# Multi-Agent RAG System - Full-Stack Integration Guide

## Overview

This document provides a comprehensive guide to the full-stack architecture of the Multi-Agent RAG System, explaining how the frontend and backend components work together to create a powerful research document analysis platform.

## System Architecture

The system follows a modern client-server architecture:

1. **Frontend** : Next.js application providing user interface
2. **Backend** : FastAPI service orchestrating AI agents
3. **External** : Azure OpenAI API and filesystem storage

### Architectural Principles

- **Separation of Concerns**: Clear boundaries between presentation, business logic, and data access
- **Asynchronous Processing**: Long-running tasks handled via background processing
- **Event-Driven Communication**: Real-time status updates through polling and events
- **Modular Agent Design**: Specialized AI agents with focused responsibilities

## Data Flow

### 1. Query Submission

1. User enters a query and selects crew type in the frontend
2. Frontend sends a POST request to `/api/crew` with query and crew type
3. Backend generates a job ID and initializes a job in the job manager
4. Backend starts a background task to process the request
5. Job ID is returned to the frontend for status tracking

### 2. Job Processing

1. Backend creates the appropriate crew based on front-end user choice (Analysis or Summary)
2. Crew orchestrates its agents to perform document processing
3. Each agent performs its specialized task using tools
4. Events are logged throughout the process

5. Results are accumulated and stored in the job entry

### 3. Status Updates

1. Frontend polls the `/api/crew/{job_id}` endpoint at regular intervals
2. Backend returns current job status, events, and any available results
3. Frontend updates its UI with the latest information
4. When the job completes, frontend displays the final result and stops polling

# Integration Points

## API Contract

### Start Job Endpoint

**Request:**

```
POST /api/crew
Content-Type: application/json

{
  "user_query": "string",  // Required for analysis crew
  "crew_type": "analysis|summary"  // Crew type selection
}
```

**Response:**

```
{
  "job_id": "uuid-string"
}
```

### Job Status Endpoint

**Request:**

```
GET /api/crew/{job_id}
```

**Response:**

```
{
  "job_id": "uuid-string",
  "status": "STARTED|COMPLETE|ERROR",
  "result": {
    "user_query": "string",
    "result": "string"
  },
  "events": [
    {
      "timestamp": "ISO-8601 datetime",
      "data": "string or object"
    }
  ]
}
```

## Data Structures

### Job Object

```
{
  status: string;       // Current job status
  events: Event[];      // List of job events
  result: string;       // Final job result (JSON string)
}
```

### Event Object

```
{
  timestamp: DateTime;  // When the event occurred
  data: string | any;   // Event data (string or structured object)
}
```

### Result Object

```
// For Analysis Crew
{
  user_query: string;   // Original user query
  result: string;       // Analysis result (often markdown text)
}

// For Summary Crew
{
  result: string;       // Summary result (often JSON string)
}
```

# Deployment Configuration

## Development Environment

**Frontend**:

- Server: `http://localhost:3000`
- Environment: `.env.local` with `NEXT_PUBLIC_API_URL=http://localhost:3001`

**Backend**:

- Server: `http://localhost:3001`
- Environment: `.env` with Azure OpenAI credentials

### Production Considerations

1. **Environment Variables**:
   - Store sensitive credentials in secure environment variables
   - Use production URLs for API endpoints
2. **CORS Configuration**:
   - Update backend CORS settings to allow only production frontend origin
   - Set appropriate headers for security
3. **Scaling**:
   - Deploy frontend to a static hosting service (Vercel, Netlify)
   - Deploy backend to a container orchestration platform (Kubernetes, Docker Swarm)
   - Implement message queue for job processing (RabbitMQ, Redis, Azure Service Bus and Azure SQL)
   - Use containerized workers for document processing

## Local Development Setup

### Prerequisites

- Node.js 16+ for frontend
- Python 3.8+ for backend
- Azure OpenAI API access

### Start Development Environment

1. **Start Backend**:

```
cd backend
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
pip install -r requirements.txt
uvicorn fast_crew_api:app --host 0.0.0.0 --port 3001
```

2. **Start Frontend**:

```
cd frontend
npm install
npm run dev
```

3. Access the application at `http://localhost:3000`

## Best Practices

### Error Handling

1. **Frontend**:
    - Use toast notifications for user-friendly error messages
    - Implement error boundaries for component-level error handling
    - Handle network errors gracefully with retries and fallbacks
2. **Backend**:
    - Return appropriate HTTP status codes
    - Provide meaningful error messages
    - Log detailed error information for debugging

### Performance Optimization

1. **Frontend**:
    - Optimize bundle size with code splitting
    - Implement memoization for expensive computations
    - Use pagination or virtualization for large event logs
2. **Backend**:
    - Implement caching for frequently accessed resources
    - Use efficient document processing algorithms
    - Optimize database queries with proper indexing

### Security Considerations

1. **API Security**:
    - Implement rate limiting
    - Add authentication for production environments
    - Validate and sanitize all inputs
2. **Data Security**:
    - Ensure secure storage of sensitive documents
    - Implement proper access controls
    - Consider encryption for sensitive data

## Extending the System

### Adding New Crew Types

1. **Backend Changes**:
    - Create new agent classes in `agents.py`
    - Add new task definitions in `tasks.py`
    - Create a new crew class in `crew.py`
    - Update API endpoint to handle the new crew type
2. **Frontend Changes**:
    - Add the new crew type to the dropdown in `InputSection.tsx`
    - Update the useCrewJob hook to handle any crew-specific logic
    - Add specialized rendering for the new crew's results if needed

### Adding New Document Types

1. **Backend Changes**:
   - Add new document loaders in the tool implementations
   - Update text processing for the new document type
   - Add specialized handling for format-specific features
2. **Frontend Changes**:
   - Add support for uploading the new document type
   - Implement any specialized visualization for the new format

## Troubleshooting

### Common Issues

1. **CORS Errors**:
   - Ensure backend CORS middleware is properly configured
   - Check that frontend is using the correct API URL
2. **Job Processing Timeouts**:
   - Increase the timeout settings in axios requests
   - Implement checkpoint saving for long-running jobs
3. **Document Processing Failures**:
   - Check file format compatibility
   - Verify document encoding is supported
   - Ensure document size is within token limits

### Debugging Tools

1. **Frontend**:
   - React Developer Tools for component inspection
   - Network tab in browser DevTools for API requests
   - Console logging with specific tags for tracking
2. **Backend**:
   - Request logging middleware
   - Debug-level logging for detailed operations
   - Interactive API documentation via Swagger UI

## Reference Documentation

- Backend Documentation
- Frontend Documentation
- Azure OpenAI API Documentation
- CrewAI Documentation
- FastAPI Documentation
- Next.js Documentation