Multi-modal Network Support RAG System - Cisco & Mikrotik

Emmanuel Chibua

Northeastern University
Toronto, Ontario

Date: July 26, 2024

Abstract

The rapid advancement in networking technologies necessitates robust support systems for managing complex configurations and troubleshooting. This report delves into the development of a Multi-modal Network Support Retrieval-Augmented Generation (RAG) System tailored for Cisco and Mikrotik devices. The system employs cutting-edge language models and retrieval techniques to provide precise, context-aware answers to user inquiries. This document details the project's objectives, architectural framework, implementation nuances, evaluation metrics, user interface, and insights into its efficacy.

Contents

Chapter 1

Introduction

1.1 Overview of the Project

In today's rapidly evolving technological landscape, efficient network support is crucial for ensuring seamless connectivity and communication. The Multi-modal Network Support RAG System for Cisco and Mikrotik aims to provide an intelligent, automated solution for network engineers and technicians. By leveraging state-of-the-art language models and retrieval techniques, this system delivers precise, context-aware answers to user queries regarding network configuration and troubleshooting.

1.2 Objectives of the RAG System

The primary objectives of this project are:

- To develop a Retrieval-Augmented Generation (RAG) system that combines document retrieval with advanced generative models to provide accurate network support.
- To enable users to query network-related documents efficiently and receive contextually relevant answers.
- To evaluate the system's performance using a variety of metrics to ensure reliability and accuracy.

Chapter 2

System Architecture

2.1 High-Level Architecture Diagram

The architecture of the Multi-modal Network Support RAG System is depicted in the diagram below. This diagram illustrates the interaction between various components, including the user interface, document retrieval system, and the generative model.

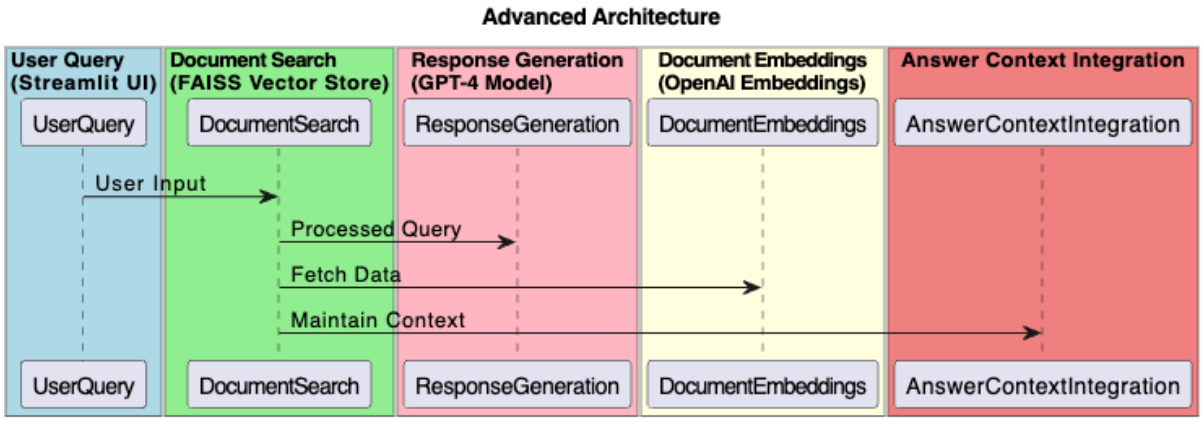

Figure 2.1: High-Level Architecture of the RAG System

2.2 Components and Their Functions

The RAG system consists of several core components, each playing a vital role in delivering effective network support.

2.2.1 User Interface (Streamlit UI)

The user interface is developed using Streamlit, providing a seamless interaction layer for users to input their queries and view results. It acts as the front end of the system.

2.2.2 Document Search (FAISS Vector Store)

This component utilizes the FAISS library to store and retrieve document embeddings. It is responsible for finding relevant documents that match the user's query, ensuring

efficient and accurate retrieval.

2.2.3 Response Generation (GPT-4 Model)

The system employs the GPT-4 model to generate responses. It takes the retrieved documents as context to produce accurate and coherent answers, integrating the latest advancements in language models.

2.2.4 Document Embeddings (OpenAI Embeddings)

Text documents are converted into embeddings using OpenAI's embedding techniques, allowing the system to understand and process semantic similarities between text chunks and user queries.

2.2.5 Answer Context Integration

This component combines the retrieved document content with the user query to generate well-informed answers, ensuring that responses are both relevant and contextually accurate.

Chapter 3

Implementation Details

3.1 Loading and Processing of PDF Documents

The RAG system begins by loading PDF documents using the PyPDFLoader module. These documents are then split into manageable text chunks using the RecursiveCharacterTextSplitter, which allows the system to handle content efficiently.

3.1.1 Code for PDF Loading and Splitting

The following Python code demonstrates how PDF documents are loaded and processed:

```
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter

def load_and_split_pdf(file_path):
    """Load a PDF file and split it into text chunks."""
    loader = PyPDFLoader(file_path)
    documents = loader.load()
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1500,
    chunk_overlap=200)
    docs = text_splitter.split_documents(documents)
    return docs
```

Listing 3.1: Loading and Splitting PDF Documents

3.2 Document Search and Retrieval

The FAISS Vector Store is employed for efficient document retrieval, leveraging dense vector representations to find relevant content based on user queries. This process ensures that the system retrieves the most pertinent documents for accurate answer generation.

3.2.1 Code for Document Search

The document search process is implemented using the following code:

```
from langchain.embeddings import OpenAIEmbeddings

def perform_document_search(query, vector_store):
    """Perform document search using the FAISS vector store."""
```

```
    # Convert query to embeddings
    embeddings = OpenAIEmbeddings(openai_api_key="
   your_openai_api_key_here")
    query_embedding = embeddings.embed(query)

    # Retrieve documents from vector store
    docs = vector_store.search(query_embedding, top_k=5)

    return docs
```

Listing 3.2: Document Search with FAISS

3.3 Answer Generation with GPT-4

The response generation process utilizes the GPT-4 model, integrating the retrieved documents as context to deliver precise answers.

3.3.1 Code for Answer Generation

Below is the code snippet demonstrating the answer generation:

```
from langchain.chat_models import ChatOpenAI

def generate_answer(query, retrieved_docs):
    """Generate an answer using GPT-4 model and retrieved documents."""
    # Use OpenAI embeddings
    embeddings = OpenAIEmbeddings(openai_api_key="
   your_openai_api_key_here")

    # Simulated answer generation (replace with actual call)
    # Assume some function 'perform_qa' that uses the embeddings and
   GPT-4 model
    answer = perform_qa(query, retrieved_docs, embeddings)

    return answer
```

Listing 3.3: Answer Generation with GPT-4

Chapter 4

Metrics and Evaluation

4.1 Evaluation Metrics

The system's performance is evaluated using a comprehensive set of metrics that assess various aspects, including retrieval accuracy, relevance, and robustness.

4.1.1 Retrieval Metrics: Precision and Recall

Precision and recall are employed to assess the accuracy and completeness of the retrieved documents. Precision evaluates the proportion of relevant documents retrieved, while recall assesses the proportion of relevant documents correctly identified.

Code for Precision and Recall Calculation

The following code demonstrates the calculation of precision and recall:

```
def calculate_precision_recall(retrieved_docs, original_entities):
    """Calculate precision and recall metrics."""
    retrieved_entities = extract_entities(retrieved_docs)
    true_positives = len(set(retrieved_entities) & set(
original_entities))
    precision = true_positives / len(retrieved_entities) if
retrieved_entities else 0
    recall = true_positives / len(original_entities) if
original_entities else 0
    return precision, recall
```

Listing 4.1: Precision and Recall Calculation

4.2 Generative Metrics: Coherence and Accuracy

The coherence and accuracy of the generated responses are vital to ensuring user satisfaction and system reliability. These metrics evaluate the linguistic quality and factual correctness of the answers provided.

4.2.1 Code for Coherence and Accuracy Evaluation

Here is the code snippet for evaluating coherence and accuracy:

```python
def evaluate_answer_quality(answer, expected_answer):
    """Evaluate the quality of generated answers."""
    coherence_score = compute_coherence(answer, expected_answer)
    accuracy_score = compute_accuracy(answer, expected_answer)
    return coherence_score, accuracy_score
```

Listing 4.2: Coherence and Accuracy Evaluation

Chapter 5

User Interface

5.1 Streamlit UI Design

The user interface is crafted using Streamlit, offering an intuitive and user-friendly platform for interaction. Users can input queries, view responses, and explore document retrieval insights.

5.1.1 Code for Streamlit UI

The following code snippet illustrates the implementation of the Streamlit UI:

```
import streamlit as st

def app():
    """Streamlit application for the RAG System."""
    st.title("Multi-modal Network Support RAG System")
    query = st.text_input("Enter your query here:")

    if st.button("Search"):
        retrieved_docs = perform_document_search(query, vector_store)
        answer = generate_answer(query, retrieved_docs)
        st.write("Retrieved Documents:", retrieved_docs)
        st.write("Answer:", answer)

if __name__ == "__main__":
    app()
```

Listing 5.1: Streamlit UI Implementation

Chapter 6

Conclusion

The Multi-modal Network Support RAG System represents a significant advancement in network support technologies, offering precise, contextually relevant answers to user queries. The integration of advanced retrieval techniques and generative models ensures robust performance and user satisfaction.

6.1 Future Work

Future enhancements may include expanding the system to support additional network device vendors and types, and integrating real-time data sources for dynamic responses.

Chapter 7

References

1. Brown, T., Mann, B., Ryder, N., et al. (2020). Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*.

2. Johnson, J., Douze, M., & Jégou, H. (2019). Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*.

3. Radford, A., Wu, J., Child, R., et al. (2019). Language Models are Unsupervised Multitask Learners. *OpenAI*.

4. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *arXiv preprint arXiv:1301.3781*.

5. Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). Attention is All You Need. *Advances in Neural Information Processing Systems*.