Intelligent Agentic RAG System Documentation

This document provides detailed technical documentation for the Python-based Intelligent Agentic RAG (Retrieval Augmented Generation) system. It covers its architecture, key components, data flow, configuration, and setup instructions. This system is exposed via a Flask REST API, which is then consumed by the .NET backend.

1. Overview

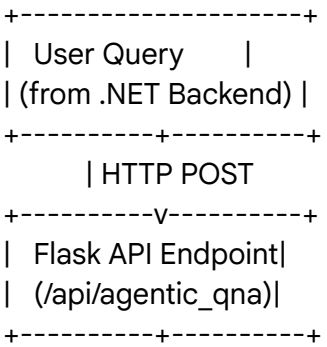
The Intelligent Agentic RAG system is designed to provide accurate, contextually relevant, and confident answers to user queries by leveraging both internal knowledge bases and external (LLM's general knowledge) sources. It employs an "agentic" approach, meaning it makes decisions on how to best answer a query based on its classification and the availability/quality of information. A crucial aspect is its caching mechanism to improve efficiency and responsiveness for repeated or similar queries.

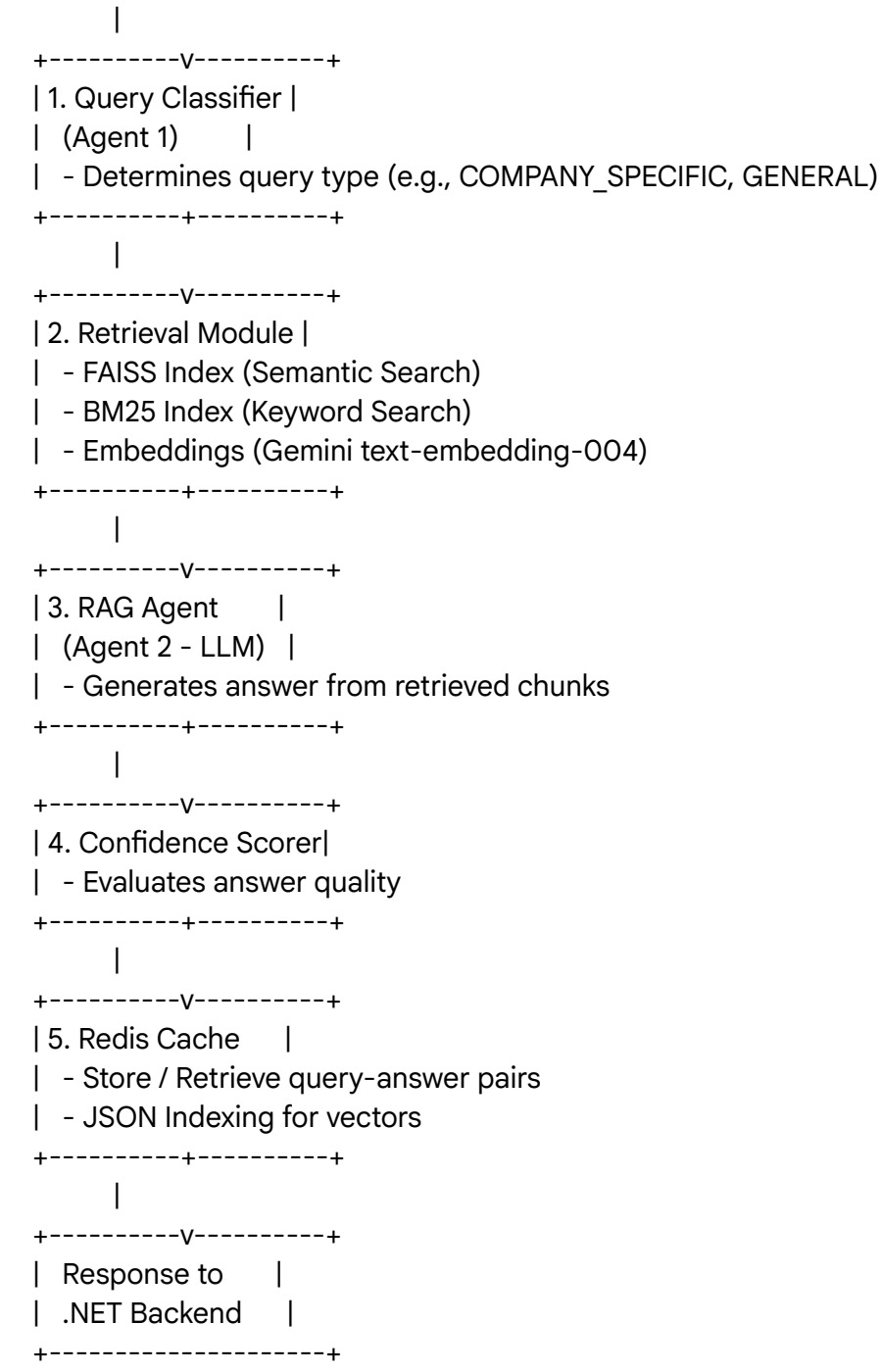
Core Capabilities:

- **Query Classification:** Determines the nature of an incoming query (e.g., company-specific, general knowledge).
- **Hybrid Retrieval:** Combines semantic (vector) search with keyword (sparse) search for comprehensive document retrieval.
- **Contextual Generation:** Uses a Large Language Model (LLM) to synthesize answers from retrieved information.
- **Confidence Scoring:** Evaluates the quality and trustworthiness of the generated answer.
- **Caching:** Stores query-answer pairs in Redis to serve subsequent similar queries quickly.

2. Architecture

The RAG system operates as a pipeline, with several interconnected components working in sequence to process a query. It's exposed as a Flask API, making it a microservice that the .NET backend interacts with.





2.1. Agentic Flow (Paths Taken)

The system is designed to follow different "paths" based on query classification and retrieval success:

- **Company_Specific_RAG_Success:** Query classified as company-specific, and an answer was successfully generated from internal documents.
- **Company_Specific_RAG_Failure:** Query classified as company-specific, but no

relevant information was found, or the LLM could not generate a confident answer from internal documents.
- **Internet_Search_Success:** Query classified as general, and an answer was successfully generated using external (simulated) internet search.
- **Internet_Search_Failure:** Query classified as general, but external search failed or yielded no confident answer.
- **General_QA_LLM_Knowledge:** Query answered directly by the LLM using its pre-trained general knowledge, without specific document retrieval.
- **Ambiguous_Query:** The query classifier or RAG agent determined the query was unclear or required more context.
- **No_Answer_Found:** No path could yield a satisfactory answer.

3. Key Components and Modules

3.1. Query Classifier (Agent 1)

- **Purpose:** Determines the intent or domain of the user's query.
- **Mechanism:** Uses an LLM (Gemini gemini-2.0-flash) with a specific prompt to classify the query into predefined categories (e.g., COMPANY_SPECIFIC, GENERAL, AMBIGUOUS).
- **Output:** A classification label.

3.2. Embedding Model

- **Model:** Google Gemini text-embedding-004.
- **Purpose:** Converts text (user queries, document chunks) into dense vector representations (embeddings). These embeddings are crucial for semantic similarity search.
- **Integration:** Used for generating embeddings for:
    - Incoming user queries (for caching and retrieval).
    - Document chunks (for building FAISS index).

3.3. Retrieval Module (Hybrid Search)

- **Purpose:** Finds the most relevant document chunks from the knowledge base based on the user's query.
- **Components:**
    - **FAISS Index (Semantic Search):** Stores vector embeddings of document chunks. Used for finding semantically similar chunks (Approximate Nearest Neighbor search).
        - **File:** cache/faiss_index.bin
        - **Metadata:** cache/faiss_metadata.json (maps FAISS IDs to original document content and metadata).
    - **BM25 Index (Keyword Search):** A sparse retrieval model that excels at keyword matching.

        - **File:** cache/bm25_index.pkl
    - **Hybrid Approach:** Combines results from both FAISS and BM25 to get a more comprehensive set of relevant chunks. Weights can be applied to prioritize one over the other.

3.4. RAG Agent (Agent 2 - LLM for Generation)

- **Model:** Google Gemini gemini-2.0-flash.
- **Purpose:** Takes the user's query and the retrieved document chunks (context) to generate a concise and accurate answer.
- **Mechanism:** Prompts the LLM with the query and context, guiding it to extract information from the provided chunks.
- **Output:** The generated answer text and citations (sources).

3.5. Confidence Scoring

- **Purpose:** Quantifies the reliability of the generated answer.
- **Mechanism:** Calculates a score based on various factors, including:
    - **Semantic Similarity (Query-Answer):** How semantically close the generated answer is to the original query.
    - **Retrieval Score:** The relevance scores of the retrieved chunks (e.g., cosine similarity from FAISS, BM25 scores).
    - **Path Taken:** Different paths might inherently have different confidence baselines.
- **Output:** A float value (e.g., 0.0 to 1.0), indicating confidence.

3.6. Redis Cache Manager

- **Purpose:** Stores query-answer pairs to provide fast responses for repeated or semantically similar queries, reducing LLM calls and processing time.
- **Technology:** Redis Stack with RediSearch module.
- **Indexing Strategy:** Uses **JSON indexing** for storing cache entries.
    - Cache entries are stored as JSON documents (e.g., rag_cache:timestamp).
    - The query's embedding is stored as a JSON array (list of floats) within the document.
    - RediSearch's VECTOR_RANGE query is used to find similar query embeddings.
- **Cache Hit Logic:** A query is considered a cache hit if its embedding is sufficiently similar (above CACHING_SIMILARITY_THRESHOLD) to a previously cached query's embedding.
- **Cache Storage Logic:** Answers are stored in the cache only if they come from a "cacheable" agent path (e.g., Company_Specific_RAG_Success) and their calculated confidence score meets a minimum threshold (RAG_CONFIDENCE_FOR_CACHING).
- **Configuration:** Redis host, port, database, password, and index name are configured via environment variables.

4. Data Flow: Processing a Query

1. **Incoming Request:** A user query arrives at the Flask API's /api/agentic_qna endpoint (typically from the .NET backend).
2. **Query Embedding:** The incoming query is immediately converted into a vector embedding using text-embedding-004.
3. **Cache Lookup:** The system first attempts to find a similar query in the Redis cache using vector similarity search.
    - If a sufficiently similar query (above CACHING_SIMILARITY_THRESHOLD) is found, the cached answer is returned immediately.
4. **Cache Miss - Agentic Path:** If no cache hit:
    - **Query Classification (Agent 1):** The query is passed to an LLM to classify its type (e.g., COMPANY_SPECIFIC, GENERAL).
    - **Conditional Retrieval/Generation:**
        - **If COMPANY_SPECIFIC:**
            - **Hybrid Retrieval:** Relevant chunks are retrieved from the internal FAISS and BM25 indexes.
            - **RAG Generation (Agent 2):** The LLM generates an answer using the query and the retrieved chunks as context.
        - **If GENERAL:**
            - (Simulated) Internet Search: The query might be sent to an external tool or LLM for general knowledge.
            - LLM Generation: The LLM generates an answer based on its general knowledge or the simulated search results.
    - **Confidence Scoring:** After an answer is generated, a confidence score is calculated based on semantic similarity and retrieval scores.
    - **Cache Storage:** If the answer's generation path is "cacheable" and its confidence score meets RAG_CONFIDENCE_FOR_CACHING, the query, its embedding, and the full answer data are stored in the Redis cache.
5. **Response:** The final answer, confidence score, sources, and the agent's path taken are returned as a JSON response.

5. Configuration

The RAG system relies on environment variables for sensitive information and configuration.

- **GEMINI_API_KEY**: Your API key for accessing Google Gemini models.
- **REDIS_HOST**: Hostname for your Redis Stack instance (e.g., localhost).
- **REDIS_PORT**: Port for Redis (e.g., 6379).
- **REDIS_DB**: Redis database number (e.g., 0).
- **REDIS_PASSWORD**: Password for Redis (if applicable, leave empty string if none).
- **REDIS_INDEX_NAME**: Name of the RediSearch index for caching (e.g., rag_cache_index).
- **FAISS_INDEX_FILE**: Path to the saved FAISS index file (e.g., cache/faiss_index.bin).
- **FAISS_METADATA_FILE**: Path to the FAISS metadata JSON file (e.g., cache/faiss_metadata.json).

- **BM25_INDEX_FILE**: Path to the saved BM25 index file (e.g., cache/bm25_index.pkl).
- **VECTOR_DIMENSION**: The dimension of the embeddings generated by your model (e.g., 768 for text-embedding-004).
- **CACHING_SIMILARITY_THRESHOLD**: Minimum cosine similarity for a cache hit (e.g., 0.8).
- **RAG_CONFIDENCE_FOR_CACHING**: Minimum confidence score for a RAG answer to be stored in cache (e.g., 0.55).

6. Setup & Running

6.1. Prerequisites

- **Python 3.9+:** Recommended for compatibility.
- **Pip:** Python package installer.
- **Docker:** For running Redis Stack (PostgreSQL is for .NET backend).
- **NLTK Data:** punkt tokenizer is required.

6.2. Installation

1. **Clone the repository:** (Assuming your RAG code is in a repository)
   git clone <your-rag-repo-url>
   cd <your-rag-repo-directory>

2. **Create a virtual environment:**
   python -m venv venv
   source venv/bin/activate # On Windows: venv\Scripts\activate

3. **Install Python dependencies:**
   pip install -r requirements.txt

   (Ensure your requirements.txt includes flask, redis, numpy, scikit-learn, sentence-transformers, nltk, google-generativeai, python-dotenv, rank_bm25, faiss-cpu or faiss-gpu).
4. **Set Environment Variables:** Create a .env file in the root of your RAG project and populate it with the configuration variables listed in Section 5.
   GEMINI_API_KEY="YOUR_GEMINI_API_KEY"
   REDIS_HOST="localhost"
   REDIS_PORT="6379"
   REDIS_DB="0"
   REDIS_PASSWORD="" # Or your password
   REDIS_INDEX_NAME="rag_cache_index"
   FAISS_INDEX_FILE="cache/faiss_index.bin"
   FAISS_METADATA_FILE="cache/faiss_metadata.json"
   BM25_INDEX_FILE="cache/bm25_index.pkl"

VECTOR_DIMENSION="768"
CACHING_SIMILARITY_THRESHOLD="0.8"
RAG_CONFIDENCE_FOR_CACHING="0.55"

5. **Run Redis Stack:**
   docker run -d --name redis-stack-server -p 6379:6379 redis/redis-stack:latest

6. **Prepare Knowledge Base (if not already done):**
   - You'll need a separate script or process to:
     - Load your internal documents (e.g., PDFs, text files).
     - Chunk them appropriately.
     - Generate embeddings for these chunks using text-embedding-004.
     - Build and save the FAISS index (faiss_index.bin, faiss_metadata.json).
     - Build and save the BM25 index (bm25_index.pkl).
   - Ensure these files are located at the paths specified in your .env file.
   - **Crucial:** After preparing the knowledge base, run FLUSHDB on Redis to ensure the cache starts clean and the JSON index is correctly created by the Flask app on startup.
     docker exec -it redis-stack-server redis-cli FLUSHDB

6.3. Running the Flask API

Navigate to your RAG project directory in the terminal (with your virtual environment activated) and run the Flask application:
python your_flask_app_file.py # e.g., python app.py or python flask_api.py

The Flask API will typically start on http://localhost:8000.

7. Testing the Flask API

You can test the Flask API directly using curl or Postman/Insomnia.

7.1. Example Request

Send a POST request to http://localhost:8000/api/agentic_qna with a JSON body.
First Query (Expected Cache Miss, then storage):

```
curl -X POST \
  http://localhost:8000/api/agentic_qna \
  -H 'Content-Type: application/json' \
  -d '{
    "query": "how many days of sick leave can I take?",
    "conversation_id": "test_conv_123",
    "conversation_history": []
  }'
```

Second Query (Expected Cache Hit):
Run the exact same curl command again immediately. You should see cached: true in the response and Cache hit! in your Flask application's logs.
Example with Conversation History:

```
curl -X POST \
  http://localhost:8000/api/agentic_qna \
  -H 'Content-Type: application/json' \
  -d '{
    "query": "What about casual leave?",
    "conversation_id": "test_conv_123",
    "conversation_history": [
      {"role": "user", "content": "how many days of sick leave can I take?"},
      {"role": "agent", "content": "You are entitled to 12 days of Sick Leave in a calendar year."}
    ]
  }'
```

7.2. Expected Response Structure

The Flask API will return a JSON object similar to this:

```
{
  "agent_path_taken": "Company_Specific_RAG_Success",
  "answer": "Employees earn 2 days of leave for every month of service, totaling 24 days per calendar year, which includes 12 days of Casual Leave and 12 days of Sick Leave (Source: Employee Handbook - 202343.pdf, Page 13).",
  "cache_similarity": 0.9999999403953552, // Present if cached
  "cached": true, // True if served from cache, False otherwise
  "confidence_score": 0.5200288934603754,
  "sources": [
    {
      "content": "...",
      "page_number": 13,
      "source": "Employee Handbook - 202343.pdf",
      "type": "Internal Document"
    }
  ]
}
```

8. Future Enhancements

- **Advanced Agent Orchestration:** Implement more complex decision-making logic for the agent, potentially using tools (e.g., calculator, calendar API).

- **Streaming Responses:** Implement server-sent events (SSE) or WebSockets for real-time streaming of LLM responses.
- **Fine-tuning/RAG Optimization:** Continuously evaluate and improve retrieval quality, chunking strategies, and LLM prompting.
- **External Tool Integration:** Integrate with actual external search APIs (e.g., Google Search API) or other domain-specific tools.
- **User Feedback Loop:** Implement mechanisms for users to provide feedback on answer quality, which can be used to improve the RAG system.
- **Observability:** Add more detailed metrics, tracing, and logging for better monitoring and debugging in production.
- **GPU Acceleration:** Utilize faiss-gpu for faster vector search if a compatible GPU is available.
- **Scalability:** Consider deploying Flask API with a WSGI server (Gunicorn, uWSGI) and using a load balancer for horizontal scaling.