Financial RAG Pipeline Technical Documentation

Table of Contents

- Overview
- Architecture
- Core Components
- Method Analysis
- Error Handling Strategy

Overview

The **Financial RAG (Retrieval-Augmented Generation) Pipeline** is a document processing and question-answering system designed specifically for financial documents. It combines advanced PDF parsing, multi-modal content extraction, vector storage, and large language model integration to create an intelligent financial document analysis system.

Key Capabilities

- Multi-modal content extraction (text, tables, images)
- Intelligent chunking and summarization
- Vector-based semantic search
- Financial domain-specific querying
- Hybrid local/cloud model architecture

Architecture

The system follows given architecture format with distinct phases:

```
PDF Document → Content Extraction → Summarization → Vector Storage → Query Processing → Resp
```

Technology Stack

Document Processing Unstructured library is used for PDF parsing and content extraction.

Vector Storage ChromaDB is used for embeddings storage and retrieval.

Embeddings

For embedding Ollama (Llama3.2:3b) model embedding is used .

Local LLM

Local llm models Ollama (Llama3.2:3b, Gemma3:4b) are used for processing and summarization.

Framework

LangChain and Pipeline orchestration is used for proper rag pipeline.

Storage

LocalFileStore is used to store documents locally.

Core Components

1. FinancialRAGPipeline Class

Constructor (`__init__`)

```python
def __init__(self, pdf_path):
    self.pdf_path = pdf_path
    self.load_environment_variables()
    self.setup_models()
    self.setup_storage()
```

> **Purpose**: Initializes the pipeline with proper dependency injection

2. Environment Management

`load_environment_variables()`

```python
def load_environment_variables(self):
    try:
        load_dotenv()
        self.openai_api_key = os.getenv("OPENAI_API_KEY")
        if not self.openai_api_key:
            logger.warning("OPEN AI API not found in environment variables")
    except Exception as e:
        logging.error(f"Error loading environment variables {e}")
        raise
```

- This method loads environment variables as API keys are externalized from code and provides warning

Method Analysis

Content Extraction Methods

`extract_pdf_content()`

```python
def extract_pdf_content(self):
    # Primary extraction with chunking
    chunks = partition_pdf(
        filename = self.pdf_path,
        infer_table_structure= True,
        strategy = "hi_res",
        extract_image_block_types=['Image'],
        extract_image_block_to_payload= True,
```

```python
        chunking_strategy="by_title",
        max_characters = 7000,
        combine_text_under_n_chars = 2000,
        new_after_n_chars = 6000
    )

    # Secondary extraction for tables
    table_chunks = partition_pdf(
        filename = self.pdf_path,
        strategy = "hi_res"
    )
```

- This method implements dual pass extraction strategy where it first processes the entire document with intelligent chunking parameters (`max_characters=10000`, `combine_text_under_n_chars=2000`, `new_after_n_chars=6000`) to extract text and images while preserving document structure through `chunking_strategy="by_title"`. The second pass specifically targets table extraction using a separate `partition_pdf` call with `hi_res` strategy, which is essential because financial documents contain complex tabular data that requires specialized parsing. The method uses `infer_table_structure=True` and `extract_image_block_to_payload=True` to ensure comprehensive multimodal content capture, then delegates to specialized helper methods for content type segregation.

Content Type Segregation

`_get_texts(chunks)`

```python
def _get_texts(self, chunks):
    texts = []
    for chunk in chunks:
        if "CompositeElement" in str(type(chunk)):
            texts.append(chunk)
    return texts
```

- This helper method filters extracted chunks to identify text blocks by checking "CompositeElement" in chunk type string.

`_get_tables(chunks)`

```python
def _get_tables(self, chunks):
    tables = []
    for chunk in chunks:
        if chunk.category == 'Table':
            tables.append(chunk)
    return tables
```

- This helper method performs category based filtering by examining chunk.category attribute to identify table elements.

_get_images(chunks)

```
def _get_images(self, chunks):
    images_base64 = []
    for chunk in chunks:
        if hasattr(chunk, 'metadata') and hasattr(chunk.metadata,'orig_elements'):
            for element in chunk.metadata.orig_elements:
                if 'Image' in str(type(element)):
                    if hasattr(element.metadata, 'image_base64'):
                        images_base64.append(element.metadata.image_base64)
    return images_base64
```

- This helper method implements metadata traversal to extract base64 encoded images by navigating through complex nested structure of chunk.metadata.orig_elements. It looks for 'Image' type and extracts image_base64 attribute.

Summarization Engine

generate_summaries(texts, tables)

```
def generate_summaries(self, texts, tables):
    prompt_text = """
    You are an assistant tasked with summarizing tables and text.
    Give a concise summary of the table or text.

    Respond only with the summary, no additional comment.
    Do not start your message by saying "Here is a summary" or anything like that.
    Just give the summary as it is.

    Table or text chunk: {element}
    """

    prompt = ChatPromptTemplate.from_template(prompt_text)
    summarize_chain = {"element": lambda x:x} | prompt| self.local_model| StrOutputParser()

    # Batch processing with concurrency control
    text_summaries = summarize_chain.batch(texts, {"max_concurrency": 3})
    table_summaries = summarize_chain.batch(tables_text, {"max_concurrency":3})
```

- This method implements batch summarization pipeline to process both textual and tabular content from financial documents. It also implements Langchain pipe operator .

`generate_image_summaries(images)`

```python
def generate_image_summaries(self, images):
    image_prompt_template = """Describe the image in detail. For context,
    the image is a logo of APPLE company and it is the part of quarterly report of financial

    messages = [
        (
            "user",
            [
                {"type": "text", "text": image_prompt_template},
                {
                    "type": "image_url",
                    "image_url": {"url": "data:image/jpeg;base64,{image}"},
                },
            ],
        )
    ]
```

- This method generates textual description for list of images . It method constructs a messages object that combines text prompt and image(converted in base64) for chat based image analysis model. A `ChatPromptTemplate` is created from these messages, which is then connected in a pipeline (`chain`) to the `ChatOllama` model (here using `"gemma3:4b"`) and a `StrOutputParser` to produce plain text output. The `chain.batch(images)` call processes all images in a batch, generating summaries for each.

Storage Architecture

Vector Store Configuration

```python
def setup_storage(self):
    self.vector_store = Chroma(
        collection_name = "rag_collection",
        embedding_function = self.embeddings,
        persist_directory = "./chroma_langchain_db"
    )

    self.retriever = MultiVectorRetriever(
        vectorstore = self.vector_store,
        docstore = self.store,
        id_key = self.id_key
    )
```

- This `setup_storage` function is responsible for initializing and connecting all the storage components required for a retrieval-augmented generation (RAG) pipeline. First, it sets `self.id_key` to `"doc_id"`, which will act as a unique identifier for each document stored. Then, it initializes `self.store` as a `LocalFileStore`, which manages persistent storage of raw documents in the local directory `./local_docstore`. Next, it sets up `self.vector_store` as a Chroma vector database collection named `"rag_collection"`, which uses `self.embeddings` to convert documents into vector embeddings for semantic search, and persists the data in `./chroma_langchain_db`. The `self.retriever` is then initialized as a `MultiVectorRetriever`, which bridges the vector store and the document store, allowing efficient retrieval of documents using vector similarity while referencing the stored raw documents by `id_key`. The function wraps the setup in a `try-except` block to log and raise any initialization errors, ensuring failures in storage setup are captured immediately. Essentially, this function prepares all the underlying infrastructure for storing documents and performing semantic search in a RAG system.

Document Storage Strategy

```
def _store_text_documents(self, texts, texts_summaries):
    text_ids = [str(uuid.uuid4()) for _ in texts]

    # Create summary documents for vector search
    summary_texts = [
        Document(page_content = summary, metadata = {self.id_key: text_ids[i], "type":"text
        for i, summary in enumerate(texts_summaries)
    ]

    # Create full documents for retrieval
    full_texts = [
        Document(page_content = text.text, metadata = {self.id_key:text_ids[i],"type":"full
        for i,text in enumerate(texts)
    ]

    # Store in vector database and document store
    self.retriever.vectorstore.add_documents(summary_texts+full_texts)
    self.retriever.docstore.mset(list(zip(text_ids, encoded_texts)))
```

- **`store_documents` method**:
- stored_documents method is the main entry point for storing all types of documents—text, tables, and images—along with their summaries. It first checks whether each document type and its summaries exist, and if so, calls the respective helper method (`_store_text_documents`, `_store_table_documents`, `_store_image_documents`) to handle storage. The function ensures that all documents are processed and persisted in both the vector store (for semantic search) and the document store (for raw retrieval). Any errors during this process are logged and raised to prevent silent failures. This method centralizes the storage workflow,

making it easier to maintain and expand for different document types.

- **`_store_text_documents` method:**
This method stores text documents and their summaries. It generates a unique ID for each text using `uuid.uuid4()`, then creates `Document` objects for both the summaries and the full text with appropriate metadata (`type` and ID). The summaries and full text are added to the vector store for semantic search. The original text is encoded as UTF-8 and stored in the document store keyed by the generated IDs. Logging confirms how many text documents were successfully stored. This separation of summaries and full text allows for efficient retrieval either for concise or detailed queries.
- **`_store_table_documents` method:** this method handles tables and their summaries. Each table is assigned a unique ID, and `Document` objects are created for summaries (`"table summary"`) and full table text (`"full table text"`). The summaries and full text are added to the vector store, while the original table text (encoded in UTF-8) is stored in the document store with the corresponding IDs. Logging confirms successful storage. This method ensures both human-readable summaries and raw table data are preserved, enabling semantic search on content while keeping full data for exact retrieval.
- **`_store_image_documents` method:** This method stores images and their textual summaries. Each image gets a unique ID, and a `Document` is created for each image summary and added to the vector store for semantic retrieval. The raw images are encoded in UTF-8 and stored in the document store using the generated IDs. Unlike text and tables, only summaries are vectorized, since images themselves are stored as raw data. Logging tracks how many image documents were stored.

Query Processing

`query(question)`

```
def query(self, question):
    retrieved_data = self.vector_store.similarity_search(question, k = 3)

    if not retrieved_data:
        return "relavant information regarding your question not found"

    context = "\n\n".join([doc.page_content for doc in retrieved_data])

    chat_prompt = """
    You are an auditor/expert in reading financial statements.
    I will provide you report information. Based on information provide concise report.
    Context: {context}
    Question: {question}
    """
```

```
    prompt_chat = ChatPromptTemplate.from_template(chat_prompt)
    result_chain = prompt_chat| self.local_model | StrOutputParser()

    response = result_chain.invoke({"context": context,"question":question})
```

- **query method** This method processes a user's question using a retrieval-augmented approach. It first logs the query and performs a similarity search on the vector store to fetch the top 3 relevant documents. If no data is found, it returns a default message. Otherwise, it combines the retrieved content into a `context` string and constructs a prompt for the local language model, specifying the role of a financial auditor. The prompt is passed through a `ChatPromptTemplate` and `StrOutputParser` to generate a concise answer.

Model Selection Strategy

Local vs. Cloud Hybrid Approach

```
def setup_models(self):
    self.local_model = ChatOllama(temperature = 0.5, model = "llama3.2:3b")
    self.embeddings = OllamaEmbeddings(model = "llama3.2:3b")
    self.openai_model = ChatOpenAI(
        model = "gpt-5-mini-2025-08-07",
        temperature = 0,
        max_tokens  = None,
        timeout = None,
        max_retries = 2
    )
```

- The `setup_models` method initializes the AI models used in the RAG system. `self.local_model` is a ChatOllama instance (Llama 3.2, 3B parameters) for generating responses with moderate randomness (`temperature=0.5`). `self.embeddings` is an OllamaEmbeddings instance using the same Llama model to convert text into vector embeddings for semantic search. `self.openai_model` is a ChatOpenAI model (GPT-5-mini) configured for deterministic output (`temperature=0`), with no token limit or timeout and up to two retries. This setup provides a combination of local LLMs for embeddings and text generation, alongside a robust OpenAI model for high-quality language responses.

Error Handling Strategy

The code implements comprehensive error handling with specific strategies:

```
try:
    # Operation
    logger.info("Success message")
```

```python
except Exception as e:
    logger.error(f"Error context: {e}")
    raise  # or return fallback
```

Usage Guide

Basic Implementation

```python
# Initialize pipeline
pdf_path = 'financial_report.pdf'
pipeline = FinancialRAGPipeline(pdf_path)

# Process document
pipeline.run_pipelines()

# Query the system
answer = pipeline.query("What are the main revenue streams?")
print(answer)
```

Example Queries

```python
questions = [
    "What is net sales?",
    "What are the main financial highlights?",
    "What tables are available in the report?",
    "How did revenue change compared to last quarter?",
    "What are the key risk factors mentioned?"
]

for question in questions:
    print(f"\ Question: {question}")
    answer = pipeline.query(question)
    print(f" Answer: {answer}")
```