Chat-bot Assistant with RAG Architecture

Overview

This project is a chatbot web application designed to provide responses to user queries by utilizing context from a set of training documents. The application follows the RAG (Retrieval-Augmented Generation) architecture, leveraging a combination of neural language models and vector-based document retrieval to deliver accurate and contextually relevant responses. The system comprises three main components: a UI app, a server app, and an LLM (Large Language Model) service.

System Components

1. **UI App**

- **Framework**: React
- **Function**: Provides a user interface for interacting with the chatbot. The UI app sends user queries to the server app and displays responses generated by the LLM.
- **Features**:
  - Responsive chat interface.
  - Query input and response display.
  - Handles multiple conversations by tracking session states in the frontend.

2. **Server App**

- **Purpose**: Manages client requests, ensures efficient request processing, and coordinates with the LLM service to retrieve responses.
- **Key Technologies**:
  - **RabbitMQ**: Utilized for managing multiple requests and supporting asynchronous communication. RabbitMQ enables the server app to handle high concurrency by queueing requests and ensuring each is processed efficiently.
  - **Cache Management**: Caches responses to repeated queries to reduce response times for frequently asked questions, improving user experience.
- **Functionality**:
  - Receives and queues incoming queries from the UI app.
  - Checks the cache for any stored responses to similar queries to optimize processing.
  - Forwards queries to the LLM service if no cached response is found.
  - Returns responses to the UI app after processing.

3. **LLM Service**

- **Framework**: Flask-based app
- **Model**: Utilizes the Gemini language model to generate responses and perform embedding generation for document retrieval.
- **Database**: ChromaDB, a vector database, is used for efficient storage and retrieval of embeddings.

- **Workflow**:
  - When a query is received from the server app, the LLM service generates embeddings of the query using the Gemini model.
  - Embeddings are then compared against a pre-generated vector index of training document embeddings stored in ChromaDB to find the most relevant documents.
  - Relevant documents are passed to the Gemini model to generate context-rich responses.
- **Advantages**:
  - The Gemini model's embeddings facilitate precise document retrieval, ensuring relevant and accurate responses.
  - ChromaDB's vector database allows for high-speed retrieval and scalability.

Architecture

The system's architecture follows the RAG design, ensuring contextually accurate responses by augmenting the generative model's outputs with retrieved documents. The architecture is divided into three layers:

1. **Presentation Layer**: The React-based UI app serves as the user-facing interface, facilitating query input and response presentation.
2. **Application Layer**: The server app manages backend processing, load balancing, and caching. RabbitMQ ensures the system can handle multiple requests by queuing tasks and enabling efficient resource utilization.
3. **Data Layer**: The LLM service, incorporating the Gemini model and ChromaDB, is responsible for both document retrieval and response generation.

Workflow Summary

1. **User Query Submission**: A user inputs a query in the React-based UI app.
2. **Server Request Handling**: The server app receives the query, checks the cache, and, if necessary, forwards it to the LLM service.
3. **Document Retrieval and Response Generation**:
   - The LLM service generates an embedding for the query.
   - ChromaDB retrieves relevant document embeddings for context.
   - The Gemini model uses these documents to generate a response.
4. **Response Delivery**: The generated response is sent back to the server app, cached if needed, and then displayed on the UI app.

Conclusion

This RAG-based chatbot web app combines powerful document retrieval with advanced language generation, enabling efficient and contextually aware responses. By leveraging RabbitMQ for request handling, caching, and a scalable vector database with the Gemini model, the application is designed to serve a high volume of user queries accurately and responsively.