Multi-Agent RAG System - Backend Documentation

Table of Contents

System Overview

The Multi-Agent RAG System is a FastAPI-based backend that orchestrates multiple specialized AI agents to overcome traditional RAG limitations. The system is designed to process complex queries across large document collections and provide comprehensive scientific paper analysis.

Key Features

- Document ingestion and summarization
- Query analysis and document matching
- Information synthesis across multiple documents
- Structured scientific paper analysis
- Asynchronous job processing
- Event-based status tracking

Core Components

- FastAPI web server
- CrewAI agent orchestration
- Document processing tools
- Embedding and similarity calculation
- Job management system

Setup and Installation

Prerequisites

- Python 3.8+
- Azure OpenAI API access
- Required Python packages

Installation Steps

1. Clone the repository:

```
git clone https://github.com/yourusername/multi-agent-rag.git
cd multi-agent-rag
```

2. Create and activate a virtual environment:

```
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. Install dependencies:

```
pip install -r requirements.txt
```

4. Create necessary directories:

```
mkdir -p logs
mkdir -p ~/Desktop/crew_docs/documents
mkdir -p ~/Desktop/crew_docs/summaries
mkdir -p ~/Desktop/crew_docs/summary_reports
```

Environment Configuration

Create a `.env` file in the root directory with the following variables:

```
# Azure OpenAI Configuration
AZURE_API_KEY=your_azure_api_key
AZURE_API_BASE=your_azure_endpoint
AZURE_API_VERSION=your_api_version

# Other Configuration Options
LOG_LEVEL=INFO
```

Required Azure OpenAI Models

- `gpt-4o`: For complex reasoning and synthesis tasks
- `gpt-4`: For detailed document analysis
- `text-embedding-ada-002`: For document and query embeddings

API Endpoints

Run Crew Analysis

- **Endpoint**: `/api/crew`
- **Method**: POST
- **Description**: Initiates document analysis or summary generation
- **Request Body**:
- `{ "user_query": "Your complex query here", "crew_type": "analysis" // or "summary"}`
- **Response**:
- `{ "job_id": "uuid-string"}`

Get Job Status

- **Endpoint**: `/api/crew/{job_id}`
- **Method**: GET
- **Description**: Retrieves the status and results of a job
- **Response**:
- `{ "job_id": "uuid-string", "status": "STARTED|COMPLETE|ERROR", "result": {}, "events": [ { "timestamp": "2025-05-01T12:34:56.789Z", "data": "Event description" } ]}`

Job Management

The system uses a job-based architecture for asynchronous processing:

1. When a request is received, a new job ID is generated (UUID)
2. The job is initialized in the `jobs` dictionary with a `STARTED` status
3. Background tasks perform the actual processing
4. Events are logged throughout the process using the `append_event` function
5. When processing is complete, the job status is updated to `COMPLETE` or `ERROR`
6. Results are stored in the job entry for retrieval

Job Status Codes

- `STARTED`: Job has been created and processing has begun
- `COMPLETE`: Job has finished successfully
- `ERROR`: An error occurred during processing

Agent System Architecture

The system implements two primary agent workflows:

Document Analysis Crew

- **Purpose**: Answer complex queries across multiple documents
- **Agents**:
    1. **Document Summary Agent**: Creates comprehensive document summaries

2. **Query Analysis Agent**: Analyzes queries and identifies relevant documents
3. **Document Analysis Agent**: Synthesizes information across documents

Document Summary Crew

- **Purpose**: Create detailed scientific paper summaries
- **Agents**:
    1. **Scientific Document Summary Agent**: Creates structured research paper summaries
    2. **Report Agent**: Formats summaries into well-organized reports

Agent Communication

Agents communicate through well-defined task inputs and outputs. Each agent has specific responsibilities and expertise areas, and the CrewAI framework orchestrates their interactions.

Document Processing Pipeline

Document Ingestion

1. Documents are placed in the `~/Desktop/crew_docs/documents` directory
2. Supported formats: PDF, TXT, DOCX
3. Documents are loaded using appropriate loaders based on file extension

Document Summarization

1. Text is extracted and cleaned (removing tabs, excess whitespace, bullets)
2. Documents are split into semantic chunks using `SemanticChunker`
3. Summaries are generated using the specified LLM
4. Structured information is extracted (for scientific papers)
5. Summaries are saved to `summaries.json` and as individual text files

Query Processing

1. User query is embedded using `text-embedding-ada-002`
2. Document summaries are embedded using the same model
3. Cosine similarity is calculated between query and document embeddings
4. Documents exceeding the similarity threshold (default: 0.76) are selected
5. Selected documents are synthesized to generate a comprehensive response

Troubleshooting

Common Issues

API Connection Errors

- Check Azure API key and endpoint configuration
- Verify network connectivity
- Ensure API rate limits haven't been exceeded

Document Processing Failures

- Check file formats and encoding
- Verify file permissions
- Check for corrupted or malformed documents

Out of Memory Errors

- Reduce batch sizes for document processing
- Implement pagination for large document collections
- Consider using a server with more RAM

Logging

- Logs are stored in the `logs` directory
- Log level can be configured in the `.env` file
- Each job maintains its own event log for detailed tracking

Performance Considerations

Optimizing Document Processing

- Limit document size to optimize token usage
- Consider preprocessing documents to remove irrelevant content
- Use semantic chunking with appropriate overlap settings

Scaling Considerations

- The current implementation uses in-memory job storage, which is suitable for moderate workloads
- For production deployments, consider implementing:
    - Database-backed job storage
    - Message queue for job processing
    - Container-based deployment for horizontal scaling

Cost Management

- Monitor token usage to control API costs
- Implement caching strategies for frequently accessed documents

- Consider using smaller models for initial preprocessing steps

---

This documentation covers the core functionality of the Multi-Agent RAG System. For additional details or specific implementation questions, please refer to the source code or contact the system administrator.