"""

Integration tests for the complete RAG pipeline.

This module tests the end-to-end functionality of the RAG system,

from document loading to question answering and evaluation.

"""

import pytest

from pathlib import Path

import tempfile

import shutil

from unittest.mock import patch, Mock

import json

from config.settings import Settings

from src.data\_loader.pdf\_loader import PDFDocumentLoader

from src.text\_processing.chunker import DocumentChunker

from src.embeddings.openai\_embeddings import OpenAIEmbeddingModel

from src.vector\_store.chroma\_store import ChromaVectorStore

from src.llm.openai\_llm import OpenAILLM

from src.retrieval.qa\_chain import QAChain

from src.evaluation.ragas\_evaluator import RagasEvaluator

from main import RAGPipeline

from langchain.schema import Document

@pytest.mark.integration

class TestRAGPipelineIntegration:

"""Integration tests for the RAG pipeline."""

@pytest.fixture

def temp\_dir(self):

"""Create a temporary directory for test data."""

temp\_dir = tempfile.mkdtemp()

yield Path(temp\_dir)

shutil.rmtree(temp\_dir, ignore\_errors=True)

@pytest.fixture

def mock\_settings(self, temp\_dir):

"""Create mock settings for testing."""

settings = Settings(

openai\_api\_key="test-key",

vector\_store\_persist\_directory=str(temp\_dir / "vector\_store"),

data\_directory=temp\_dir,

documents\_directory=temp\_dir / "documents",

evaluation\_directory=temp\_dir / "evaluation",

log\_file=str(temp\_dir / "test.log")

)

return settings

@pytest.fixture

def sample\_documents(self):

"""Create sample documents for testing."""

return [

Document(

page\_content="The iPhone is a smartphone made by Apple. It runs iOS operating system.",

metadata={"source": "iphone\_guide.pdf", "page": 1}

),

Document(

page\_content="To install a SIM card, insert the ejection tool into the SIM tray hole.",

metadata={"source": "iphone\_guide.pdf", "page": 10}

),

Document(

page\_content="Android is a mobile operating system developed by Google.",

metadata={"source": "android\_guide.pdf", "page": 1}

)

]

@patch('src.embeddings.openai\_embeddings.OpenAI')

@patch('src.llm.openai\_llm.ChatOpenAI')

def test\_full\_pipeline\_flow(self, mock\_chat\_openai, mock\_openai,

mock\_settings, sample\_documents):

"""Test the complete pipeline from loading to querying."""

# Setup mocks

mock\_openai\_client = Mock()

mock\_openai.return\_value = mock\_openai\_client

# Mock embeddings

mock\_openai\_client.embeddings.create.return\_value = Mock(

data=[Mock(embedding=[0.1, 0.2, 0.3, 0.4, 0.5])]

)

# Mock LLM

mock\_llm = Mock()

mock\_chat\_openai.return\_value = mock\_llm

with patch('config.settings.settings', mock\_settings):

# Initialize pipeline

pipeline = RAGPipeline()

# Mock document loading

with patch.object(pipeline.pdf\_loader, 'load\_multiple',

return\_value=sample\_documents):

# Load documents

num\_chunks = pipeline.load\_documents([Path("fake1.pdf"), Path("fake2.pdf")])

assert num\_chunks > 0

# Initialize QA chain

pipeline.initialize\_qa\_chain()

assert pipeline.qa\_chain is not None

# Mock QA chain response

with patch.object(pipeline.qa\_chain, 'run') as mock\_run:

mock\_run.return\_value = {

"query": "What is iPhone?",

"answer": "iPhone is a smartphone made by Apple.",

"source\_documents": sample\_documents[:2],

"metadata": {"processing\_time": 1.0}

}

# Answer question

response = pipeline.answer\_question("What is iPhone?")

assert response["answer"] == "iPhone is a smartphone made by Apple."

assert len(response["source\_documents"]) == 2

def test\_document\_processing\_pipeline(self, mock\_settings):

"""Test document loading and chunking process."""

with patch('config.settings.settings', mock\_settings):

# Create components

loader = PDFDocumentLoader()

chunker = DocumentChunker(chunk\_size=100, chunk\_overlap=20)

# Create test documents

test\_docs = [

Document(

page\_content="This is a test document. " \* 50, # Long content

metadata={"source": "test.pdf"}

)

]

# Process documents

chunks = chunker.process(test\_docs)

# Verify chunking

assert len(chunks) > 1 # Should create multiple chunks

assert all(len(chunk.page\_content) <= 120 for chunk in chunks) # Allow some flexibility

assert all(chunk.metadata["source"] == "test.pdf" for chunk in chunks)

@patch('src.embeddings.openai\_embeddings.OpenAI')

def test\_vector\_store\_persistence(self, mock\_openai, mock\_settings, sample\_documents):

"""Test vector store save and load functionality."""

# Setup mock embeddings

mock\_client = Mock()

mock\_openai.return\_value = mock\_client

mock\_client.embeddings.create.return\_value = Mock(

data=[Mock(embedding=[0.1, 0.2, 0.3])]

)

with patch('config.settings.settings', mock\_settings):

# Create embedding model and vector store

embedding\_model = OpenAIEmbeddingModel(api\_key="test-key")

vector\_store = ChromaVectorStore(

embedding\_model=embedding\_model,

persist\_directory=str(mock\_settings.vector\_store\_persist\_directory)

)

# Add documents

vector\_store.add\_documents(sample\_documents)

# Persist

vector\_store.persist()

# Create new vector store and load

new\_vector\_store = ChromaVectorStore(

embedding\_model=embedding\_model,

persist\_directory=str(mock\_settings.vector\_store\_persist\_directory)

)

new\_vector\_store.load()

# Verify loaded store works

with patch.object(new\_vector\_store.\_vectorstore, 'similarity\_search') as mock\_search:

mock\_search.return\_value = sample\_documents[:2]

results = new\_vector\_store.similarity\_search("test query")

assert len(results) == 2

@patch('src.embeddings.openai\_embeddings.OpenAI')

@patch('src.llm.openai\_llm.ChatOpenAI')

@patch('src.evaluation.ragas\_evaluator.evaluate')

def test\_evaluation\_integration(self, mock\_ragas\_evaluate, mock\_chat\_openai,

mock\_openai, mock\_settings):

"""Test the evaluation component integration."""

# Setup mocks

mock\_openai.return\_value = Mock()

mock\_llm = Mock()

mock\_chat\_openai.return\_value = mock\_llm

# Mock Ragas evaluation

mock\_result = Mock()

mock\_result.to\_pandas.return\_value = pd.DataFrame({

'context\_precision': [0.8],

'faithfulness': [0.9]

})

mock\_ragas\_evaluate.return\_value = mock\_result

with patch('config.settings.settings', mock\_settings):

# Create pipeline

pipeline = RAGPipeline()

pipeline.initialize\_qa\_chain()

# Mock QA chain

with patch.object(pipeline.qa\_chain, 'run') as mock\_run:

mock\_run.return\_value = {

"answer": "Test answer",

"source\_documents": []

}

# Run evaluation

eval\_data = [{

"question": "Test?",

"ground\_truth\_answer": "Ground truth"

}]

results = pipeline.evaluate(eval\_data)

assert "metrics" in results

assert "results\_dataframe" in results

@patch('main.argparse.ArgumentParser.parse\_args')

@patch('src.data\_loader.pdf\_loader.PyPDFLoader')

@patch('src.embeddings.openai\_embeddings.OpenAI')

@patch('src.llm.openai\_llm.ChatOpenAI')

def test\_main\_index\_mode(self, mock\_chat\_openai, mock\_openai,

mock\_pdf\_loader\_class, mock\_parse\_args,

mock\_settings, temp\_dir):

"""Test main function in index mode."""

# Create test PDF files

pdf1 = temp\_dir / "test1.pdf"

pdf2 = temp\_dir / "test2.pdf"

pdf1.write\_text("PDF content")

pdf2.write\_text("PDF content")

# Mock arguments

mock\_parse\_args.return\_value = Mock(

mode="index",

documents=[str(pdf1), str(pdf2)],

load\_existing=False

)

# Mock PDF loading

mock\_loader = Mock()

mock\_loader.load.return\_value = [

Document(page\_content="Test content", metadata={})

]

mock\_pdf\_loader\_class.return\_value = mock\_loader

# Mock OpenAI

mock\_openai.return\_value = Mock()

mock\_chat\_openai.return\_value = Mock()

with patch('config.settings.settings', mock\_settings):

with patch('sys.exit'):

from main import main

main()

# Verify PDF loader was called

assert mock\_pdf\_loader\_class.call\_count >= 1

def test\_error\_propagation(self, mock\_settings):

"""Test that errors are properly propagated through the pipeline."""

with patch('config.settings.settings', mock\_settings):

pipeline = RAGPipeline()

# Test with invalid file

with pytest.raises(Exception):

pipeline.load\_documents([Path("nonexistent.pdf")])

def test\_configuration\_loading(self, temp\_dir):

"""Test configuration loading from environment."""

env\_vars = {

"OPENAI\_API\_KEY": "test-key-123",

"LLM\_MODEL": "gpt-4",

"CHUNK\_SIZE": "500",

"LOG\_LEVEL": "DEBUG"

}

with patch.dict('os.environ', env\_vars):

# Create .env file

env\_file = temp\_dir / ".env"

env\_file.write\_text("EMBEDDING\_MODEL=text-embedding-3-small")

with patch('pathlib.Path.cwd', return\_value=temp\_dir):

settings = Settings()

assert settings.openai\_api\_key == "test-key-123"

assert settings.llm\_model == "gpt-4"

assert settings.chunk\_size == 500

assert settings.log\_level == "DEBUG"

@pytest.mark.parametrize("chain\_type,expected\_behavior", [

("stuff", "should work with small context"),

("map\_reduce", "should handle large documents"),

("refine", "should iteratively refine answer"),

])

def test\_different\_chain\_types(self, chain\_type, expected\_behavior,

mock\_settings):

"""Test different chain types in QA chain."""

with patch('config.settings.settings', mock\_settings):

with patch('src.embeddings.openai\_embeddings.OpenAI'):

with patch('src.llm.openai\_llm.ChatOpenAI'):

# Create components

embedding\_model = OpenAIEmbeddingModel(api\_key="test")

vector\_store = ChromaVectorStore(embedding\_model=embedding\_model)

llm = OpenAILLM(api\_key="test")

# Create QA chain with specific type

with patch('src.retrieval.qa\_chain.RetrievalQA'):

qa\_chain = QAChain(

llm=llm,

vector\_store=vector\_store,

chain\_type=chain\_type

)

assert qa\_chain.chain\_type == chain\_type

# Import pandas for the evaluation test

import pandas as pd