RAG Pipeline Documentation

Table of Contents

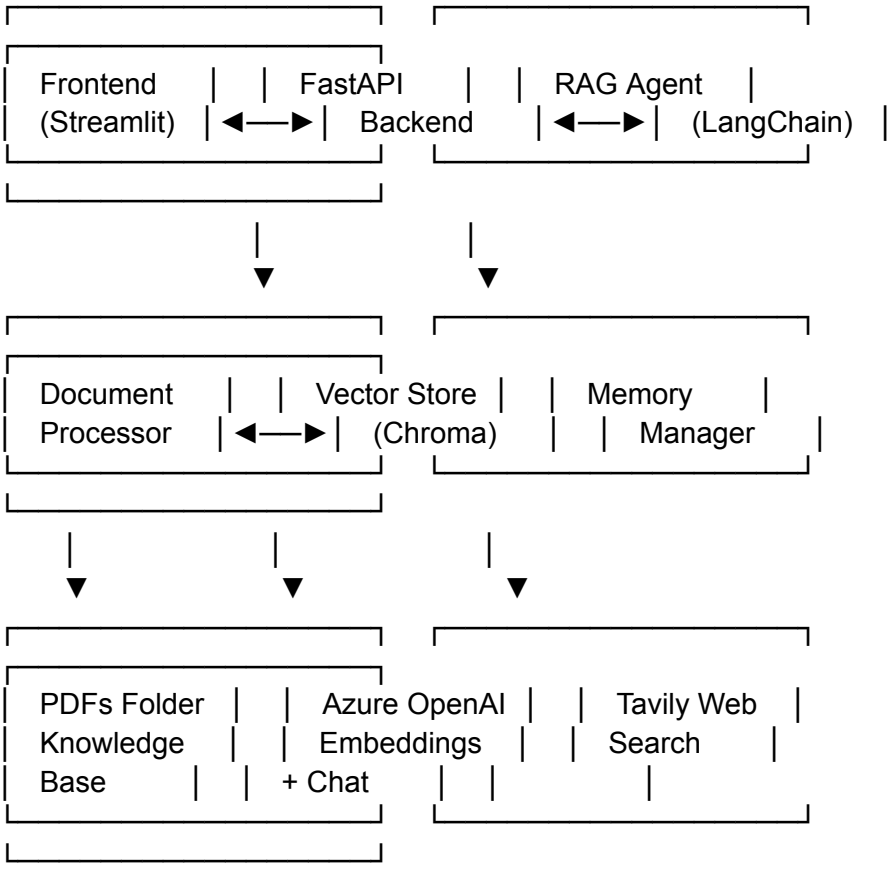
Overview

This is a production-ready Retrieval Augmented Generation (RAG) system that combines knowledge base search with web search capabilities. The system provides intelligent document retrieval, conversational memory, and fallback mechanisms to deliver accurate, contextual responses.

Key Features

- **Hierarchical Vector Storage**: Parent-child chunking for better context preservation
- **Intelligent Retrieval**: Query reformulation and LLM-based reranking
- **Conversational Memory**: Token-aware conversation management with summarization
- **Hybrid Search**: Knowledge base + web search fallback
- **Agent-based Architecture**: ReAct agent with tool selection
- **RESTful API**: FastAPI backend with comprehensive endpoints

Architecture



File Structure & Components

Core Components

1. `config.py` - Configuration Management

Purpose: Centralized configuration for all system components

Key Configurations:

- Azure OpenAI API settings (endpoint, key, API version)
- Model deployment names (chat, embeddings)
- Vector store paths and chunking parameters
- Memory management settings
- API server configuration

Environment Variables:

AZURE_OPENAI_API_KEY=your_key
AZURE_OPENAI_ENDPOINT=your_endpoint
CHAT_MODEL_DEPLOYMENT=chat-heavy
EMBEDDING_MODEL_DEPLOYMENT=embed-large
TAVILY_API_KEY=your_tavily_key

2. `embeddings.py` - Embedding Service

Purpose: Handles text embeddings using Azure OpenAI

Key Functions:

- `embed_documents()`: Batch embed multiple documents
- `embed_query()`: Embed single search queries
- `similarity_search()`: Calculate cosine similarity between embeddings

Technical Details:

- Uses `text-embedding-3-large` model
- 3072-dimensional embeddings
- Implements cosine similarity for vector search

3. `vectorstore.py` - Hierarchical Vector Storage

Purpose: Manages document storage with parent-child chunking strategy

Key Features:

- **Parent Chunks**: Larger context chunks (1500 chars)
- **Child Chunks**: Smaller searchable chunks (500 chars)
- **Hierarchical Retrieval**: Search children, retrieve parent context
- **Persistent Storage**: Uses Chroma with disk persistence

Architecture Benefits:

- Better context preservation
- Improved search precision
- Reduced information loss during chunking

4. `retriever.py` - Intelligent Retrieval System

Purpose: Advanced document retrieval with multiple enhancement techniques

Key Techniques:

- **Query Reformulation**: LLM generates alternative search queries
- **Multi-query Search**: Searches with original + reformulated queries
- **LLM-based Reranking**: Semantic reranking of search results
- **Deduplication**: Removes duplicate content across queries
- **Metadata Filtering**: Hybrid search with metadata constraints

Workflow:

1. Reformulate query into 3 variations
2. Search vector store with all queries
3. Deduplicate results by content
4. Rerank using LLM for relevance
5. Return top-k most relevant documents

5. `memory.py` - Conversation Memory Manager

Purpose: Manages conversation context with token-aware summarization

Key Features:

- **Session-based Memory**: Separate memory for each conversation
- **Token Counting**: Uses tiktoken for accurate token calculation
- **Auto-summarization**: Summarizes old messages when token limit exceeded
- **Context Formatting**: Formats conversation history for retrieval

Memory Strategy:

- Stores recent messages in full
- Summarizes older messages to save tokens
- Maintains conversation continuity
- Supports multiple concurrent sessions

6. `agent.py` - RAG Agent (Core Brain)

Purpose: Main orchestrator using ReAct (Reasoning + Acting) pattern

Tool Arsenal:

- **Knowledge Base Search**: Primary information source
- **Web Search**: Fallback for current/missing information

Decision Logic:

1. Always search knowledge base first
2. Use web search if KB results insufficient
3. Combine results intelligently

4. Provide confidence scoring

Agent Capabilities:

- Context-aware query processing
- Tool selection and execution
- Error handling and fallbacks
- Response confidence assessment

7. `utils.py` - Utility Functions

Purpose: Supporting utilities for document processing and system metrics

Key Components:

- **DocumentProcessor**: Load and process various file formats (PDF, TXT, DOCX, CSV, JSON)
- **QueryOptimizer**: Acronym expansion, stop word removal, keyword extraction
- **ResponseFormatter**: Format sources, create citations, highlight keywords
- **MetricsCollector**: Track system performance and usage statistics

8. `models.py` - Data Models

Purpose: Pydantic models for API request/response validation

Key Models:

- `QueryRequest`: User query with session and search preferences
- `QueryResponse`: Structured response with answer, sources, confidence
- `ChatMessage`: Individual conversation messages
- `MemoryContext`: Session conversation context
- `DocumentUpload`: File upload metadata

9. `main.py` - FastAPI Application

Purpose: RESTful API server providing HTTP interface

Startup Process:

1. Creates necessary directories
2. Loads knowledge base from PDFs folder
3. Initializes all components
4. Starts API server

Key Features:

- CORS middleware for web access
- Background task processing
- Comprehensive error handling
- Health check endpoints

RAG Workflow

1. Document Ingestion Flow

PDF Files → Document Loader → Text Splitter → Parent/Child Chunks → Embeddings → Vector Store → Knowledge Base Ready

2. Query Processing Flow

User Query → Memory Context → Query Reformulation → Multi-Search → Results Retrieval → Reranking → Response Generation → Memory Update

3. Detailed Query Workflow

1. **Input Processing**:
    - Receive user query and session ID
    - Retrieve conversation context from memory
    - Enhance query with conversation context
2. **Knowledge Base Search**:
    - Reformulate query into multiple variations
    - Search child chunks with all query variations
    - Retrieve parent chunks for context
    - Score and deduplicate results
3. **Decision Making**:
    - Evaluate knowledge base results quality
    - Check confidence scores against threshold
    - Decide whether web search is needed
4. **Agent Execution**:
    - Use ReAct agent with available tools
    - Execute knowledge base search first
    - Fallback to web search if needed
    - Combine and synthesize results
5. **Response Generation**:

    - Generate coherent response using LLM
    - Calculate confidence score
    - Format response with sources
    - Update conversation memory

Techniques & Technologies

RAG Techniques Used

1. **Hierarchical Chunking**:
    - Parent chunks (1500 chars) for context
    - Child chunks (500 chars) for precision
    - Maintains semantic coherence
2. **Query Enhancement**:
    - LLM-based query reformulation
    - Multiple query variations
    - Acronym expansion
    - Stop word handling
3. **Advanced Retrieval**:
    - Semantic similarity search
    - Hybrid search (semantic + keyword)
    - Multi-query retrieval
    - Result deduplication
4. **Intelligent Reranking**:
    - LLM-based semantic reranking
    - Relevance scoring
    - Context-aware ranking
5. **Memory Management**:
    - Conversation context preservation
    - Token-aware summarization
    - Session-based isolation

Technology Stack

Core Technologies:

- **LangChain**: Agent framework and RAG orchestration
- **Azure OpenAI**: LLM (chat) and embeddings
- **Chroma**: Vector database for document storage

- **Tavily**: Web search API for real-time information
- **FastAPI**: RESTful API framework
- **Pydantic**: Data validation and serialization

Supporting Libraries:

- **tiktoken**: Token counting for memory management
- **PyPDF**: PDF document processing
- **python-multipart**: File upload handling
- **uvicorn**: ASGI server for FastAPI

Setup & Installation

Prerequisites

- Python 3.8+
- Azure OpenAI account with deployed models
- Tavily API key for web search

Installation Steps

1. **Clone and Setup Environment**:

```
git clone <repository>
cd rag-pipeline
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
pip install -r requirements.txt
```

2. **Environment Configuration**: Create `.env` file:

```
AZURE_OPENAI_API_KEY=your_azure_openai_key
AZURE_OPENAI_ENDPOINT=https://your-resource.openai.azure.com/
AZURE_OPENAI_API_VERSION=2024-02-15-preview
CHAT_MODEL_DEPLOYMENT=gpt-4
EMBEDDING_MODEL_DEPLOYMENT=text-embedding-ada-002
TAVILY_API_KEY=your_tavily_api_key
VECTOR_DB_PATH=./vector_store
KNOWLEDGE_BASE_PATH=./pdfs
```

3. **Setup Knowledge Base**:

```
mkdir pdfs
```

Copy your PDF documents to the pdfs folder

4. **Run the Application**:

python main.py

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

API Endpoints

Core Endpoints

1. `GET /` - Health Check

Purpose: System status and configuration info **Response**:

```
{
  "status": "healthy",
  "service": "RAG Pipeline API",
  "version": "1.0.0",
  "data_source": "PDFs folder (./pdfs/)",
  "features": {
    "knowledge_base": "Active",
    "web_search": "Active"
  }
}
```

2. `POST /query` - Process Query

Purpose: Main query processing endpoint **Request**:

```
{
  "query": "How do I add a lead?",
  "session_id": "optional-session-id",
  "use_web_search": true
}
```

Response:

```
{
  "answer": "To add a lead in Studynet CRM...",
```

```
  "sources": [{"type": "knowledge_base", "content": "..."}],
  "confidence_score": 0.85,
  "web_search_used": false,
  "session_id": "session-uuid"
}
```

Document Management

3. `POST /upload/document` - Upload File

Purpose: Upload and process documents **Request**: Multipart form with file **Response**:

```
{
  "status": "success",
  "message": "Document processed successfully",
  "chunks_created": 15
}
```

4. `POST /upload/text` - Upload Text

Purpose: Add raw text content **Request**:

```
{
  "content": "Text content to add",
  "metadata": {"source": "manual_entry"}
}
```

Memory Management

5. `GET /memory/{session_id}` - Get Conversation

Purpose: Retrieve conversation history **Response**:

```
{
  "session_id": "session-uuid",
  "context": "User: Hello\nAssistant: Hi there!",
  "memory": {"chat_history": [...]}
}
```

6. `DELETE /memory/{session_id}` - Clear Session

Purpose: Clear conversation memory

7. `GET /sessions` - List Sessions

Purpose: Get all active conversation sessions

System Management

8. `GET /metrics` - System Metrics

Purpose: Get performance metrics **Response**:

```
{
  "queries_processed": 150,
  "avg_response_time": 2.3,
  "kb_hits": 120,
  "web_searches": 30,
  "errors": 2
}
```

9. `POST /metrics/reset` - Reset Metrics

Purpose: Clear system metrics

10. `GET /knowledge-base/status` - KB Status

Purpose: Knowledge base statistics **Response**:

```
{
  "status": "active",
  "parent_chunks": 245,
  "child_chunks": 1230,
  "total_documents": 1475
}
```

11. `POST /knowledge-base/reload` - Reload KB

Purpose: Reload knowledge base from PDFs folder

12. `DELETE /vectorstore/clear` - Clear Vector Store

Purpose: Clear entire vector database (destructive operation)

Configuration

Key Configuration Parameters

Chunking Strategy:

- `CHUNK_SIZE`: 500 (child chunk size)
- `CHUNK_OVERLAP`: 100 (overlap between chunks)
- `PARENT_CHUNK_SIZE`: 1500 (parent chunk size)

Retrieval Parameters:

- `RETRIEVER_K`: 5 (documents to retrieve)
- `RERANK_TOP_N`: 3 (documents after reranking)
- `SIMILARITY_THRESHOLD`: 0.7 (minimum similarity score)

Memory Management:

- `MAX_MEMORY_TOKENS`: 2000 (maximum tokens in memory)
- `CONVERSATION_BUFFER_SIZE`: 10 (message buffer size)

Usage Examples

Basic Query

```
import requests

response = requests.post("http://localhost:8000/query", json={
    "query": "How do I reset my password?",
    "use_web_search": True
})

print(response.json()["answer"])
```

Upload Document

```
with open("manual.pdf", "rb") as f:
    response = requests.post(
        "http://localhost:8000/upload/document",
        files={"file": f}
    )
```

Session Management

```
# Start conversation
response1 = requests.post("http://localhost:8000/query", json={
    "query": "What is RAG?"
})
session_id = response1.json()["session_id"]

# Continue conversation
response2 = requests.post("http://localhost:8000/query", json={
    "query": "How does it work?",
    "session_id": session_id
})
```

This documentation provides a comprehensive overview of the RAG pipeline system, covering architecture, components, workflows, and usage patterns. The system is designed for production use with proper error handling, monitoring, and scalability considerations.