specific document.

    Args:
        document_id: Document ID to retrieve

    Returns:
        List of chunks with metadata, sorted by chunk_id
    """
    chunks = []
    for vector_idx, metadata in self.metadata.items():
        if (metadata.get('doc_id') == document_id and
            not metadata.get('deleted', False)):
            chunks.append(metadata)

    # Sort by chunk_id
    chunks.sort(key=lambda x: x.get('chunk_id', 0))
```

```
        return chunks

    def delete_document(self, document_id: str) -> bool:
        """
        Delete all chunks of a document from the index.

        Args:
            document_id: Document ID to delete

        Returns:
            True if document was found and deleted, False otherwise
        """
        with self._lock:
            chunks_to_delete = []

            # Find all chunks for this document
            for vector_idx, metadata in list(self.metadata.items()):
                if (metadata.get('doc_id') == document_id and
                    not metadata.get('deleted', False)):
                    chunks_to_delete.append(vector_idx)

            if not chunks_to_delete:
                logger.warning(
                    f"Document {document_id} not found for deletion",
                    extra={"document_id": document_id}
                )
                return False

            # Mark chunks as deleted in metadata
            for vector_idx in chunks_to_delete:
                if vector_idx in self.metadata:
                    self.metadata[vector_idx]['deleted'] = True
                    self.metadata[vector_idx]['deleted_at'] = time.time()

            # Update document count
            self._recalculate_document_count()

            self._save_index()

            logger.info(
```

```
                "Document marked as deleted",
                extra={
                    "document_id": document_id,
                    "chunks_deleted": len(chunks_to_delete),
                    "total_documents": self.total_documents
                }
            )

            return True

    def compact_index(self):
        """
        Rebuild index without deleted documents.

        This removes deleted vectors from memory and rebuilds the index.
        This is an expensive operation and should be done periodically.

        Returns:
            Dictionary with compaction statistics
        """
        with self._lock:
            logger.info("Starting index compaction...")
            start_time = time.time()

            # Collect non-deleted entries
            valid_vectors = []
            valid_metadata = {}

            for idx in range(self.index.ntotal):
                metadata = self.metadata.get(str(idx))
                if metadata and not metadata.get('deleted', False):
                    # Get vector from index
                    vector = self.index.reconstruct(int(idx))
                    valid_vectors.append(vector)

                    # Store with new index
                    new_idx = len(valid_vectors) - 1
                    valid_metadata[str(new_idx)] = {
                        **metadata,
                        "vector_index": new_idx
```

```
                }

        # Create new index
        new_index = faiss.IndexFlatIP(self.embedding_dim)
        if valid_vectors:
            vectors_array = np.array(valid_vectors).astype('float32')
            new_index.add(vectors_array)

        # Store old stats
        old_vector_count = self.index.ntotal
        old_doc_count = self.total_documents

        # Replace old index
        self.index = new_index
        self.metadata = valid_metadata
        self.total_vectors = len(valid_vectors)
        self._recalculate_document_count()

        self._save_index()

        duration = time.time() - start_time

        stats = {
            "vectors_before": old_vector_count,
            "vectors_after": self.total_vectors,
            "vectors_removed": old_vector_count - self.total_vectors,
            "documents_before": old_doc_count,
            "documents_after": self.total_documents,
            "duration_seconds": round(duration, 2)
        }

        logger.info(
            "Index compaction completed",
            extra=stats
        )

        return stats

    def _recalculate_document_count(self):
        """Recalculate total_documents from metadata."""
```

```
        doc_ids = set()
        for metadata in self.metadata.values():
            if not metadata.get('deleted', False):
                doc_ids.add(metadata['doc_id'])
        self.total_documents = len(doc_ids)

    def _save_index(self):
        """Persist FAISS index, metadata, and configuration to disk."""
        try:
            # Save FAISS index
            faiss.write_index(self.index, str(self.index_path))

            # Save metadata
            with open(self.metadata_path, 'w', encoding='utf-8') as f:
                json.dump(self.metadata, f, ensure_ascii=False, indent=2)

            # Save configuration
            config = {
                "doc_counter": self.doc_counter,
                "total_vectors": self.total_vectors,
                "total_documents": self.total_documents,
                "embedding_dim": self.embedding_dim,
                "last_updated": time.time(),
                "index_type": "IndexFlatIP"
            }

            with open(self.config_path, 'w') as f:
                json.dump(config, f, indent=2)

            logger.debug("FAISS index persisted to disk")

        except Exception as e:
            logger.error(
                "Failed to persist FAISS index",
                extra={"error": str(e)},
                exc_info=True
            )
            raise RuntimeError(f"Index persistence failed: {str(e)}")

    def get_statistics(self) -> Dict:
```

```
        """Get statistics about the FAISS index."""
        with self._lock:
            deleted_count = sum(
                1 for m in self.metadata.values()
                if m.get('deleted', False)
            )

            return {
                "total_vectors": self.total_vectors,
                "total_documents": self.total_documents,
                "deleted_vectors": deleted_count,
                "active_vectors": self.total_vectors - deleted_count,
                "embedding_dimension": self.embedding_dim,
                "index_size": self.index.ntotal,
                "doc_counter": self.doc_counter,
                "index_type": "IndexFlatIP"
            }

    def health_check(self) -> Dict:
        """Perform health check on the FAISS service."""
        try:
            # Test search with random vector
            test_vector = np.random.rand(1,
self.embedding_dim).astype('float32')
            results = self.search(test_vector, top_k=1, min_score=0)

            return {
                "status": "healthy",
                "index_loaded": True,
                "total_vectors": self.total_vectors,
                "total_documents": self.total_documents,
                "embedding_dimension": self.embedding_dim,
                "test_search_works": True
            }

        except Exception as e:
            return {
                "status": "unhealthy",
                "index_loaded": False,
                "error": str(e)
```

```
            }

    def clear_index(self):
        """Clear the entire FAISS index (for testing purposes)."""
        with self._lock:
            logger.warning("Clearing entire FAISS index")
            self._create_new_index()


# Global instance for dependency injection
faiss_service = FAISSService()
```

This is the vector database service for the Healthcare Search Assistant. It's the core search engine that stores and retrieves medical documents using AI-powered similarity search.

What it does:

Stores document vectors for fast similarity search

Finds the most relevant medical information for any query

Manages document metadata and chunking

Provides persistent storage so data survives restarts

Key Components:

Storage System:

FAISS Index: Stores vectors for fast similarity search

Metadata: Stores document text, IDs, languages, and other info

Configuration: Tracks document counts and settings

Automatic Persistence: Saves everything to disk automatically

Core Functions:

add_document() - Add documents to search index

Takes document text chunks and their vector embeddings

- Stores vectors in FAISS and metadata in JSON
- Generates unique document IDs
- Handles both English and Japanese content

search() - Find similar documents

- Takes a query vector and finds most similar documents
- Uses cosine similarity (0.0-1.0) to rank results
- Supports language filtering and score thresholds
- Returns results with text, scores, and source info

Document Management

- get_document_chunks() - Retrieve all parts of a document
- delete_document() - Remove documents (soft delete)
- compact_index() - Clean up deleted content

Technical Details:

Search Algorithm:

- Uses FAISS IndexFlatIP (Inner Product) for cosine similarity
- All vectors are L2-normalized so dot product = cosine similarity
- Returns scores from 0.0 (no similarity) to 1.0 (perfect match)

Data Organization:

- Each document is split into chunks (sentences/paragraphs)
- Each chunk gets its own vector and metadata
- Documents can have multiple chunks but one ID

Performance Features:

Thread-safe: Multiple requests won't corrupt data

Lazy Loading: Only loads from disk when needed

Automatic Recovery: Creates new index if loading fails

Efficient Storage: Compaction removes deleted items

Monitoring & Maintenance:

get_statistics() - Shows vector counts, document counts, storage info

health_check() - Verifies service is working

Automatic persistence on every change

Periodic compaction to optimize performance

app/service/llm__service.py:

```python
# app/services/llm_service.py
"""
Mock LLM service for generating structured medical responses.

Provides template-based response generation with medical domain knowledge
for both English and Japanese languages.
"""

import logging
import re
import time
from typing import List, Dict, Optional, Set
import nltk
from app.config import settings
from app.utils.language_detector import detect_language
from app.services.translation_service import translation_service

logger = logging.getLogger(__name__)

# Download NLTK data if not present
```

```
try:
    nltk.data.find('tokenizers/punkt')
except LookupError:
    nltk.download('punkt', quiet=True)


class LLMService:
    """
    Service for generating structured medical responses using retrieved
documents.

    Provides mock LLM functionality with template-based response
generation,
    medical keyword extraction, and bilingual support.

    Features:
        - Template-based response generation
        - Medical domain knowledge
        - Bilingual support (English/Japanese)
        - Source tracking
        - Medical disclaimers

    Example:
        >>> service = LLMService()
        >>> response = service.generate_response(query, documents, "en")
        >>> print(response["response"])
    """

    def __init__(self):
        """Initialize LLM service with response templates and medical
patterns."""

        self.response_templates = {
            "en": {
                "introduction": "Based on the retrieved medical guidelines
regarding {topic}:",
                "findings_header": "Key Findings:",
                "recommendations_header": "Recommendations:",
```

```
                "disclaimer": "\n⚠️ Disclaimer: This information is
AI-generated based on retrieved documents. Please consult healthcare
professionals for medical advice.",
                "no_results": "I couldn't find specific information about
'{topic}' in the available medical guidelines. Please consult a healthcare
provider for personalized medical advice.",
                "summary_template": "The guidelines indicate that
{summary}",
                "error": "I apologize, but an error occurred while
generating the response. Please try again later."
            },
            "ja": {
                "introduction": "{topic}に関する医療ガイドラインによると：",
                "findings_header": "主な知見：",
                "recommendations_header": "推奨事項：",
                "disclaimer": "\n⚠️ 免責事項：この情報は取得した文書に基づくAI生成
です。医療アドバイスについては医療専門家にご相談ください。",
                "no_results": "利用可能な医療ガイドラインに'{topic}'に関する具体的
な情報が見つかりませんでした。個別の医療アドバイスについては医療提供者にご相談ください。
",
                "summary_template": "ガイドラインによれば、{summary}",
                "error": "申し訳ございません。応答の生成中にエラーが発生しました。し
ばらくしてからもう一度お試しください。"
            }
        }

        # Medical term patterns for extraction
        self.medical_patterns = {
            "recommendation_keywords": {
                "en": ["recommend", "suggest", "should", "advise",
"indicated", "preferable", "appropriate"],
                "ja": ["推奨", "勧告", "すべき", "提案", "適応", "望ましい"]
            },
            "treatment_keywords": {
                "en": ["treatment", "therapy", "medication", "dose",
"management", "intervention"],
                "ja": ["治療", "療法", "薬物", "投与量", "管理", "介入"]
            },
            "diagnosis_keywords": {
```

```
                "en": ["diagnosis", "symptoms", "signs", "test",
"evaluation", "assessment"],
                "ja": ["診断", "症状", "徴候", "検査", "評価", "判定"]
            }
        }

        # Medical topics with synonyms
        self.medical_topics = {
            "en": {
                "diabetes": ["diabetes", "diabetic", "blood sugar",
"glucose", "insulin", "glycemic"],
                "hypertension": ["hypertension", "high blood pressure",
"blood pressure", "bp"],
                "heart disease": ["heart disease", "cardiac",
"cardiovascular", "coronary", "cvd"],
                "cancer": ["cancer", "tumor", "oncology", "malignant",
"carcinoma"],
                "asthma": ["asthma", "bronchial", "respiratory"],
            },
            "ja": {
                "糖尿病": ["糖尿病", "血糖", "インスリン", "血糖値"],
                "高血圧": ["高血圧", "血圧"],
                "心臓病": ["心臓病", "心疾患", "心血管", "冠動脈"],
                "がん": ["がん", "癌", "腫瘍", "悪性"],
                "喘息": ["喘息", "気管支"],
            }
        }

        self.response_count = 0
        logger.info("LLMService initialized with template-based response
generation")

    def generate_response(
        self,
        query: str,
        retrieved_docs: List[Dict],
        language: str = "en"
    ) -> Dict:
        """
        Generate structured medical response from retrieved documents.
```

```
        Args:
            query: User's query string
            retrieved_docs: List of retrieved document chunks with metadata
            language: Output language ('en' or 'ja')

        Returns:
            Dictionary containing:
                - response: Generated response text
                - sources: List of source documents used
                - language: Output language
                - generation_time_seconds: Time taken to generate
                - documents_used: Number of documents used

        Raises:
            ValueError: If query is empty

        Example:
            >>> response = service.generate_response(
            ...     "What are diabetes guidelines?",
            ...     documents,
            ...     "en"
            ... )
        """
        start_time = time.time()

        try:
            # Validate inputs
            if not query or not query.strip():
                raise ValueError("Query cannot be empty")

            if retrieved_docs is None:
                retrieved_docs = []

            # Validate and normalize language
            if language not in ["en", "ja"]:
                logger.warning(
                    f"Unsupported language '{language}', defaulting to
'en'",
                    extra={"requested_language": language}
```

```
                )
                language = "en"

            if not retrieved_docs:
                return self._generate_no_results_response(query, language)

            # Extract main topic from query
            topic = self._extract_topic(query, language)

            # Build structured response
            response_parts = []

            # 1. Introduction
            introduction = self._generate_introduction(topic, language)
            response_parts.append(introduction)

            # 2. Key findings from documents
            findings = self._extract_key_findings(retrieved_docs, language)
            if findings:
                findings_header =
self.response_templates[language]["findings_header"]
                response_parts.append(f"\n{findings_header}\n{findings}")

            # 3. Recommendations
            recommendations = self._extract_recommendations(retrieved_docs,
language)
            if recommendations:
                rec_header =
self.response_templates[language]["recommendations_header"]
                response_parts.append(f"\n{rec_header}\n{recommendations}")

            # 4. Summary if no specific findings/recommendations
            if not findings and not recommendations:
                summary = self._generate_summary(retrieved_docs, language)
                if summary:
                    response_parts.append(f"\n{summary}")

            # 5. Medical disclaimer
            disclaimer = self.response_templates[language]["disclaimer"]
            response_parts.append(disclaimer)
```

```
        # Combine all parts
        response_text = "\n".join(response_parts)

        # Prepare sources information
        sources = self._prepare_sources(retrieved_docs)

        self.response_count += 1
        generation_time = time.time() - start_time

        logger.info(
            "Response generated successfully",
            extra={
                "query_preview": query[:50],
                "language": language,
                "documents_used": len(retrieved_docs),
                "generation_time": round(generation_time, 3),
                "response_length": len(response_text)
            }
        )

        return {
            "response": response_text,
            "sources": sources,
            "language": language,
            "generation_time_seconds": round(generation_time, 3),
            "documents_used": len(retrieved_docs)
        }

    except ValueError as e:
        logger.warning(f"Validation error: {str(e)}")
        raise
    except Exception as e:
        logger.error(
            "Response generation failed",
            extra={
                "query_preview": query[:50] if query else None,
                "error": str(e)
            },
            exc_info=True
```

```
            )
            return self._generate_error_response(query, language)

    def _extract_topic(self, query: str, language: str) -> str:
        """
        Extract main medical topic from query.

        Args:
            query: User query
            language: Query language

        Returns:
            Extracted topic string
        """
        query_lower = query.lower()
        topics_dict = self.medical_topics.get(language,
self.medical_topics["en"])

        # Check for topic matches with synonyms
        for topic, synonyms in topics_dict.items():
            if any(syn.lower() in query_lower for syn in synonyms):
                return topic

        # Fallback: return first few words or entire short query
        words = query.split()
        if len(words) <= 5:
            return query
        else:
            return " ".join(words[:3]) + "..."

    def _generate_introduction(self, topic: str, language: str) -> str:
        """Generate introduction section of response."""
        template = self.response_templates[language]["introduction"]
        return template.format(topic=topic)

    def _extract_key_findings(self, documents: List[Dict], language: str)
-> str:
        """
        Extract key findings from retrieved documents.
```

```
        Args:
            documents: List of document chunks
            language: Output language

        Returns:
            Formatted findings text
        """
        findings = []

        for i, doc in enumerate(documents[:3], 1):
            text = doc.get('text', '')
            doc_lang = doc.get('language', 'en')

            if not text:
                continue

            # Translate document text if language mismatch
            if doc_lang != language:
                try:
                    logger.info(f"Translating document text from {doc_lang}
to {language}")
                    # Translate first 400 chars
                    text_to_translate = text[:400]
                    translated_text =
translation_service.translate(text_to_translate, doc_lang, language,
max_length=300)

                    # Use translation even if quality is uncertain
                    if translated_text and len(translated_text) > 5:
                        text = translated_text
                        logger.info(f"Translation completed: {len(text)}
chars")
                    else:
                        logger.warning(f"Translation returned empty/short
result, using original")

                except Exception as e:
                    logger.warning(f"Translation failed: {e}, using
original text")
                    # Use original if translation completely fails
```

```
            # Extract first 1-2 meaningful sentences

            sentences = self._split_into_sentences(text, language)
            meaningful_sentences = []

            for sentence in sentences[:2]:
                if len(sentence.strip()) > 10:
                    meaningful_sentences.append(sentence.strip())

            if meaningful_sentences:
                summary = ('。' if language == 'ja' else '.
').join(meaningful_sentences)
                if not summary.endswith('.') and not summary.endswith('。
'):
                    summary += '。' if language == 'ja' else '.'
                findings.append(f"{i}. {summary}")

        return "\n".join(findings) if findings else ""

    def _extract_recommendations(self, documents: List[Dict], language:
str) -> str:
        """
        Extract recommendation sentences from documents.

        Args:
            documents: List of document chunks
            language: Output language

        Returns:
            Formatted recommendations text
        """
        recommendations = []
        seen_sentences: Set[str] = set()
        keywords =
self.medical_patterns["recommendation_keywords"][language]

        for doc in documents[:5]:
            text = doc.get('text', '')
            doc_lang = doc.get('language', 'en')
```

```
            if not text:
                continue

            # Translate document text if language mismatch
            if doc_lang != language:
                try:
                    logger.info(f"Translating recommendation from
{doc_lang} to {language}")
                    # Translate first 400 chars
                    text_to_translate = text[:400]
                    translated_text =
translation_service.translate(text_to_translate, doc_lang, language,
max_length=300)

                    # Use translation
                    if translated_text and len(translated_text) > 5:
                        text = translated_text
                        logger.info(f"Translation completed: {len(text)}
chars")
                    else:
                        logger.warning(f"Translation returned empty/short,
using original")

                except Exception as e:
                    logger.warning(f"Translation failed: {e}, using
original")

            # Use target language for sentence splitting
            sentences = self._split_into_sentences(text, language)

            for sentence in sentences[:3]:
                sentence_normalized = ' '.join(sentence.split())

                # Skip if already seen (avoid duplicates)
                if sentence_normalized.lower() in seen_sentences:
                    continue

                # Include sentence if it has keywords OR is substantial
(>20 chars)
```

```
                has_keyword = any(keyword.lower() in sentence.lower() for
keyword in keywords)
                is_substantial = len(sentence.strip()) > 20

                if has_keyword or is_substantial:
                    clean_sentence = sentence.strip()
                    recommendations.append(f"• {clean_sentence}")
                    seen_sentences.add(sentence_normalized.lower())
                    if len(recommendations) >= 5:  # Limit to 5
recommendations
                        break

            if len(recommendations) >= 5:
                break

        return "\n".join(recommendations[:5])

    def _generate_summary(self, documents: List[Dict], language: str) ->
str:
        """
        Generate summary when no specific findings/recommendations are
found.

        Args:
            documents: List of document chunks
            language: Output language

        Returns:
            Summary text
        """
        # Extract and translate summary from documents
        translated_texts = []

        for doc in documents[:3]:
            text = doc.get('text', '')
            doc_lang = doc.get('language', 'en')

            if not text:
                continue
```

```
            # Translate if language mismatch
            if doc_lang != language:
                try:
                    logger.info(f"Translating summary text from {doc_lang}
to {language}")
                    text_to_translate = text[:300]
                    translated =
translation_service.translate(text_to_translate, doc_lang, language,
max_length=200)

                    if translated and len(translated) > 5:
                        text = translated
                        logger.info(f"Summary translation successful")
                    else:
                        logger.warning(f"Translation short, using
original")

                except Exception as e:
                    logger.warning(f"Summary translation failed: {e}, using
original")

            translated_texts.append(text)

        if not translated_texts:
            # Fallback if no text
            if language == "ja":
                fallback_text = "個別の症例については医療専門家にご相談ください"
            else:
                fallback_text = "consultation with healthcare professionals
is recommended for specific cases"
            template =
self.response_templates[language]["summary_template"]
            return template.format(summary=fallback_text)

        # Combine texts and split into sentences
        all_text = " ".join(translated_texts)
        sentences = self._split_into_sentences(all_text, language)

        # Use first meaningful sentence as summary
        for sentence in sentences:
```

```
            if self._is_meaningful_sentence(sentence, language):
                template =
self.response_templates[language]["summary_template"]
                return template.format(summary=sentence.strip())

        # Fallback
        if language == "ja":
            fallback_text = "個別の症例については医療専門家にご相談ください"
        else:
            fallback_text = "consultation with healthcare professionals is
recommended for specific cases"

        template = self.response_templates[language]["summary_template"]
        return template.format(summary=fallback_text)

    def _prepare_sources(self, documents: List[Dict]) -> List[Dict]:
        """Prepare source information for response."""
        sources = []
        for doc in documents:
            sources.append({
                "document_id": doc.get('doc_id', 'unknown'),
                "chunk_id": doc.get('chunk_id', 0),
                "similarity_score": round(doc.get('similarity_score', 0.0),
3),
                "language": doc.get('language', 'en')
            })
        return sources

    def _split_into_sentences(self, text: str, language: str) -> List[str]:
        """
        Split text into sentences using NLTK for English, regex for
Japanese.

        Args:
            text: Input text
            language: Text language

        Returns:
            List of sentences
        """
```

```
        try:
            if language == "ja":
                # For Japanese, use regex (NLTK doesn't handle Japanese
well)
                sentences = re.split(r'(?<=[。！？])\s*', text)
            else:
                # Use NLTK for English (better handling of abbreviations)
                from nltk.tokenize import sent_tokenize
                sentences = sent_tokenize(text)

            # Clean and filter sentences
            clean_sentences = []
            for sentence in sentences:
                sentence = sentence.strip()
                if sentence and len(sentence) > 10:
                    clean_sentences.append(sentence)

            return clean_sentences

        except Exception as e:
            logger.debug(f"Sentence tokenization failed, using fallback:
{e}")
            # Fallback to simple regex
            if language == "ja":
                sentences = re.split(r'[。！？]', text)
            else:
                sentences = re.split(r'[.!?]', text)
            return [s.strip() for s in sentences if s.strip() and
len(s.strip()) > 10]

    def _is_meaningful_sentence(self, sentence: str, language: str) ->
bool:
        """
        Check if sentence contains meaningful medical content.

        Args:
            sentence: Sentence to check
            language: Sentence language

        Returns:
```

```
            True if sentence is meaningful
        """
        if len(sentence) < 20:
            return False

        # Check for medical keywords
        all_keywords = []
        for pattern in self.medical_patterns.values():
            all_keywords.extend(pattern.get(language, []))

        sentence_lower = sentence.lower()
        has_keyword = any(keyword in sentence_lower for keyword in
all_keywords)

        # Also accept sentences with numbers
        has_number = bool(re.search(r'\d', sentence))

        return has_keyword or has_number

    def _generate_no_results_response(self, query: str, language: str) ->
Dict:
        """Generate response when no documents are retrieved."""
        topic = self._extract_topic(query, language)
        template = self.response_templates[language]["no_results"]
        response_text = template.format(topic=topic)

        logger.info(
            "Generated no-results response",
            extra={"query_preview": query[:50], "language": language}
        )

        return {
            "response": response_text,
            "sources": [],
            "language": language,
            "generation_time_seconds": 0.0,
            "documents_used": 0
        }

    def _generate_error_response(self, query: str, language: str) -> Dict:
```

```
        """Generate error response when generation fails."""
        response_text = self.response_templates[language]["error"]

        return {
            "response": response_text,
            "sources": [],
            "language": language,
            "generation_time_seconds": 0.0,
            "documents_used": 0
        }

    def get_service_info(self) -> Dict:
        """Get information about the LLM service."""
        return {
            "service_type": "template_based",
            "response_count": self.response_count,
            "supported_languages": ["en", "ja"],
            "templates_available": list(self.response_templates.keys()),
            "medical_topics_count": sum(len(topics) for topics in
self.medical_topics.values())
        }

    def health_check(self) -> Dict:
        """Perform health check on LLM service."""
        try:
            # Test response generation
            test_query = "diabetes management"
            test_docs = [{
                "text": "Diabetes management involves regular monitoring of
blood glucose levels and adherence to prescribed medications.",
                "doc_id": "test_doc",
                "chunk_id": 0,
                "similarity_score": 0.9,
                "language": "en"
            }]

            response = self.generate_response(test_query, test_docs, "en")

            return {
                "status": "healthy",
```

```
                "response_generation_works": True,
                "response_count": self.response_count,
                "service_type": "template_based",
                "test_response_length": len(response["response"])
            }

        except Exception as e:
            logger.error(f"Health check failed: {e}", exc_info=True)
            return {
                "status": "unhealthy",
                "error": str(e),
                "response_count": self.response_count
            }


# Global instance for dependency injection
llm_service = LLMService()


# For testing purposes
if __name__ == "__main__":
    print("Testing LLM Service...\n")

    service = LLMService()

    # Test data
    test_documents = [
        {
            "text": "Type 2 diabetes management requires comprehensive
lifestyle modifications including dietary changes and physical activity.
Regular monitoring of HbA1c levels is essential for glycemic control.",
            "doc_id": "doc_1",
            "chunk_id": 0,
            "similarity_score": 0.95,
            "language": "en"
        },
        {
            "text": "First-line pharmacological therapy for type 2 diabetes
typically includes metformin. Patients should be advised on proper
medication adherence and potential side effects.",
```

```
            "doc_id": "doc_1",
            "chunk_id": 1,
            "similarity_score": 0.88,
            "language": "en"
        }
    ]

    # Test English response
    print("=" * 60)
    print("TEST 1: English Response")
    print("=" * 60)
    response_en = service.generate_response(
        "What are the guidelines for diabetes management?",
        test_documents,
        "en"
    )
    print(response_en["response"])
    print(f"\nSources: {response_en['sources']}")
    print(f"Generation time:
{response_en['generation_time_seconds']:.3f}s\n")

    # Test Japanese response
    print("=" * 60)
    print("TEST 2: Japanese Response")
    print("=" * 60)
    response_ja = service.generate_response(
        "糖尿病管理のガイドラインは何ですか？",
        test_documents,
        "ja"
    )
    print(response_ja["response"])
    print(f"\nSources: {response_ja['sources']}")
    print(f"Generation time:
{response_ja['generation_time_seconds']:.3f}s\n")

    # Test no results
    print("=" * 60)
    print("TEST 3: No Results Response")
    print("=" * 60)
    response_no_results = service.generate_response(
```

```
        "What about quantum physics?",
        [],
        "en"
    )
    print(response_no_results["response"])
    print()

    # Test service info
    print("=" * 60)
    print("TEST 4: Service Info")
    print("=" * 60)
    info = service.get_service_info()
    print(f"Service info: {info}\n")

    # Test health check
    print("=" * 60)
    print("TEST 5: Health Check")
    print("=" * 60)
    health = service.health_check()
    print(f"Health check: {health}\n")

    print("All tests completed!")
```

This is the AI response generator for the Healthcare Search Assistant. It creates structured medical answers based on retrieved documents.

What it does:

- Takes medical questions and found documents
- Generates well-organized, professional medical responses
- Supports both English and Japanese output
- Creates responses with proper medical formatting

How it works:

Response Structure:

- Introduction - "Based on the retrieved medical guidelines regarding [topic]:"

- Key Findings - Main medical facts from documents
- Recommendations - Treatment suggestions and advice
- Disclaimer - Medical disclaimer for safety

Key Features:

Template-Based System:

- Uses predefined templates for consistent formatting
- Different templates for English and Japanese
- Handles medical terminology appropriately

Medical Intelligence:

- Extracts key medical topics (diabetes, hypertension, cancer, etc.)
- Identifies recommendations and findings
- Uses medical keywords to find important content
- Filters out irrelevant or duplicate information

Bilingual Support:

- Full support for English and Japanese
- Automatic translation of document content when needed
- Language-specific sentence splitting
- Culturally appropriate medical phrasing

Content Processing:

- Sentence Extraction: Gets 1-2 most relevant sentences per document
- Duplicate Removal: Avoids repeating the same information
- Quality Filtering: Only uses meaningful medical content

Source Tracking: Cites which documents were used

Error Handling:

No Documents: "I couldn't find specific information about [topic]..."

Generation Errors: "I apologize, but an error occurred..."

Translation Failures: Falls back to original text

Quality Controls:

Minimum sentence length requirements

Medical keyword detection

Duplicate sentence filtering

Meaningful content validation

Monitoring:

Tracks response counts and performance

Health checks verify service is working

Detailed logging of generation process

app/services/translation_service.py:

```
# app/services/translation_service.py
"""
Translation service for English-Japanese bidirectional translation.

Provides reliable translation using Helsinki-NLP MarianMT models with
lazy loading, batch processing, and GPU support.
"""

import logging
import time
from typing import List, Optional, Dict, Any
from transformers import MarianMTModel, MarianTokenizer
```

```python
import torch
from app.config import settings
from app.utils.language_detector import detect_language

logger = logging.getLogger(__name__)


class TranslationService:
    """
    Service for handling translation between English and Japanese.

    Features:
        - Lazy loading of translation models
        - Bidirectional EN↔JA translation
        - Batch processing for efficiency
        - GPU support with automatic detection
        - Error handling and fallbacks

    Example:
        >>> service = TranslationService()
        >>> result = service.translate("Hello", "en", "ja")
        >>> print(result)  # こんにちは
    """

    def __init__(self, device: Optional[str] = None):
        """
        Initialize translation service with lazy loading.

        Args:
            device: Device to use ('cuda', 'cpu', or None for auto-detect)
        """
        # Model names for Helsinki-NLP MarianMT models
        self.en_ja_model_name = "Helsinki-NLP/opus-mt-en-jap"
        self.ja_en_model_name = "Helsinki-NLP/opus-mt-jap-en"

        # Device configuration
        if device is None:
            self.device = "cuda" if torch.cuda.is_available() else "cpu"
        else:
            self.device = device
```

```
        # Lazy loading - models loaded on first use
        self.en_ja_model: Optional[MarianMTModel] = None
        self.en_ja_tokenizer: Optional[MarianTokenizer] = None
        self.ja_en_model: Optional[MarianMTModel] = None
        self.ja_en_tokenizer: Optional[MarianTokenizer] = None

        # Model loading flags
        self.en_ja_loaded = False
        self.ja_en_loaded = False

        # Translation statistics
        self.translation_count = 0
        self.error_count = 0

        logger.info(
            "TranslationService initialized",
            extra={
                "device": self.device,
                "en_ja_model": self.en_ja_model_name,
                "ja_en_model": self.ja_en_model_name
            }
        )

    def _load_en_ja_model(self):
        """Load English to Japanese translation model and tokenizer."""
        if self.en_ja_loaded:
            return

        try:
            logger.info(f"Loading English→Japanese model:
{self.en_ja_model_name}")

            self.en_ja_tokenizer = MarianTokenizer.from_pretrained(
                self.en_ja_model_name
            )
            self.en_ja_model = MarianMTModel.from_pretrained(
                self.en_ja_model_name
            )
```

```
            # Set model to evaluation mode and move to device
            self.en_ja_model.eval()
            self.en_ja_model = self.en_ja_model.to(self.device)

            self.en_ja_loaded = True
            logger.info(f"English→Japanese model loaded on {self.device}")

        except Exception as e:
            logger.error(
                "Failed to load English→Japanese model",
                extra={"error": str(e)},
                exc_info=True
            )
            raise RuntimeError(f"Could not load translation model:
{str(e)}")

    def _load_ja_en_model(self):
        """Load Japanese to English translation model and tokenizer."""
        if self.ja_en_loaded:
            return

        try:
            logger.info(f"Loading Japanese→English model:
{self.ja_en_model_name}")

            self.ja_en_tokenizer = MarianTokenizer.from_pretrained(
                self.ja_en_model_name
            )
            self.ja_en_model = MarianMTModel.from_pretrained(
                self.ja_en_model_name
            )

            # Set model to evaluation mode and move to device
            self.ja_en_model.eval()
            self.ja_en_model = self.ja_en_model.to(self.device)

            self.ja_en_loaded = True
            logger.info(f"Japanese→English model loaded on {self.device}")

        except Exception as e:
```

```
            logger.error(
                "Failed to load Japanese→English model",
                extra={"error": str(e)},
                exc_info=True
            )
            raise RuntimeError(f"Could not load translation model:
{str(e)}")

    def translate(
        self,
        text: str,
        source_lang: str,
        target_lang: str,
        max_length: int = 512
    ) -> str:
        """
        Translate text between English and Japanese.

        Args:
            text: Text to translate
            source_lang: Source language ('en' or 'ja')
            target_lang: Target language ('en' or 'ja')
            max_length: Maximum length for translation

        Returns:
            Translated text

        Raises:
            ValueError: For unsupported language pairs or invalid input
            RuntimeError: If translation fails

        Example:
            >>> result = service.translate("Hello", "en", "ja")
            >>> print(result)  # こんにちは
        """
        # Validate inputs
        if not text or not text.strip():
            return text if text is not None else ""

        if source_lang == target_lang:
```

```
            return text.strip()

        # Validate language codes
        if source_lang not in ['en', 'ja'] or target_lang not in ['en',
'ja']:
            raise ValueError(
                f"Unsupported language pair: {source_lang} → {target_lang}.
"
                "Only 'en' and 'ja' are supported."
            )

        # Check text length (rough estimate: 1 token ≈ 4 chars)
        max_chars = max_length * 4
        if len(text) > max_chars:
            logger.warning(
                "Text too long, truncating",
                extra={
                    "text_length": len(text),
                    "max_chars": max_chars,
                    "truncated": True
                }
            )
            text = text[:max_chars]

        start_time = time.time()

        try:
            if source_lang == 'en' and target_lang == 'ja':
                self._load_en_ja_model()
                result = self._translate_with_model(
                    text, self.en_ja_model, self.en_ja_tokenizer,
max_length
                )

            elif source_lang == 'ja' and target_lang == 'en':
                self._load_ja_en_model()
                result = self._translate_with_model(
                    text, self.ja_en_model, self.ja_en_tokenizer,
max_length
                )
```

```
            else:
                raise ValueError(f"Unsupported translation direction:
{source_lang} → {target_lang}")

            self.translation_count += 1
            translation_time = time.time() - start_time

            logger.debug(
                "Translation completed",
                extra={
                    "source_lang": source_lang,
                    "target_lang": target_lang,
                    "text_length": len(text),
                    "translation_time": round(translation_time, 3),
                    "device": self.device
                }
            )

            return result

        except Exception as e:
            self.error_count += 1
            logger.error(
                "Translation failed",
                extra={
                    "source_lang": source_lang,
                    "target_lang": target_lang,
                    "text_length": len(text),
                    "error": str(e)
                },
                exc_info=True
            )
            raise RuntimeError(f"Translation failed: {str(e)}")

    def _translate_with_model(
        self,
        text: str,
        model: MarianMTModel,
        tokenizer: MarianTokenizer,
```

```
        max_length: int
    ) -> str:
        """
        Perform translation using a loaded model and tokenizer.

        Args:
            text: Text to translate
            model: Loaded MarianMT model
            tokenizer: Loaded tokenizer
            max_length: Maximum sequence length

        Returns:
            Translated text
        """
        try:
            # Tokenize input text
            inputs = tokenizer(
                text,
                return_tensors="pt",
                padding=True,
                truncation=True,
                max_length=max_length
            )

            # Move to device
            inputs = {k: v.to(self.device) for k, v in inputs.items()}

            # Generate translation
            with torch.no_grad():
                translated_tokens = model.generate(
                    **inputs,
                    max_length=max_length,
                    num_beams=4,
                    early_stopping=True
                )

            # Decode translation
            result = tokenizer.decode(
                translated_tokens[0],
                skip_special_tokens=True
```

```
            )

            return result.strip()

        except Exception as e:
            logger.error(f"Model translation failed: {str(e)}",
exc_info=True)
            raise

    def batch_translate(
        self,
        texts: List[str],
        source_lang: str,
        target_lang: str,
        max_length: int = 512,
        batch_size: int = 8
    ) -> List[str]:
        """
        Translate multiple texts in batches for efficiency.

        Args:
            texts: List of texts to translate
            source_lang: Source language ('en' or 'ja')
            target_lang: Target language ('en' or 'ja')
            max_length: Maximum sequence length per text
            batch_size: Number of texts to process in batch

        Returns:
            List of translated texts in same order as input

        Example:
            >>> texts = ["Hello", "How are you?", "Good morning"]
            >>> results = service.batch_translate(texts, "en", "ja")
        """
        if not texts:
            return []

        # Handle same language
        if source_lang == target_lang:
            return [text.strip() if text else "" for text in texts]
```

```
        # Validate language pair
        if source_lang not in ['en', 'ja'] or target_lang not in ['en',
'ja']:
            raise ValueError(f"Unsupported language pair: {source_lang} →
{target_lang}")

        # Filter and track empty texts
        texts_to_translate = []
        empty_indices = set()
        for i, text in enumerate(texts):
            if text and text.strip():
                texts_to_translate.append(text.strip())
            else:
                empty_indices.add(i)

        if not texts_to_translate:
            return [""] * len(texts)

        # Load appropriate model
        if source_lang == 'en' and target_lang == 'ja':
            self._load_en_ja_model()
            model = self.en_ja_model
            tokenizer = self.en_ja_tokenizer
        elif source_lang == 'ja' and target_lang == 'en':
            self._load_ja_en_model()
            model = self.ja_en_model
            tokenizer = self.ja_en_tokenizer
        else:
            raise ValueError(f"Unsupported language pair: {source_lang} →
{target_lang}")

        results = []

        try:
            # Process in batches
            for i in range(0, len(texts_to_translate), batch_size):
                batch = texts_to_translate[i:i + batch_size]

                # Tokenize entire batch at once
```

```
                inputs = tokenizer(
                    batch,
                    return_tensors="pt",
                    padding=True,
                    truncation=True,
                    max_length=max_length
                )

                # Move to device
                inputs = {k: v.to(self.device) for k, v in inputs.items()}

                # Generate translations
                with torch.no_grad():
                    translated_tokens = model.generate(
                        **inputs,
                        max_length=max_length,
                        num_beams=4,
                        early_stopping=True
                    )

                # Decode each translation
                batch_results = [
                    tokenizer.decode(tokens,
skip_special_tokens=True).strip()
                    for tokens in translated_tokens
                ]

                results.extend(batch_results)
                self.translation_count += len(batch_results)

            logger.debug(
                "Batch translation completed",
                extra={
                    "source_lang": source_lang,
                    "target_lang": target_lang,
                    "text_count": len(texts_to_translate),
                    "batch_size": batch_size
                }
            )
```

```
            # Reconstruct full results with empty strings in original
positions
            final_results = []
            result_idx = 0
            for i in range(len(texts)):
                if i in empty_indices:
                    final_results.append("")
                else:
                    final_results.append(results[result_idx])
                    result_idx += 1

            return final_results

        except Exception as e:
            self.error_count += len(texts_to_translate)
            logger.error(
                "Batch translation failed, falling back to individual",
                extra={"error": str(e)},
                exc_info=True
            )
            # Fallback: translate individually
            fallback_results = []
            for text in texts:
                try:
                    if text and text.strip():
                        fallback_results.append(
                            self.translate(text, source_lang, target_lang,
max_length)
                        )
                    else:
                        fallback_results.append("")
                except Exception:
                    fallback_results.append(text if text else "")
            return fallback_results

    def detect_and_translate(self, text: str, target_lang: str) -> str:
        """
        Detect language and translate to target language if needed.

        Args:
```

```
            text: Text to potentially translate
            target_lang: Target language ('en' or 'ja')

        Returns:
            Translated text if language differs, otherwise original text

        Example:
            >>> result = service.detect_and_translate("Hello", "ja")
            >>> print(result)  # こんにちは
        """
        if not text or not text.strip():
            return text if text is not None else ""

        source_lang = detect_language(text)

        if source_lang == target_lang:
            return text

        try:
            return self.translate(text, source_lang, target_lang)
        except Exception as e:
            logger.warning(
                f"Auto-translation failed, returning original text: {e}",
                extra={"detected_lang": source_lang, "target_lang":
target_lang}
            )
            return text

    def warmup(self):
        """
        Warm up translation models with dummy translations.

        Ensures models are fully loaded and compiled for better
        first-call performance.
        """
        logger.info("Warming up translation models...")

        try:
            # Warm up EN→JA
            self._load_en_ja_model()
```

```
            _ = self._translate_with_model(
                "test",
                self.en_ja_model,
                self.en_ja_tokenizer,
                50
            )

            # Warm up JA→EN
            self._load_ja_en_model()
            _ = self._translate_with_model(
                "テスト",
                self.ja_en_model,
                self.ja_en_tokenizer,
                50
            )

            logger.info("Translation models warmed up successfully")

        except Exception as e:
            logger.warning(f"Translation warmup failed: {e}")

    def unload_models(self):
        """
        Unload models from memory to free resources.

        Useful for testing or when switching configurations.
        """
        if self.en_ja_model is not None:
            del self.en_ja_model
            del self.en_ja_tokenizer
            self.en_ja_model = None
            self.en_ja_tokenizer = None
            self.en_ja_loaded = False

        if self.ja_en_model is not None:
            del self.ja_en_model
            del self.ja_en_tokenizer
            self.ja_en_model = None
            self.ja_en_tokenizer = None
            self.ja_en_loaded = False
```

```
        # Clear CUDA cache
        if torch.cuda.is_available():
            torch.cuda.empty_cache()

        logger.info("Translation models unloaded from memory")

    def get_model_info(self) -> Dict[str, Any]:
        """Get information about loaded translation models."""
        return {
            "en_ja_loaded": self.en_ja_loaded,
            "ja_en_loaded": self.ja_en_loaded,
            "en_ja_model": self.en_ja_model_name,
            "ja_en_model": self.ja_en_model_name,
            "translation_count": self.translation_count,
            "error_count": self.error_count,
            "device": self.device,
            "cuda_available": torch.cuda.is_available()
        }

    def health_check(self) -> Dict[str, Any]:
        """Perform health check on translation service."""
        try:
            # Test translation in both directions
            test_text_en = "Hello, how are you?"
            test_text_ja = "こんにちは、元気ですか？"

            # Test EN→JA
            self._load_en_ja_model()
            result_ja = self._translate_with_model(
                test_text_en, self.en_ja_model, self.en_ja_tokenizer, 100
            )

            # Test JA→EN
            self._load_ja_en_model()
            result_en = self._translate_with_model(
                test_text_ja, self.ja_en_model, self.ja_en_tokenizer, 100
            )

            return {
```

```
                "status": "healthy",
                "en_ja_loaded": self.en_ja_loaded,
                "ja_en_loaded": self.ja_en_loaded,
                "test_translation_works": True,
                "device": self.device,
                "translation_count": self.translation_count,
                "error_count": self.error_count
            }

        except Exception as e:
            return {
                "status": "unhealthy",
                "en_ja_loaded": self.en_ja_loaded,
                "ja_en_loaded": self.ja_en_loaded,
                "error": str(e)
            }

    def preload_models(self):
        """
        Preload both translation models (useful for warm startup).

        Alias for warmup() for backward compatibility.
        """
        logger.info("Preloading translation models...")
        self._load_en_ja_model()
        self._load_ja_en_model()
        logger.info("Translation models preloaded successfully")

        # Also do warmup
        self.warmup()


# Global instance for dependency injection
translation_service = TranslationService()


# For testing purposes
if __name__ == "__main__":
    print("Testing Translation Service...\n")
```

```
service = TranslationService()

print("=" * 60)
print("TEST 1: English to Japanese")
print("=" * 60)
result_ja = service.translate("Hello, how are you today?", "en", "ja")
print(f"EN→JA: {result_ja}\n")

print("=" * 60)
print("TEST 2: Japanese to English")
print("=" * 60)
result_en = service.translate("こんにちは、今日は元気ですか？", "ja", "en")
print(f"JA→EN: {result_en}\n")

print("=" * 60)
print("TEST 3: Batch Translation")
print("=" * 60)
texts_en = ["Hello", "How are you?", "Good morning"]
results_ja = service.batch_translate(texts_en, "en", "ja")
print(f"Batch EN→JA:")
for orig, trans in zip(texts_en, results_ja):
    print(f"  {orig} → {trans}")
print()

print("=" * 60)
print("TEST 4: Auto-detect and Translate")
print("=" * 60)
result_auto = service.detect_and_translate("Hello", "ja")
print(f"Auto (Hello → JA): {result_auto}\n")

print("=" * 60)
print("TEST 5: Model Info")
print("=" * 60)
info = service.get_model_info()
print(f"Model info: {info}\n")

print("=" * 60)
print("TEST 6: Health Check")
print("=" * 60)
health = service.health_check()
```

```
    print(f"Health check: {health}\n")

    print("All tests completed!")
```

This is the translation service for the Healthcare Search Assistant. It handles converting medical text between English and Japanese.

What it does:

- Translates medical text between English and Japanese
- Uses AI translation models for accurate medical terminology
- Supports both single texts and batch processing
- Automatically detects when translation is needed

Key Features:

Translation Models:

- English → Japanese: Uses Helsinki-NLP/opus-mt-en-jap model
- Japanese → English: Uses Helsinki-NLP/opus-mt-jap-en model
- GPU Support: Automatically uses GPU if available for faster processing
- Lazy Loading: Only loads models when needed to save memory

Core Functions:

- translate(text, source_lang, target_lang) - Single translation
  - Translates one piece of text
  - Handles text length limits automatically
  - Returns cleaned, trimmed translation
- batch_translate(texts, source_lang, target_lang) - Multiple translations
  - Processes multiple texts efficiently in batches
  - Preserves original order of texts

Handles empty texts gracefully

detect_and_translate(text, target_lang) - Smart translation

Automatically detects text language

Only translates if needed

Falls back to original text if translation fails

Technical Details:

Performance Optimizations:

Batch Processing: Translates multiple texts together for efficiency

Beam Search: Uses 4 beams for better translation quality

Length Management: Automatically truncates very long texts

Memory Management: Can unload models to free memory

Error Handling:

Same Language: Returns original text if no translation needed

Empty Text: Handles empty inputs gracefully

Translation Failures: Falls back to original text with warnings

Model Loading: Comprehensive error reporting for model issues

Monitoring & Maintenance:

Health Checks: Verifies both translation directions work

Model Info: Shows loaded models and usage statistics

Performance Tracking: Counts translations and errors

Warmup: Pre-loads models for faster first response

Use Cases in the System:

Translating document content for consistent language responses

Converting user queries for better search results

Ensuring generated responses are in the requested language

Handling mixed-language document collections

This concludes all the files and their explanation in the project.

Things to be added in the future for further improvements:

1. Unit and Integration Tests
2. PDF/DOCX support
3. Real LLM Integration (OpenAI, Anthropic)
4. Prometheus Metrics
5. User management and RBAC
6. Automatic Backups
7. Multi Language Support (beyond En/ Jap)
8. Chat History and context
9. Fine Tuned Models
10. Caching(via Redis, celery)
11. Admin dashboard
12. Database Integration (MongoDB or Postgresql)
13. Rate Limiting Tiers
14. Document Versioning