Team Member: Shuo Deng(sd3715)&Shifei Zheng(sz3196)

Professor Name: Thomas Woo

Subject Name: [ELENE 6770] TOPICS IN NETWORKING

Project Type: Implementation Project

10 December 2024

Cloud-Based Retrieval-Augmented Generation (RAG) System: Enabling Knowledge Base Creation and LLM-Driven Question Answering

Abstract:

The Retrieval-Augmented Generation (RAG) system is a powerful solution for processing unstructured documents and delivering intelligent, context-aware answers. Leveraging technologies like MinIO for scalable storage, Redis for task management, and OLLAMA's Llama 3.1 and 3.2 models for embedding generation, it ensures efficient knowledge retrieval and semantic understanding. Deployed on Google Cloud with containerized services, the system features robust task orchestration, optimized resource management, and seamless user interaction, making it a scalable and reliable architecture for intelligent query-answering tasks.

Software Components:

The Retrieval-Augmented Generation (RAG) system is designed to efficiently process unstructured documents and provide intelligent, context-aware answers to user queries. By integrating advanced technologies across storage, processing, and retrieval layers, it transforms raw data into structured knowledge. The following sections detail the system's core components and their roles in achieving this functionality.

Storage and Task Management Layer. The foundation of the RAG system lies in its robust storage and task management infrastructure. This layer utilizes MinIO, a high-performance, scalable object storage system, to handle large document files and intermediate processing results. Documents are retrieved efficiently in binary format to support downstream tasks like chunking. Complementing this, Redis serves as a real-time in-memory database and message

broker, managing task queues to ensure smooth scheduling, task execution, and status tracking. Together, MinIO and Redis provide the essential groundwork for the system's seamless data flow and asynchronous processing.

Document Processing Layer. This layer is responsible for transforming raw documents into structured, analyzable data. It leverages OCR (Optical Character Recognition) tools, such as Tesseract, to extract text from scanned PDFs and images. Custom-built document parsers handle the complexities of text layout, tables, and hierarchical document structures, while table recognition modules extract structured data from tabular formats. The build_chunks process organizes documents into manageable pieces, combining OCR and layout analysis to ensure the proper segmentation of titles, headings, and content. This layer ensures that raw data is preprocessed into a form optimized for embedding and retrieval.

Embedding and Knowledge Representation Layer. To enable semantic understanding and efficient retrieval, this layer converts document chunks into high-dimensional vector embeddings using OLLAMA with Llama 3.1 and 3.2 models. These models offer state-of-the-art performance in embedding generation and keyword extraction, with faster inference and lower resource consumption compared to other LLMs. The embeddings are stored in a vector index, often powered by systems, enabling efficient semantic search and similarity matching. By leveraging Llama models, the system achieves higher accuracy and contextual enrichment for downstream tasks like query answering.

Task Management Layer. Orchestrating the system’s processing workflow, this layer ensures tasks are executed in an organized and efficient manner. Built with Python, it includes modules for managing task creation, progress updates, and cancellations. Redis plays a crucial role in tracking task statuses and dispatching them to the appropriate executors. The handle_task function continuously monitors the task queues, enabling seamless execution of tasks such as document chunking, embedding generation, and retrieval. This layer provides the backbone for handling concurrent processing and maintaining system stability.

Retrieval-Augmented Generation (RAG) Layer. At the heart of the system's intelligent query answering capabilities, the RAG layer combines advanced retrieval and generation techniques. RAPTOR, a custom clustering and retrieval algorithm, organizes document chunks hierarchically

for efficient knowledge retrieval. Multi-way recall methods identify the most relevant chunks from the vector index, which are then re-ranked to prioritize accuracy. Finally, LLMs transform the retrieved chunks into coherent and contextually appropriate answers. This layer bridges the gap between unstructured data and precise, human-readable responses.

API and Query Analysis Layer. Serving as the system's interface between the frontend and backend, the API layer processes user inputs and manages task flow. Built with frameworks like Python FastAPI or Flask, the API interprets queries through a Query Analyze module, identifying the required backend processes such as chunk retrieval or document parsing. Nginx acts as a reverse proxy, optimizing web traffic management and endpoint communication. By efficiently dispatching tasks to the processing layers, this layer ensures a responsive and streamlined interaction with the system.

Front-End Layer. The front-end layer of the RAG system is built using TypeScript, chosen over traditional JavaScript or React.js for its static typing and enhanced maintainability. TypeScript's strong type system reduces runtime errors and simplifies debugging, making it ideal for a complex, large-scale project like RAG. Styled with Tailwind CSS, the interface is clean and professional, allowing users to upload documents, submit queries, and receive context-rich answers effortlessly. This approach ensures reliability and scalability while bridging backend complexity with a user-friendly design.

Logging and Monitoring. To ensure system reliability and scalability, the logging and monitoring layer captures detailed execution logs and tracks real-time system performance. Python Logging records task-level details, while Redis Monitoring keeps tabs on task queue health and system metrics like memory usage. The report_status module provides continuous updates on task execution progress and resource utilization. This layer is crucial for debugging, performance optimization, and maintaining the system's robustness under varying workloads.

Summary. Each layer in the RAG system integrates specific technologies tailored to its function. From MinIO's scalable storage to Redis's real-time task management, OCR's document parsing to LLM-powered embeddings, the architecture is designed for performance, scalability, and precision. By combining these technologies seamlessly, the system delivers a robust solution for intelligent document processing and query answering, as shown below.

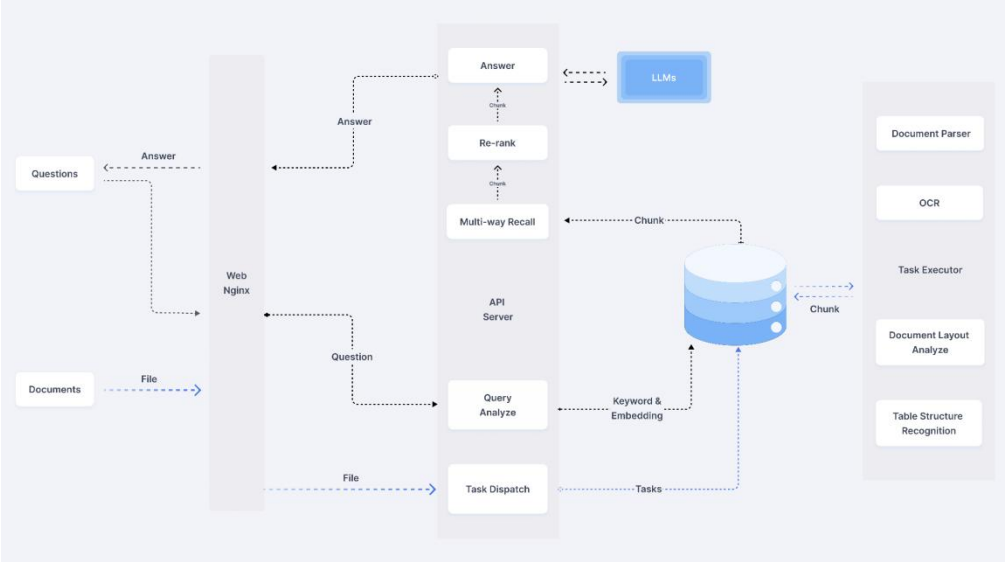

Fig.1 System Architecture

Deployment on Cloud

We deploy the application onto a server running on Google Cloud. We create a VM with 8v CPU, 20 GB of memory, and 64 GB of disk space. We also configured a firewall rule related to port 11434 to expose the Ollama container to certain traffic.

Since this application needs intensive third party services (redis DB, mySQL, MinIO, etc), we use docker to containerize these services, ensuring consistency across different environments, simplifying deployment, and managing dependencies efficiently. Ollama is also applied to run open-source large language models, since it integrates model weights, configurations, and data into a single package as well as optimizes setup and configurations, including GPU usage.

We use Redis for our RAG application because it is a NoSQL database that efficiently handles diverse data types and supports high-throughput I/O operations.

The project implements client connection and load control through multiple NGINX mechanisms. Global concurrency control is managed by setting worker_processes auto and worker_connections 1024, allowing a maximum of 4096 concurrent connections (4 core， 8v core), preventing server overload. Request size is controlled by client_max_body_size 128M, which limits the maximum request size to avoid excessive resource usage. keepalive_timeout 65

maintains connections efficiently while preventing idle connections from holding resources for too long. Load balancing is achieved using proxy_pass http://ip:9380 to forward API requests to the backend server, with additional configurations potentially distributing load and managing timeouts effectively. Finally, gzip compression is enabled to minimize the size of transmitted files, reducing network load, particularly for large text files. These combined strategies ensure stable and efficient handling of client requests and server resources.

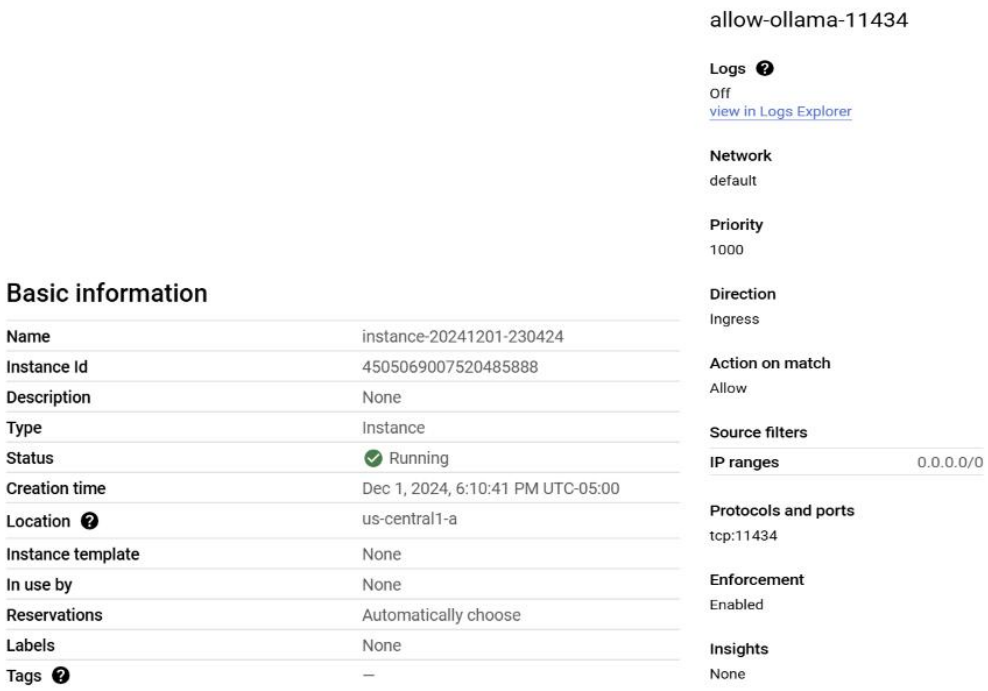

Fig.2 VM Instance Configuration

Works Cited

[1] https://github.com/infiniflow/ragflow

[2] https://github.com/NabeelGef/InformationRetrevial

[3] https://ragflow.io/docs/dev/launch_ragflow_from_source

[4] https://github.com/Chenjie669/ragflow-google-cloud-setup