A Major Project Report on

SQL-RAG : ENHANCED SQL QUERY GENERATION WITH RAG

Submitted in Partial fulfillment of requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

In

INFORMATION TECHNOLOGY

By

KARTHIKEYA KOLAHAL	**21BD1A1225**
KAPARDHI KANNEKANTI	**21BD1A1224**
VISSA VENKATA KARTHIKEYA ADITYA	**21BD1A1264**
ABHINAV VENGALA	**21BD1A1262**

Under the guidance of

Ms. B. Manasa
Assistant Professor, Department of IT

DEPARTMENT OF INFORMATION TECHNOLOGY

KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
(AN AUTONOMOUS INSTITUTION)
Accredited by NBA & NAAC, Approved by AICTE, Affiliated to JNTUH.
Narayanaguda, Hyderabad, Telangana-29

2024-25

KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
(AN AUTONOMOUS INSTITUTION)
Accredited by NBA & NAAC, Approved by AICTE, Affiliated to JNTUH.
Narayanaguda, Hyderabad, Telangana-500029

DEPARTMENT OF INFORMATION TECHNOLOGY

CERTIFICATE

This is to certify that this is a Bonafide record of the project report titled **“SQL-RAG: ENHANCED SQL QUERY GENERATION WITH RAG”** which is being presented as the Major Project report by

1. KARTHIKEYA KOLAHAL	**21BD1A1225**
2. KAPARDHI KANNEKANTI	**21BD1A1224**
3. VISSA VENKATA KARTHIKEYA ADITYA	**21BD1A1264**
4. ABHINAV VENGALA	**21BD1A1262**

In partial fulfillment for the award of the degree of Bachelor of Technology in Information Technology affiliated to the Jawaharlal Nehru Technological University Hyderabad, Hyderabad

Internal Guide
(Ms. B. Manasa)

Head of Department
(Dr. G. Narender)

Submitted for Viva Voce Examination held on ______________________

External Examiner

Vision of KMIT

- To be the fountainhead in producing highly skilled, globally competent engineers.
- Producing quality graduates trained in the latest software technologies and related tools and striving to make India a world leader in software products and services.

Mission of KMIT

- To provide a learning environment that inculcates problem solving skills, professional, ethical responsibilities, lifelong learning through multi modal platforms and prepares students to become successful professionals.
- To establish an industry institute Interaction to make students ready for the industry.
- To provide exposure to students with the latest hardware and software tools.
- To promote research-based projects/activities in the emerging areas of technology convergence.
- To encourage and enable students to not merely seek jobs from the industry but also to create new enterprises.
- To induce a spirit of nationalism which will enable the student to develop, understand India's challenges and to encourage them to develop effective solutions.
- To support the faculty to accelerate their learning curve to deliver excellent service to students.

Vision of IT Department

To produce globally competent graduates to meet the modern challenges
through contemporary knowledge and moral values committed to build a vibrant
nation.

Mission of IT Department

- To create an academic environment, which promotes the intellectual and professional development of students and faculty.
- To impart skills beyond university prescribed to transform students into a well-rounded IT professional.
- To nurture the students to be dynamic, industry ready and to have multidisciplinary skills including e-learning, blended learning and remote testing as an individual and as a team.
- To continuously engage in research and projects development, strategic use of emerging technologies to attain self-sustainability.

PROGRAM OUTCOMES (POs)

1. **Engineering Knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **Problem Analysis:** Identify formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences

3. **Design/Development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct Investigations of Complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern Tool Usage:** Create select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The Engineer and Society:** Apply reasoning informed by contextual knowledge to societal, health, and safety. Legal und cultural issues and the consequent responsibilities relevant to professional engineering practice.

7. **Environment and Sustainability:** Understand the impact of professional engineering solutions in societal and environmental contexts and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics:** Apply ethical principles and commitment to professional ethics and responsibilities and norms of engineering practice.

9. **Individual and Team Work:** Function effectively as an individual, and as a member or leader in diverse teams and in multidisciplinary settings.

10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as being able to comprehend and write effective reports and design documentation, making effective presentations, and give and receive clear instructions.

11. **Project Management and Finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Life-Long Learning:** Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1: An ability to analyze the common business functions to design and develop appropriate Information Technology solutions for social upliftment.

PSO2: Shall have expertise on the evolving technologies like Python, Machine Learning, Deep learning, IOT, Data Science, Full stack development, Social Networks, Cyber Security, Mobile Apps, CRM, ERP, Big Data, etc.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO1: Graduates will have successful careers in computer related engineering fields or will be able to successfully pursue advanced higher education degrees.

PEO2: Graduates will try and provide solutions to challenging problems in their profession by applying computer engineering principles.

PEO3: Graduates will engage in life-long learning and professional development by rapidly adapting to the changing work environment.

PEO4: Graduates will communicate effectively, work collaboratively and exhibit high levels of professionalism and ethical responsibility.

PROJECT OUTCOMES

P1: Non-technical users can retrieve insights from complex databases through natural language, reducing reliance on technical staff and promoting data-driven decision-making across all roles.

P2: Optimal usage of AI and RAG techniques to generate optimized SQL queries and get desired results. It dynamically chooses the best search method for accurate results.

P3: By automating query generation and execution, the system significantly shortens the time between a user's question and actionable insights, accelerating business workflows and responsiveness.

P4: The hybrid search mechanism intelligently combines keyword and semantic search, leading to higher relevance in query results and reducing the risk of misinformed decisions.

MAPPING PROJECT OUTCOMES WITH PROGRAM OUTCOMES

PO	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
P1	H	H		H	H	L			H	H	M	
P2	H	H		H	H				H			M
P3	H	H		H	H				H	H	M	
P4	H	H		H	H				H		M	H

L – LOW M –MEDIUM H– HIGH

PROJECT OUTCOMES MAPPING PROGRAM SPECIFIC OUTCOMES

PSO	PSO1	PSO2
P1	L	M
P2		H
P3		M
P4		H

PROJECT OUTCOMES MAPPING PROGRAM EDUCATIONAL OBJECTIVES

PEO	PEO1	PEO2	PEO3	PEO4
P1		H		H
P2	M	H	H	M
P3		M		H
P4	M	H	H	H

DECLARATION

We hereby declare that the results embodied in the dissertation entitled **“SQL-RAG: ENHANCED SQL QUERY GENERATION WITH RAG"** has been carried out by us together during the academic year 2024-25 as a partial fulfillment of the award of the B.Tech degree in Information Technology from JNTUH. We have not submitted this report to any other university or organization for the award of any other degree.

STUDENT NAME	**ROLL NO**
KARTHIKEYA KOLAHAL	**21BD1A1225**
KAPARDHI KANNEKANTI	**21BD1A1224**
VISSA VENKATA KARTHIKEYA ADITYA	**21BD1A1264**
ABHINAV VENGALA	**21BD1A1262**

ACKNOWLEDGEMENT

We take this opportunity to thank all the people who have rendered their full support for our project work. We render our thanks to **Dr. B L Malleswari**, Principal, who encouraged us to do the Project.

We are grateful to **Mr. Neil Gogte**, Founder & Director, **Mr. S. Nitin,** Director, for facilitating all the amenities required for carrying out this project.

We express our sincere gratitude to **Ms. Deepa Ganu**, Director Academic, for providing an excellent environment in the college.

We are also thankful to **Dr. G. Narender**, Head of the Department, for providing us with time to make this project a success within the given schedule.

We are also thankful to our guide **Ms. B. Manasa**, for her valuable guidance and encouragement given to us throughout the project work.

We would like to thank the entire IT Department faculty, who helped us directly and indirectly in the completion of the project.

We sincerely thank our friends and family for their constant motivation during the project work.

ABSTRACT

SQL-RAG is a system designed to enhance SQL query generation using Retrieval-Augmented Generation (RAG). As organizations increasingly rely on databases for decision-making, extracting meaningful insights efficiently becomes challenging. To address this, we propose a system that integrates retrieval-based techniques with generative AI to improve SQL query formulation and execution.

Our system leverages natural language processing / generative AI to generate optimized SQL queries based on user input. It retrieves relevant schema and query examples from a knowledge base before generating precise and context-aware SQL statements. This approach ensures that even non-expert users can interact with databases seamlessly, reducing the need for extensive SQL expertise.

The development involves techniques such as text embedding, vector search, and large language models to enhance query generation and retrieval. By improving SQL query accuracy and usability, SQRAG has the potential to streamline data analytics, business intelligence and decision-making across various industries.

LIST OF DIAGRAMS

Sl. No	**Name of the Diagram**	**Page No.**
1.	Architecture Diagram	6
6 **2.**	Use Case Diagram	19
3.	Sequence Diagram	20
4.	Class Diagram	21
5.	Activity Diagram	22

LIST OF SCREENSHOTS

CONTENTS

LIST OF ABBREVIATIONS

1. **SRS -** Software Requirement Specification
2. **STLC -** Software Test Life Cycle
3. **SDLC -** Software Development Life Cycle
4. **HTML -** Hyper Text Markup Language
5. **CSS -** Cascading Style Sheets
6. **W3C -** World Wide Web Consortium
7. **UML -** Unified Modelling Language
8. **UI -** User Interface
9. **SQL -** Structured Query Language
10. **OOP -** Object Oriented Program

CHAPTER - 1

1. INTRODUCTION

1.1 Purpose of Project

SQRAG enhances SQL accessibility by allowing users, even those without SQL expertise, to generate accurate and efficient SQL queries using natural language. Instead of requiring users to manually write complex SQL statements, SQL-RAG leverages Retrieval-Augmented Generation (RAG) to understand their intent. This makes database interaction more intuitive, reducing the learning curve for non-technical users while ensuring precise query formulation.

The system ensures that query generation is both accurate and context-aware. As a result, organizations can save time while extracting meaningful insights from their data without needing deep knowledge of existing database.

Furthermore, SQLRAG seamlessly integrates with PostgreSQL. Its modular design allows easy adoption into existing database systems, making it a practical tool for various industries. SQRAG enhances data retrieval accuracy and facilitates better decision-making. By automating SQL generation and improving query relevance, SQRAG helps organizations maximize their database utility while minimizing complexity.

1.2 Problems with Existing Systems

Existing systems for SQL query generation and database interaction have several limitations that reduce their effectiveness and accessibility:

1. **Limited Natural Interaction** – Many SQL-based systems require users to write queries manually, which can be challenging for those without SQL expertise. Even tools that support natural language queries often struggle with accuracy, leading to frustration and incorrect results.
2. **Lack of Context Awareness** – Traditional SQL assistants or auto-complete tools lack domain-specific knowledge and do not retrieve relevant schema or query examples before

generating SQL. This results in generic or incomplete queries that may not align with the database structure or user intent.

3. **Inefficient Query Handling** – Most systems rely solely on keyword-based search, which fails to understand the true intent behind a query. This often leads to irrelevant results or inefficient queries that do not optimize database performance.
4. **Integration Challenges** – Many solutions are not designed to integrate seamlessly with existing databases, requiring complex setup processes or additional configurations, making adoption difficult for organizations.

By addressing these limitations, SQL-RAG enhances SQL query generation with a hybrid search approach, contextual retrieval, and intelligent decision-making, ensuring more accurate and user-friendly database interactions.

1.3 Proposed System

The proposed system, SQRAG, is designed to improve SQL query generation by integrating retrieval-based techniques with generative AI. Instead of requiring users to manually write SQL queries, SQRAG enables natural language interaction, allowing users to describe their data needs in simple terms. The system then retrieves relevant schema and past query examples before generating optimized SQL queries, ensuring accuracy and relevance.

To enhance query retrieval and execution, SQRAG employs a hybrid search approach, combining semantic search (to understand intent) with keyword-based search (for precise matching). This ensures that results are both contextually relevant and syntactically correct. Additionally, LangGraph based agents dynamically decide the best retrieval method, adjust queries when errors occur, and optimize execution for efficiency. These intelligent agents allow the system to adapt to different query complexities and database structures.

SQRAG seamlessly integrates with PostgreSQL, utilizing pgvector for efficient vector storage and similarity search. It is designed for easy adoption in existing database systems, making it a practical solution for businesses and organizations that rely on SQL databases. By automating SQL generation, improving accuracy, and reducing complexity, SQRAG empowers both technical and non-technical users to extract meaningful insights from their data efficiently.

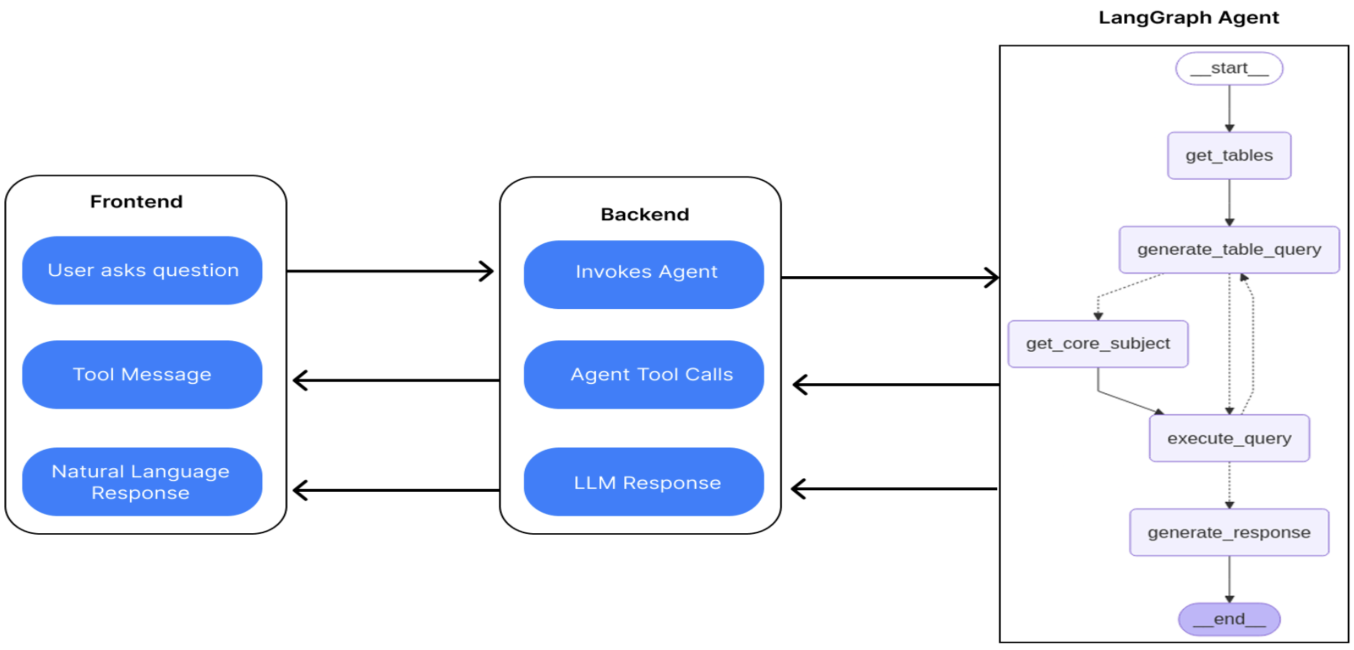
1.4 Scope of the Project

SQRAG aims to make SQL query generation accessible to everyone in an organization, eliminating the dependency on database service teams for fetching data. Instead of submitting requests for data and waiting for responses, employees can use natural language to generate accurate SQL queries instantly, ensuring users get precise and optimized SQL statements without requiring SQL expertise.

With SQRAG, organizations can significantly reduce the time spent on data retrieval, allowing employees to make faster, data-driven decisions. Additionally, its modular design ensures scalability and adaptability, making it a valuable tool across various industries that rely on structured data.

The system integrates semantic and keyword-based search, along with LangGraph-based agents, to dynamically decide the best retrieval and query generation strategy. Built for PostgreSQL, SQRAG can seamlessly integrate into existing database infrastructures, making it suitable for data analysis, reporting, and business intelligence. Future enhancements could include support for multiple database engines, improved query optimization techniques, and integration with enterprise analytics tools to further enhance decision-making efficiency.

1.5 Architecture Diagram



CHAPTER - 2

2. LITERATURE SURVEY

- **Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks** : Lewis et al. (2020) introduced the Retrieval-Augmented Generation framework, enhancing generative models by incorporating external knowledge during text generation. This approach has been applied to retrieve the relevant specific and factual information from the data improving the performance of models on knowledge-intensive tasks.
- **Retrieval-augmented GPT-3.5-based Text-to-SQL Framework with Sample-aware Prompting and Dynamic Revision Chain** : Guo et al. (2023) explored the use of large language models for automatic SQL generation, demonstrating that context-aware query formulation significantly improves SQL accuracy. Their framework incorporates retrieval-augmented prompting and dynamic revision chains to enhance SQL generation performance.
- **Hybrid Semantic Search: Unveiling User Intent Beyond Keywords** : Ahluwalia et al. (2024) introduced a hybrid search approach that combines keyword matching, semantic vector embeddings, and LLM generated structured queries. This method aims to capture both explicit and implicit user intent, enhancing the relevance and context of search results.
- **Vector Similarity Search in Databases**: pgvector is an open-source PostgreSQL extension that adds vector similarity search capabilities to the database system. By allowing the storage and indexing of vector embeddings, pgvector facilitates efficient similarity searches within PostgreSQL, making it suitable for applications like semantic search and recommendation systems.
- **Language Models are Few-Shot Learners** : Brown et al. (2020) introduced an approach where accurate results are observed without any gradient updates or fine-tuning, with tasks and few-shot demonstrations specified purely via text interaction with the model. Where the model is prompted with a few examples to perform tasks without fine-tuning. This technique is instrumental in guiding

large language models to generate accurate SQL queries by providing relevant examples within the prompts.

- **Agentic AI: Autonomy, Accountability, and the Algorithmic Society**: Mukherjee and Chang (2025) discuss Agentic Artificial Intelligence (AI) systems capable of autonomously pursuing long-term goals, making decisions, and executing complex workflows. Unlike traditional generative AI, which reacts to prompts, agentic AI proactive

CHAPTER - 3

3. SOFTWARE REQUIREMENT SPECIFICATION

3.1 Introduction to SRS:

The Software Requirements Specification (SRS) is a detailed document that defines the functional and non-functional requirements for the SQRAG system. It serves as a blueprint for development, guiding the design, implementation, and testing phases.

The SRS is important for several reasons:

- **Clear Communication:** It provides a shared understanding of the system's purpose and features for all stakeholders, including developers, testers, and end-users.
- **Baseline for Testing**: The requirements outlined in the SRS can be used to create test cases, ensuring the final product meets the specified needs.
- **Scope Management**: It sets clear project boundaries, preventing unnecessary feature additions and ensuring development stays focused.

3.2 Role of SRS:

The Software Requirements Specification (SRS) plays a critical role in the SQRAG project by providing a clear and detailed outline of the application's requirements and functionalities. It serves as a foundational document that ensures all stakeholders—developers, project managers, and end-users—are aligned in their understanding of the project's goals.

By defining functional requirements, such as query retrieval, SQL generation, and result display, as well as non-functional requirements related to performance and usability, the SRS acts as a reference for design and implementation. Furthermore, it facilitates effective communication and sets the stage for testing and validation, ensuring that the final product meets user expectations and performs reliably. In essence, the SRS is vital for managing the project's scope and guiding its development lifecycle.

3.3 Requirement Specification Document:

This document outlines both functional and non-functional requirements that are critical for the SQL-RAG system. It provides detailed specifications regarding how the system interfaces with users, connects with the database infrastructure, and implements the LLM-based query generation pipeline. The requirements describe the expected behaviors, interactions, and technical specifications necessary for the system to properly function while meeting performance, security, and usability standards. The document serves as a comprehensive blueprint for understanding how SQL-RAG should operate across these three key interaction domains and what quality attributes the system must maintain.

3.4 Functional Requirements:

1. **SQL Query Generation**: The system transforms natural language user inputs into highly optimized SQL queries through a sophisticated retrieval-augmented generation approach. This process leverages contextual understanding to create database-appropriate queries that accurately reflect user intent. By analyzing the semantic structure of requests and mapping them to database schema elements, SQRAG ensures that generated queries are both syntactically correct and logically aligned with the user's information needs.
2. **Hybrid Search Mechanism**: SQRAG employs a comprehensive dual-search strategy that combines semantic understanding with traditional keyword-based techniques. This hybrid approach ensures more precise and relevant query retrieval by capturing both conceptual meaning and specific terminology. The system dynamically balances these complementary search methods to overcome the limitations of either approach used in isolation, resulting in superior recall and precision metrics across diverse query types.

3. **LLM-Driven Decision Making**: The system incorporates intelligent agent components powered by large language models to make critical operational decisions. These agents autonomously determine appropriate search strategies, manage query retry attempts when necessary, and orchestrate the generation of coherent responses based on retrieved information. By continuously evaluating query context and results quality, the agents implement sophisticated fallback mechanisms and optimization techniques that enhance system reliability and output quality.

4. **Database Interaction**: SQRAG features robust support for PostgreSQL database systems and seamlessly integrates with pgvector extension to enable advanced vector-based similarity search capabilities. This integration allows for efficient matching of complex query patterns with stored data representations. The system maintains optimized connection pools, implements intelligent caching strategies, and provides graceful error handling to ensure reliable database operations even under varying load conditions or when dealing with complex query execution plans.

5. **Data Preprocessing**: Before operational use, the system performs thorough data preparation including cleaning procedures, embedding generation, and strategic indexing of database textual content. These preprocessing steps establish the foundation for efficient search operations and accurate information retrieval. SQRAG applies customizable normalization techniques, generates high-dimensional.

3.5 Non-Functional Requirements:

- **Scalability**: The system should handle large databases efficiently, maintaining responsive performance even when processing multi-gigabyte datasets with complex schema structures. SQRAG implements optimization techniques and caching mechanisms to ensure consistent performance as data volumes and user numbers increase.

- **Usability**: The UI should be intuitive and require minimal SQL expertise, enabling non-technical users to leverage database querying through natural language interactions. The interface presents a clean design that makes common operations simple while keeping specialized capabilities accessible when needed.

- **Reliability**: Ensure accurate query generation and result retrieval through validation mechanisms and intelligent error handling. The system implements quality assurance layers including SQL verification and result evaluation, with logging capabilities for continuous improvement.

- **Security**: Protect database access and restrict unauthorized query execution by implementing authentication, authorization, and audit mechanisms. The system enforces access controls that respect existing database security policies and prevents potentially harmful operations.

- **Maintainability**: Allow for easy updates to the retrieval and query generation logic through a modular architecture with clear separation of concerns. Components can be independently modified or replaced without affecting the entire system, supporting ongoing enhancements and adaptations to new requirements.

3.6 Performance Requirements:

- **Query Execution Time**: The system should return results within an acceptable time frame (e.g., under 2 seconds for standard queries).
- **Embedding Storage Efficiency**: Optimize vector storage in pgvector to minimize database load.
- **Search Optimization**: Implement ranking mechanisms to prioritize relevant results.

3.7 Software Requirements:

Programming Languages

- Python (for backend, query processing, and LLM interaction)
- SQL (for database management and query execution)

Backend Requirements

- FastAPI (for REST API development and request handling)
- NLP and Text Generation
- LangChain & LangGraph (for LLM-based decision-making and query generation)
- OpenAI (for generating queries, taking decisions and generating final response)
- OpenAI Embedder (for text embeddings)

Frontend Technologies

- Streamlit & HTML (for building the user interface)

3.8 Hardware Requirements:

- Operating System: Windows, macOS, or Linux
- Processor: Intel Core i5 (or equivalent AMD Ryzen)
- RAM: 8GB DDR4
- Storage: 256GB SSD (for faster database and embedding operations)
- Internet Connection: Minimum 5 Mbps (for efficient API and model interactions)
- Browser: Chrome, Firefox, or Edge (for Streamlit frontend)
- Display: HD Display (for optimal UI experience)

CHAPTER - 4

4. SYSTEM DESIGN

4.1 Introduction to UML:

The Unified Modeling Language (UML) is an essential component of software engineering, providing a standardized way to visualize the design of a system. In the context of Software Requirements Specification (SRS) documentation, UML serves as a powerful tool to capture and communicate the system's requirements and structure effectively. It facilitates better understanding among stakeholders, including developers, project managers, business analysts, and clients.

UML is particularly beneficial in an SRS because it offers various diagram types that can represent different aspects of the software system. These diagrams help in illustrating both the static and dynamic characteristics of the system, making the requirements more accessible and understandable. The use of visual representations allows for easier identification of relationships between components, system behavior, and overall architecture.

In an SRS, UML diagrams can be categorized into two main groups: structural diagrams and behavioral diagrams.

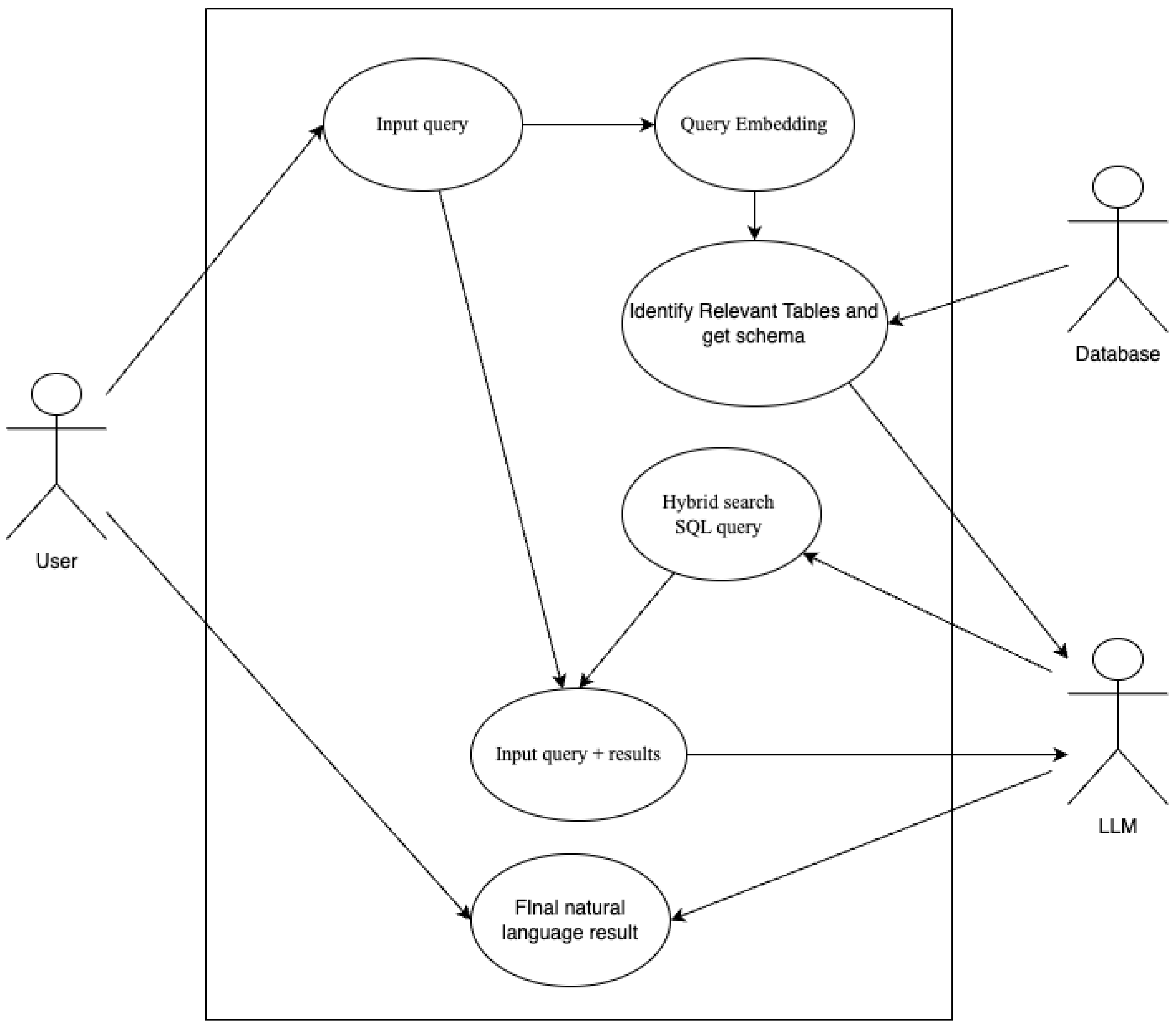
1. **Structural Diagrams**: These diagrams depict the organization and relationships among various components of the system. Key structural diagrams include:
    - **Class Diagrams**: Illustrate the system's classes, their attributes, methods, and the relationships between them, helping to define the system's static structure.
    - **Component Diagrams**: Show how components are wired together to form larger systems, focusing on the organization and dependencies of various modules.
    - **Deployment Diagrams**: Represent the physical deployment of artifacts on

nodes, helping stakeholders understand how the software will be distributed in a runtime environment.

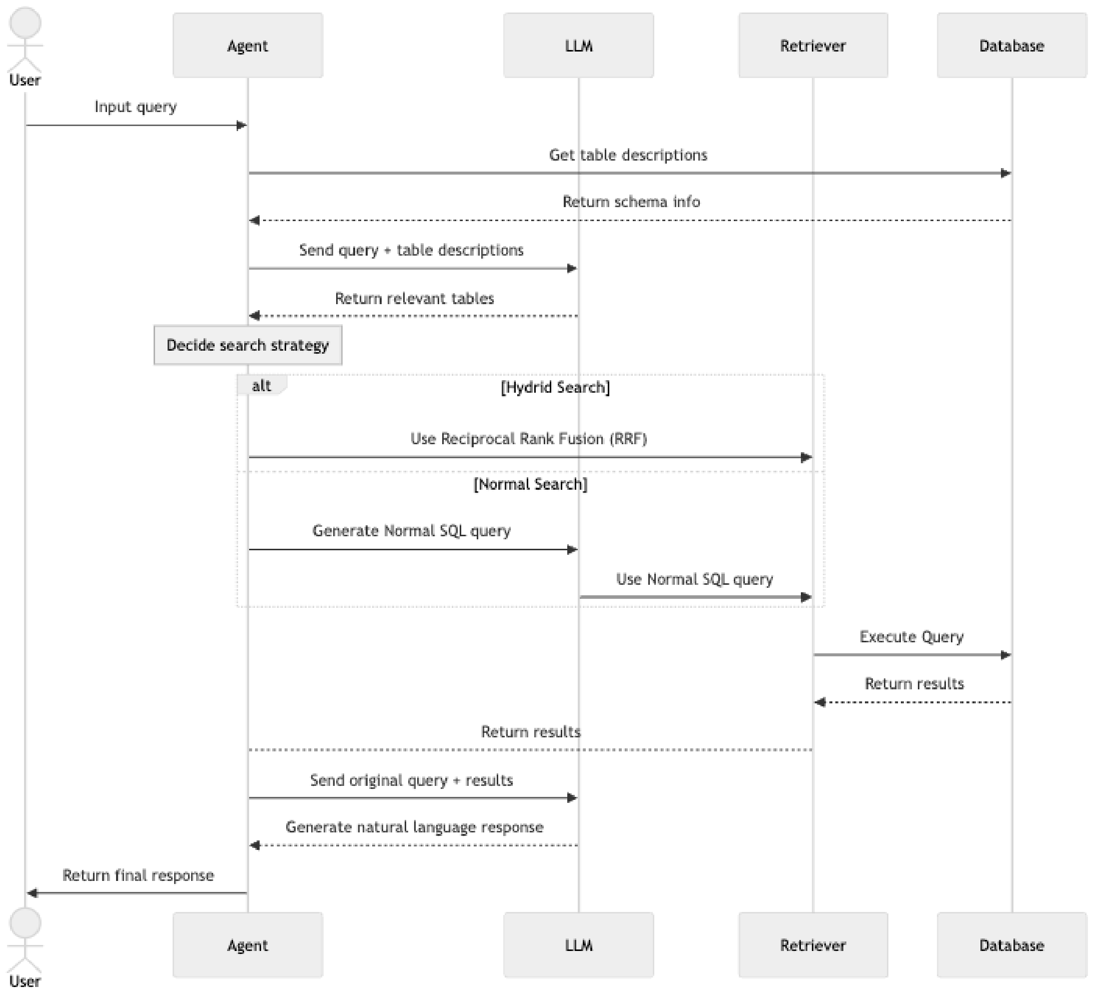
2. **Behavioural Diagrams**: These diagrams illustrate the interactions between the system and external actors, highlighting how the system behaves under different scenarios. Key behavioural diagrams include:
   - **Use Case Diagrams**: Identify the functional requirements of the system by showing the interactions between users (actors) and the system. Each use case represents a specific functionality that the system must provide.
   - **Sequence Diagrams**: Detail how objects interact in a particular sequence, outlining the flow of messages between components over time. This is crucial for understanding how various parts of the system communicate and operate.
   - **Activity Diagrams**: Depict workflows of stepwise activities and actions, showing the flow from one activity to another, which is useful for modelling the overall process of system operations.

4.2 UML Diagrams

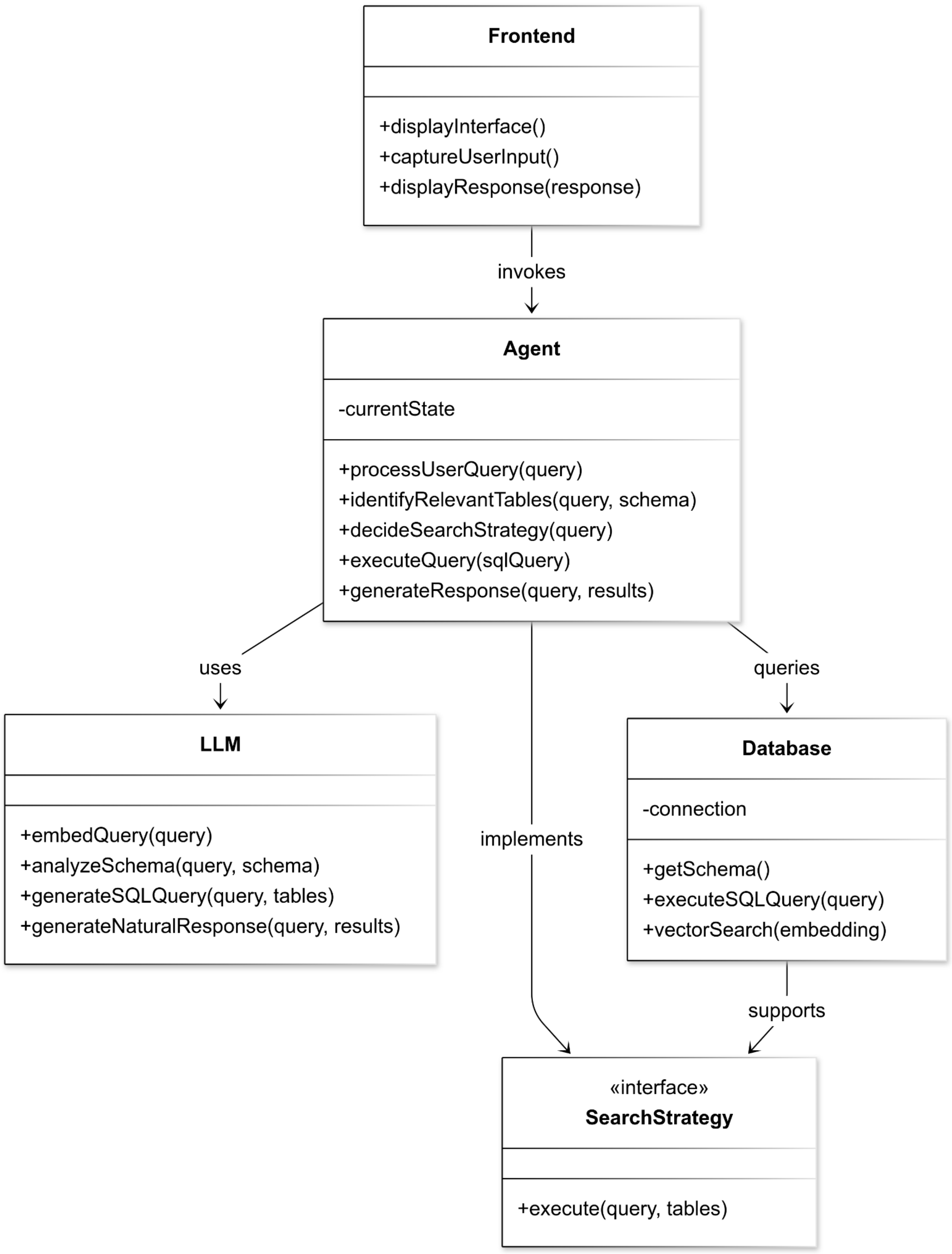
4.2.1 Use Case Diagram



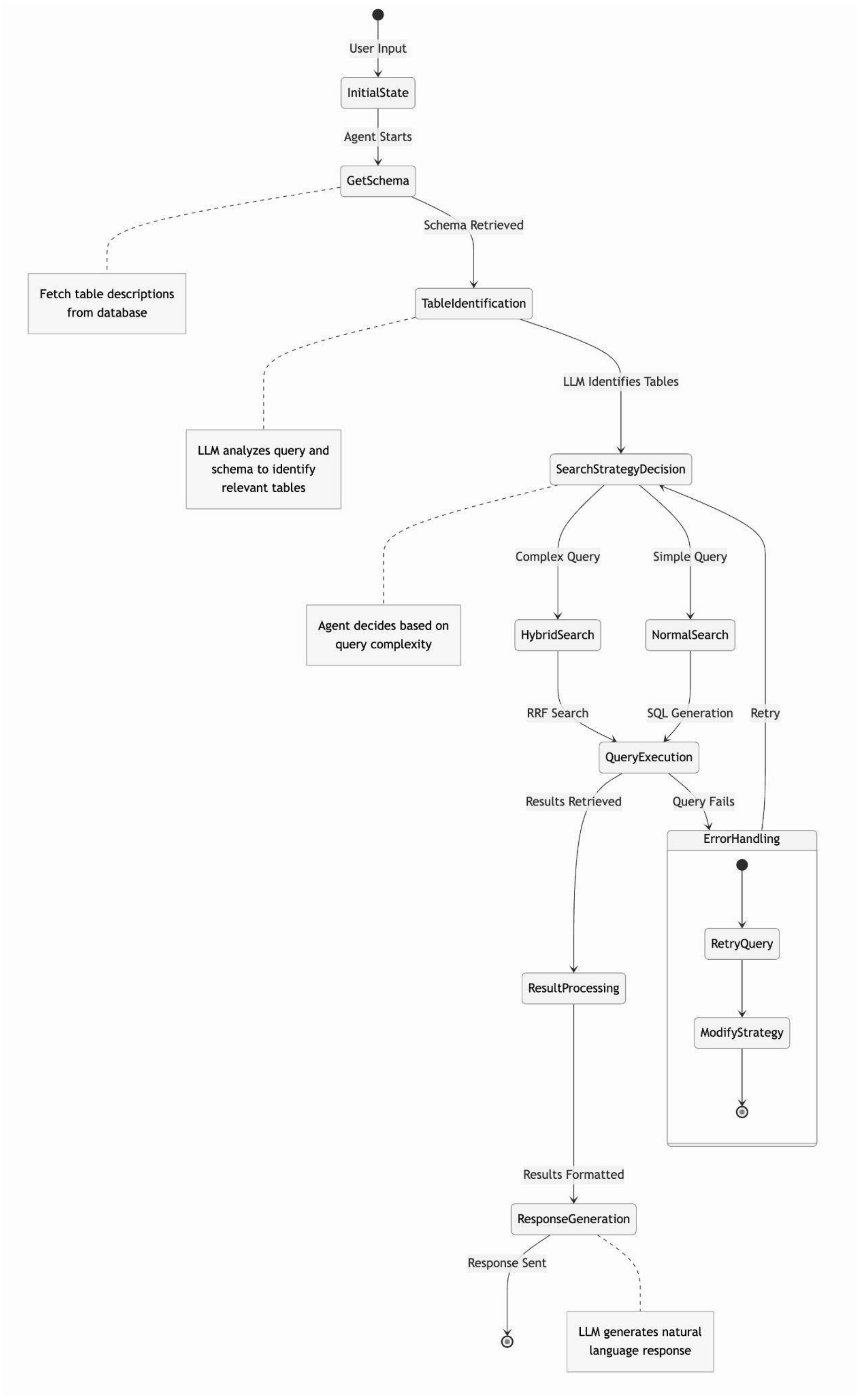
4.2.2 Sequence Diagram



4.2.3 Class Diagram



4.2.3 Activity Diagram



4.3 Technologies Used:

This section outlines the key technologies utilized across different components of the system, categorized by their respective roles in the architecture.

1. Frontend Development

Streamlit

- Used for rapidly developing the web application interface with minimal frontend code
- Provides built-in components for displaying chat messages, status indicators, and input fields
- Handles state management through `st.session_state` for preserving conversation history
- Supports asynchronous operations with `asyncio` integration

2. Backend Development

Python Ecosystem

- Core language used throughout the backend implementation
- Leverages async/await patterns for efficient handling of concurrent operations
- Used for data processing, model integration, and API development

FastAPI

- Modern, high-performance web framework for building APIs with Python
- Automatic data validation with Pydantic models
- Built-in support for asynchronous request handling

- OpenAPI documentation generation

Database Technologies

PostgreSQL

- Primary relational database for storing structured data
- Vector extension for efficient similarity searches
- Custom vector operations to support embeddings-based retrieval
- Connection management via `psycopg2` library

Data Processing Libraries

- **Pandas**: Used for data manipulation, cleaning, and transformation
  - Handles CSV parsing and dataframe operations
  - Provides numerical data processing capabilities
- **NumPy**: Supports mathematical operations on array data

3. Machine Learning & AI Components

Large Language Models (LLMs)

- **OpenAI Models**:
  - GPT-4o and GPT-4o-mini for primary text generation
  - Support for both direct OpenAI API and Azure-hosted deployments
- **Anthropic Claude Models**:
  - Claude 3 Haiku and Claude 3.5 Sonnet for alternative text generation
  - Integrated through the Anthropic API
- **Google Models**:
  - Gemini 2.0 Flash and Gemini 2.5 Flash Preview for additional text generation

options

LangChain Framework

- Core orchestration for building LLM applications
- Provides abstractions for working with different model providers
- Chat message handling and structured output parsing
- Runnable chains for complex prompt engineering

Embedding Models

- **OpenAI Embeddings**: Both direct and Azure-hosted embedding models
    - text-embedding-3-small for vector representation generation
- **Sentence Transformers**: Alternative embedding approach using local models
    - Support for various model architectures
- **Google Embeddings**: Gemini embedding capabilities for semantic similarity

RAG (Retrieval-Augmented Generation)

- Vector-based similarity search for finding relevant database records
- Custom retriever component (PGRetriever) for database interaction
- Context augmentation for grounding LLM responses in factual data

4. Evaluation & Testing

RAGAS Framework

- Comprehensive evaluation of RAG systems
- Metrics for measuring:
    - Answer correctness
    - Context precision and Context recall

- Faithfulness to source data

Synthetic Data Generation

- Test set generation for systematic evaluation
- Query distribution modeling for diverse test cases
- Document processing for creating evaluation datasets

5. Development & Deployment Tools

Environment Management

- Environment variable configuration for API keys and connection strings
- Support for different execution modes (development vs. production)

Client Libraries

- Custom client implementation for interacting with backend services
- Streaming support for real-time message updates
- Thread management for conversation context

Data Cleaning and Processing

- Custom cleaning functions for structured data (CSV files)
- Type conversion and validation for database compatibility
- Support for handling various data formats

6. Integration Technologies

API Integration

- Support for multiple LLM providers through unified interfaces

- Authentication handling for secure API access
- Rate limiting and error handling for robust operation

Database Integration

- Connection pooling for efficient database access
- Vector operations for similarity search
- SQL query generation and execution

7. Security Considerations

API Key Management

- Secure handling of API keys through environment variables
- Support for different authentication methods across providers

Data Validation

- Input validation using Pydantic models
- Type checking and conversion for preventing injection attacks

This comprehensive technology stack enables SQ-RAG to provide natural language querying capabilities for database data, with a focus on accurate retrieval, contextual understanding, and high-quality responses.

.

CHAPTER - 5

5. IMPLEMENTATION

5.1 Steps for Setting up Environment

1. Environment Initialization and Project Scaffolding

Install Python and Dependencies

- Install Python 3.9+ as the core runtime environment
- Set up a virtual environment using `venv` or `conda` for dependency isolation
- Install core packages via pip using the command **pip install -r requirements.txt**

Database Configuration

- Set up PostgreSQL database with vector extension for similarity search capabilities
- Use pgvector github repo for instructions to install vector extension
- Configure database connection parameters in environment variables
- Create schema for storing both structured data and a description table

API Key Management

- Configure environment variables for multiple LLM providers:
    - OpenAI API key
    - Anthropic API key
    - Google API key
    - Azure API endpoints and key
- Implement fallback mechanisms for when specific providers are unavailable

2. Frontend Architecture with Streamlit

UI Component Design

- Utilize Streamlit's built-in components for rapid interface development
- Implement chat message containers for conversation history
- Create sidebar with model selection options
- Add status indicators for tool calls and processing operations

Interaction Pipeline

- Configure user input via `st.chat_input()`
- Implement asynchronous message streaming for real-time responses
- Set up session state management for conversation persistence
- Create message drawing function with support for different message types (user, AI, tool)

3. Backend Development

Data Model Setup

- Define Pydantic models for structured data representation
- Implement message models for conversation handling
- Create tool call models for structured interactions with database

Client Implementation

- Create asynchronous client for communication with backend API
- Implement streaming message handling with asyncio
- Support for multiple model providers through unified interface

Database Connection Layer

- Implement PostgreSQL connection wrapper with connection pooling
- Create utility functions for table operations
- Add error handling and retry mechanisms

4. Database Processing and Embedding

Data Cleaning and Preprocessing

- Create specialized functions for cleaning different data types
- Implement numeric conversion and format standardization
- Handle missing values and edge cases

Embedding Model Integration

- Implement multiple embedding providers (OpenAI, Azure, SentenceTransformers, Google Gemini)
- Create base embedding class with standardized interface
- Add batch processing for efficient embedding generation

Vector Database Storage

- Create SQL schema for storing vector embeddings
- Implement vector column addition and population
- Add table description generation for improved context

5.2 Retrieval and Generation Integration

1. Retrieval Mechanism

PG Retriever Implementation

- Create specialized retriever for PostgreSQL vector queries

- Implement semantic similarity search using vector operations
- Support parameterized queries for flexible retrieval

Context Processing

- Convert retrieved database rows to markdown for LLM consumption
- Format table data with proper headers and formatting
- Filter irrelevant or sensitive columns

2. LLM Integration

Model Provider Configuration

- Set up multiple LLM providers for redundancy and comparison:
    - OpenAI (direct and Azure-hosted)
    - Anthropic Claude models
    - Google Gemini models
- Implement consistent interface across providers

Streaming Response Handling

- Implement asynchronous streaming for real-time responses
- Support token-by-token display in Streamlit interface
- Track and display tool call status during execution

3. Tool Integration

Tool Call System

- Create tool call mechanism for structured database interactions
- Implement tool response handling and formatting
- Support multiple tools with different capabilities

Custom Event Handling

- Implement custom event system for specialized interactions
- Support structured data payloads

5.3 Evaluation System

1. Synthetic Dataset Generation

Test Set Creation

- Implement systematic test set generation for evaluation
- Use high-quality models to create reference answers
- Generate diverse query types for comprehensive testing

2. Evaluation Metrics

RAGAS Integration

- Implement comprehensive evaluation using RAGAS framework
- Measure multiple quality dimensions:
  - Faithfulness to source data
  - Answer correctness
  - Context precision and recall
- Store evaluation results for tracking improvement

3. Model Comparison

LLM Performance Analysis

- Compare different LLM providers on standardized tasks
- Track cost and latency metrics

- Generate performance reports for informed model selection

5.4 Security and Efficiency Considerations

1. Privacy and Security

API Key Protection

- Implement secure handling of API keys via environment variables
- Avoid hard-coding sensitive credentials in source code
- Support rotation and management of credentials

Input Validation

- Sanitize all database inputs to prevent SQL injection
- Validate and clean user queries before processing
- Implement proper error handling for malformed inputs

2. Performance Optimization

Batch Processing

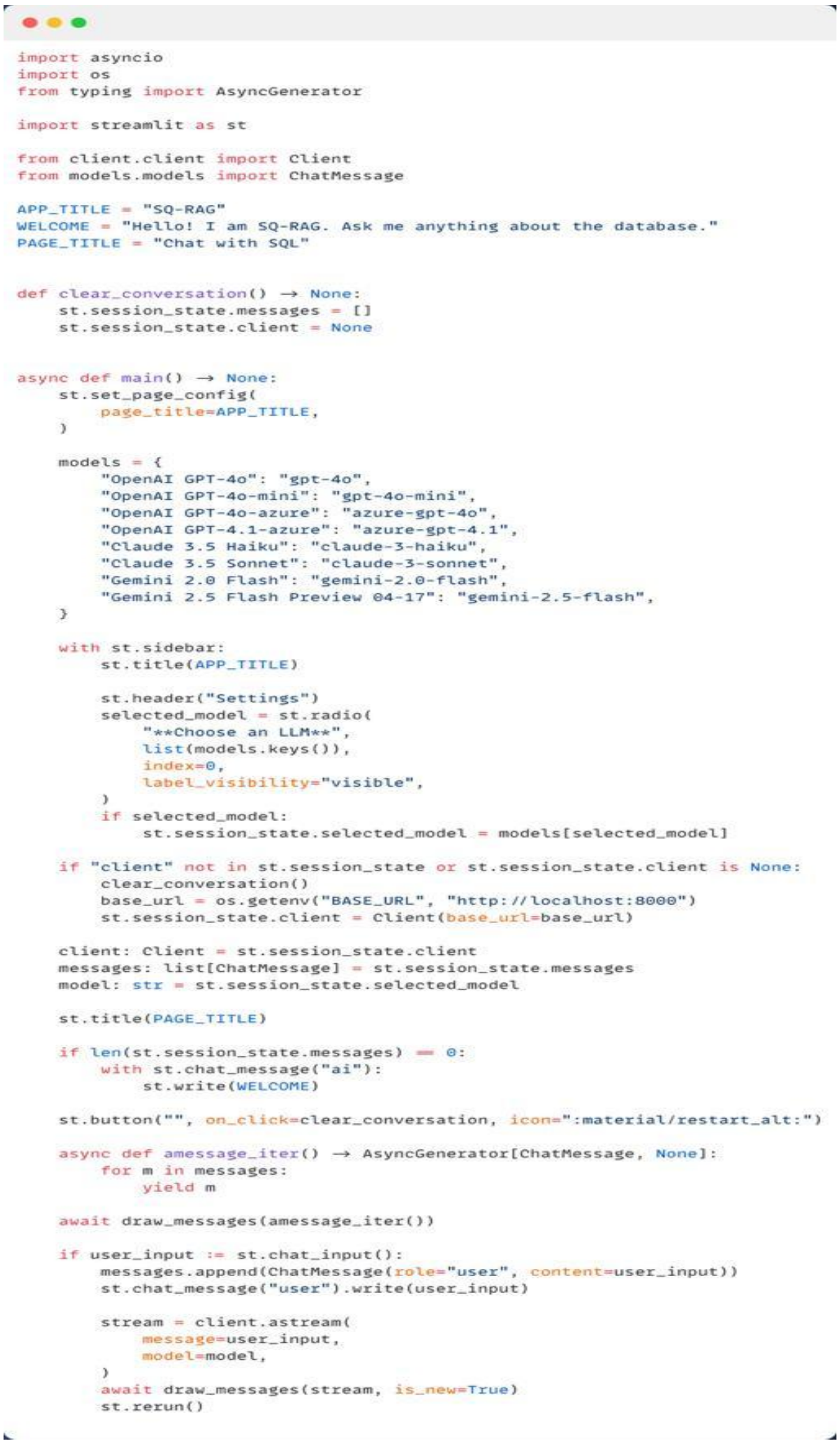
- Implement batched embedding generation for efficiency
- Use appropriate batch sizes based on model constraints
- Handle rate limiting and retry logic

Connection Pooling

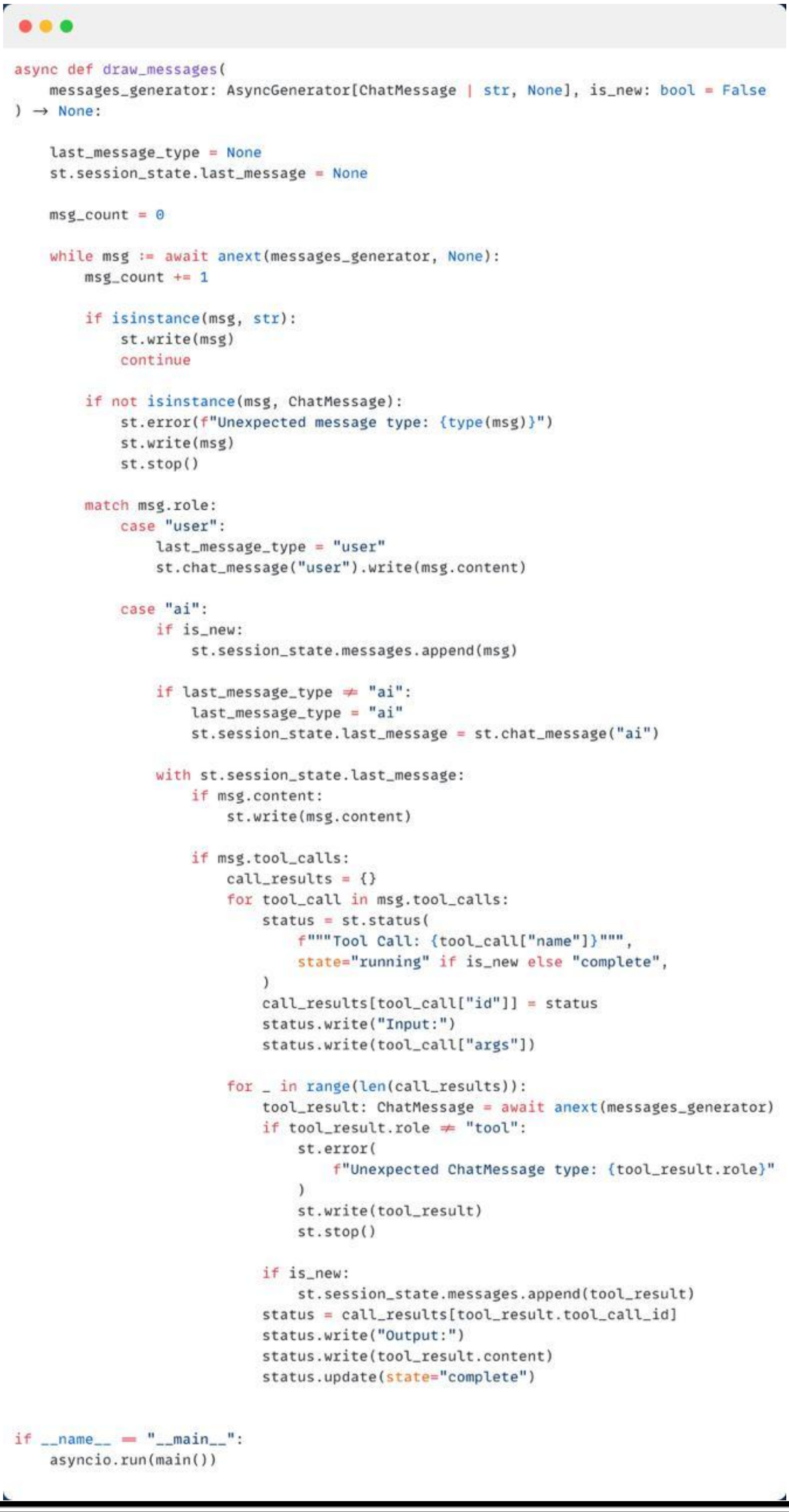
- Implement connection reuse pattern
- Add proper connection closure to prevent leaks

5.2 Coding the Logic

1. Frontend UI

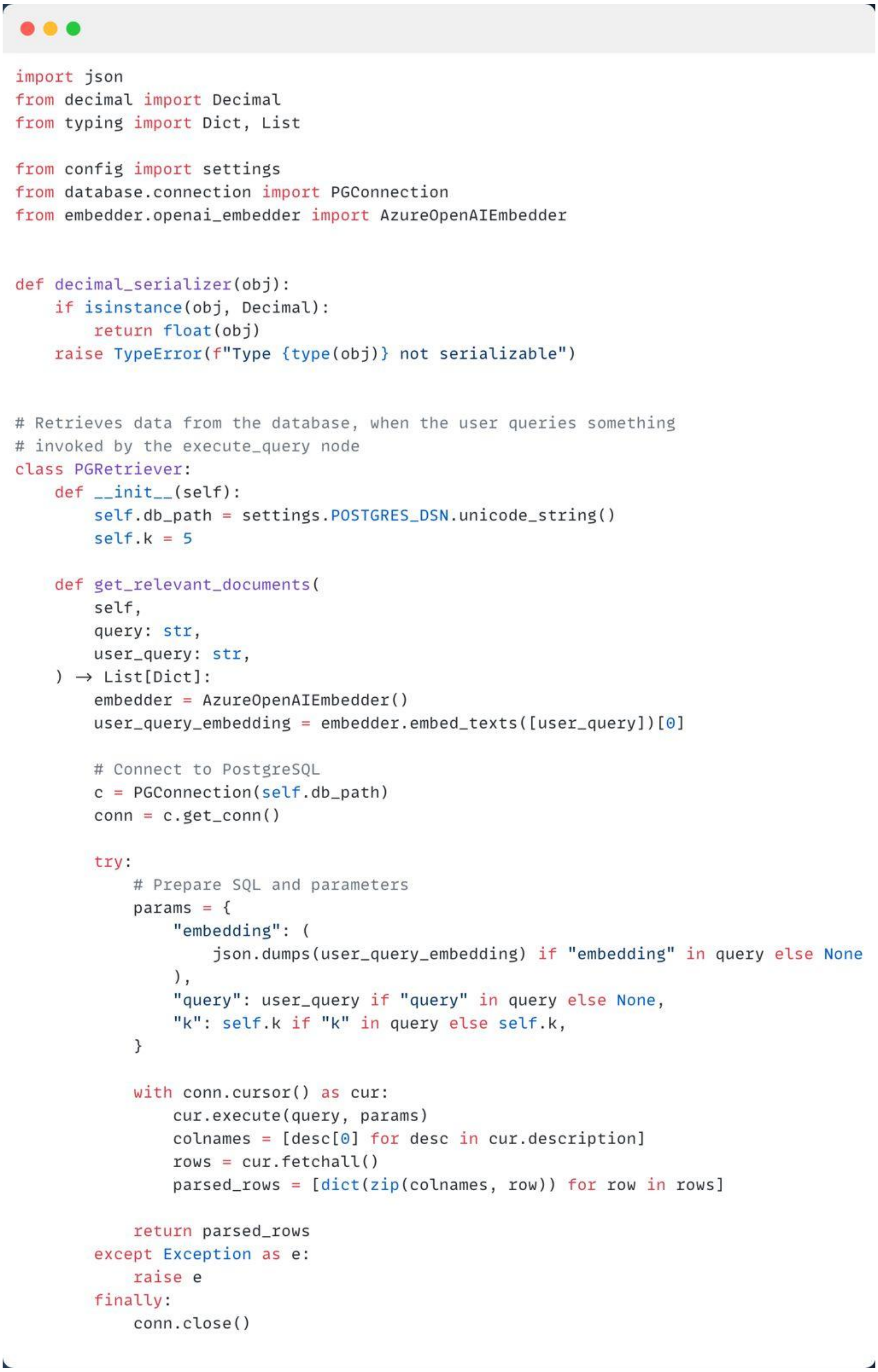


```python
import asyncio
import os
from typing import AsyncGenerator

import streamlit as st

from client.client import Client
from models.models import ChatMessage

APP_TITLE = "SQ-RAG"
WELCOME = "Hello! I am SQ-RAG. Ask me anything about the database."
PAGE_TITLE = "Chat with SQL"


def clear_conversation() -> None:
    st.session_state.messages = []
    st.session_state.client = None


async def main() -> None:
    st.set_page_config(
        page_title=APP_TITLE,
    )

    models = {
        "OpenAI GPT-4o": "gpt-4o",
        "OpenAI GPT-4o-mini": "gpt-4o-mini",
        "OpenAI GPT-4o-azure": "azure-gpt-4o",
        "OpenAI GPT-4.1-azure": "azure-gpt-4.1",
        "Claude 3.5 Haiku": "claude-3-haiku",
        "Claude 3.5 Sonnet": "claude-3-sonnet",
        "Gemini 2.0 Flash": "gemini-2.0-flash",
        "Gemini 2.5 Flash Preview 04-17": "gemini-2.5-flash",
    }

    with st.sidebar:
        st.title(APP_TITLE)

        st.header("Settings")
        selected_model = st.radio(
            "**Choose an LLM**",
            list(models.keys()),
            index=0,
            label_visibility="visible",
        )
        if selected_model:
            st.session_state.selected_model = models[selected_model]

    if "client" not in st.session_state or st.session_state.client is None:
        clear_conversation()
        base_url = os.getenv("BASE_URL", "http://localhost:8000")
        st.session_state.client = Client(base_url=base_url)

    client: Client = st.session_state.client
    messages: list[ChatMessage] = st.session_state.messages
    model: str = st.session_state.selected_model

    st.title(PAGE_TITLE)

    if len(st.session_state.messages) == 0:
        with st.chat_message("ai"):
            st.write(WELCOME)

    st.button("", on_click=clear_conversation, icon=":material/restart_alt:")

    async def amessage_iter() -> AsyncGenerator[ChatMessage, None]:
        for m in messages:
            yield m

    await draw_messages(amessage_iter())

    if user_input := st.chat_input():
        messages.append(ChatMessage(role="user", content=user_input))
        st.chat_message("user").write(user_input)

        stream = client.astream(
            message=user_input,
            model=model,
        )
        await draw_messages(stream, is_new=True)
        st.rerun()
```



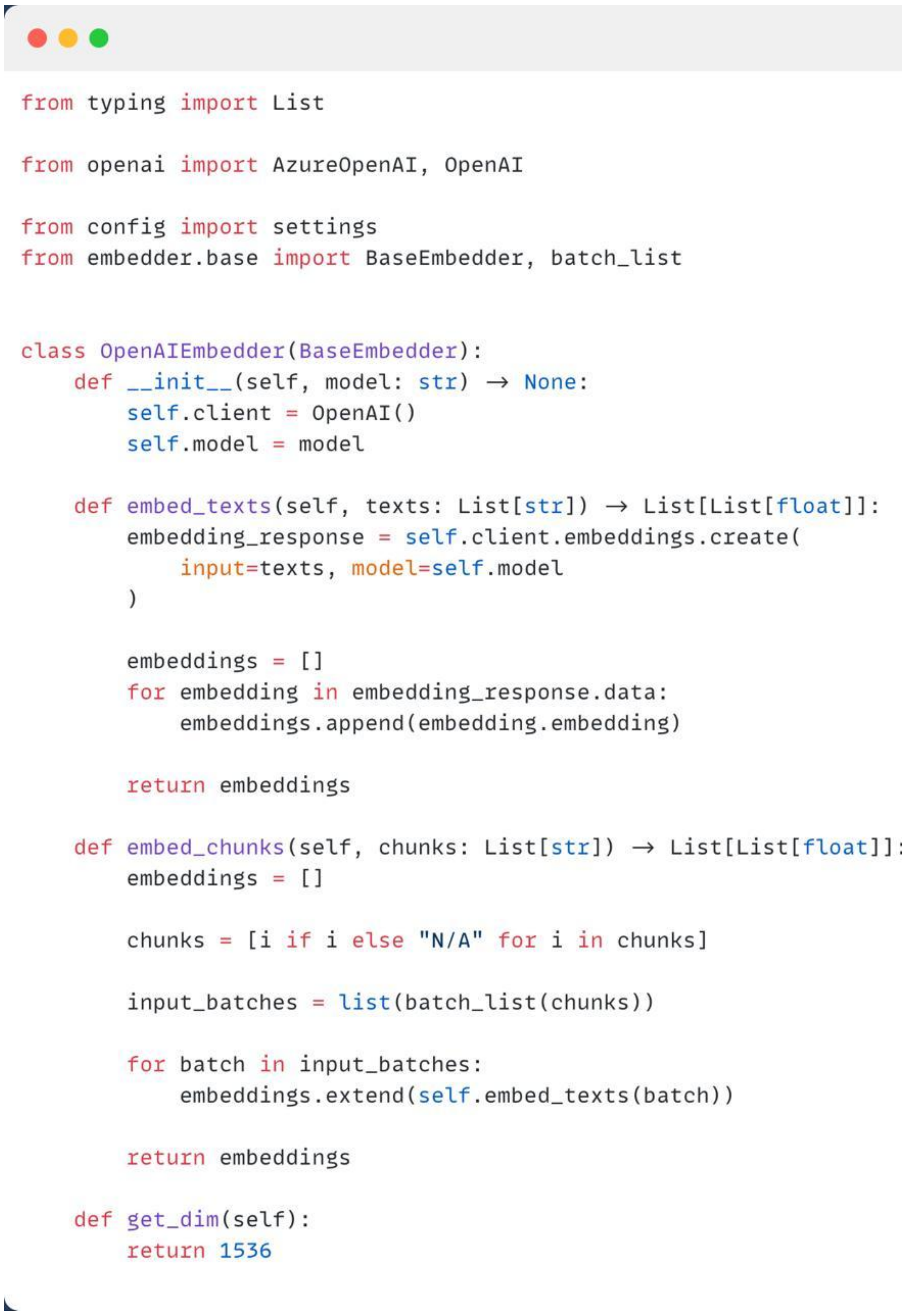
```python
async def draw_messages(
    messages_generator: AsyncGenerator[ChatMessage | str, None], is_new: bool = False
) -> None:

    last_message_type = None
    st.session_state.last_message = None

    msg_count = 0

    while msg := await anext(messages_generator, None):
        msg_count += 1

        if isinstance(msg, str):
            st.write(msg)
            continue

        if not isinstance(msg, ChatMessage):
            st.error(f"Unexpected message type: {type(msg)}")
            st.write(msg)
            st.stop()

        match msg.role:
            case "user":
                last_message_type = "user"
                st.chat_message("user").write(msg.content)

            case "ai":
                if is_new:
                    st.session_state.messages.append(msg)

                if last_message_type != "ai":
                    last_message_type = "ai"
                    st.session_state.last_message = st.chat_message("ai")

                with st.session_state.last_message:
                    if msg.content:
                        st.write(msg.content)

                    if msg.tool_calls:
                        call_results = {}
                        for tool_call in msg.tool_calls:
                            status = st.status(
                                f"""Tool Call: {tool_call["name"]}""",
                                state="running" if is_new else "complete",
                            )
                            call_results[tool_call["id"]] = status
                            status.write("Input:")
                            status.write(tool_call["args"])

                        for _ in range(len(call_results)):
                            tool_result: ChatMessage = await anext(messages_generator)
                            if tool_result.role != "tool":
                                st.error(
                                    f"Unexpected ChatMessage type: {tool_result.role}"
                                )
                                st.write(tool_result)
                                st.stop()

                            if is_new:
                                st.session_state.messages.append(tool_result)
                            status = call_results[tool_result.tool_call_id]
                            status.write("Output:")
                            status.write(tool_result.content)
                            status.update(state="complete")


if __name__ == "__main__":
    asyncio.run(main())
```

2. Retriever



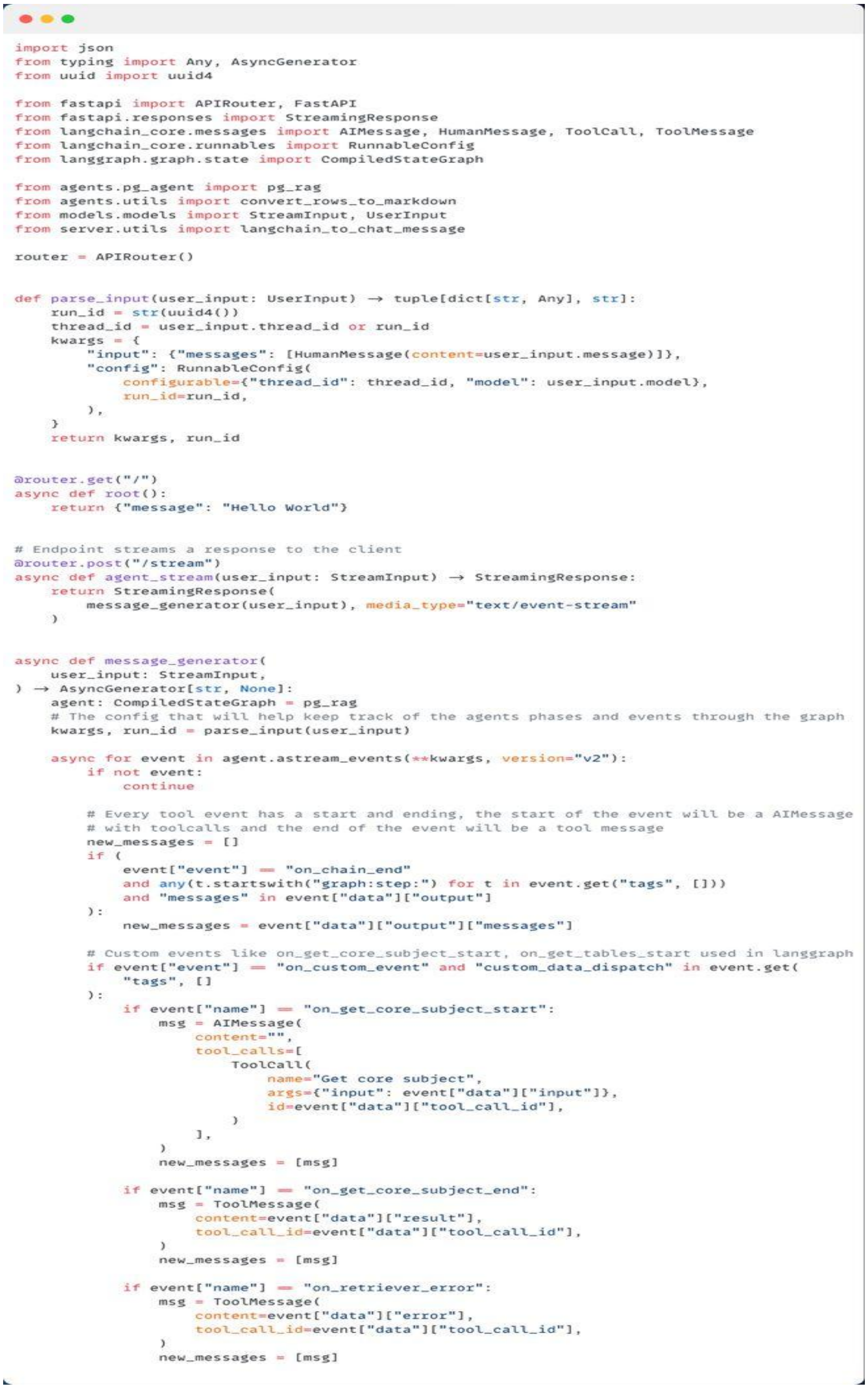
```python
import json
from decimal import Decimal
from typing import Dict, List

from config import settings
from database.connection import PGConnection
from embedder.openai_embedder import AzureOpenAIEmbedder


def decimal_serializer(obj):
    if isinstance(obj, Decimal):
        return float(obj)
    raise TypeError(f"Type {type(obj)} not serializable")


# Retrieves data from the database, when the user queries something
# invoked by the execute_query node
class PGRetriever:
    def __init__(self):
        self.db_path = settings.POSTGRES_DSN.unicode_string()
        self.k = 5

    def get_relevant_documents(
        self,
        query: str,
        user_query: str,
    ) -> List[Dict]:
        embedder = AzureOpenAIEmbedder()
        user_query_embedding = embedder.embed_texts([user_query])[0]

        # Connect to PostgreSQL
        c = PGConnection(self.db_path)
        conn = c.get_conn()

        try:
            # Prepare SQL and parameters
            params = {
                "embedding": (
                    json.dumps(user_query_embedding) if "embedding" in query else None
                ),
                "query": user_query if "query" in query else None,
                "k": self.k if "k" in query else self.k,
            }

            with conn.cursor() as cur:
                cur.execute(query, params)
                colnames = [desc[0] for desc in cur.description]
                rows = cur.fetchall()
                parsed_rows = [dict(zip(colnames, row)) for row in rows]

            return parsed_rows
        except Exception as e:
            raise e
        finally:
            conn.close()
```

3. Embedder

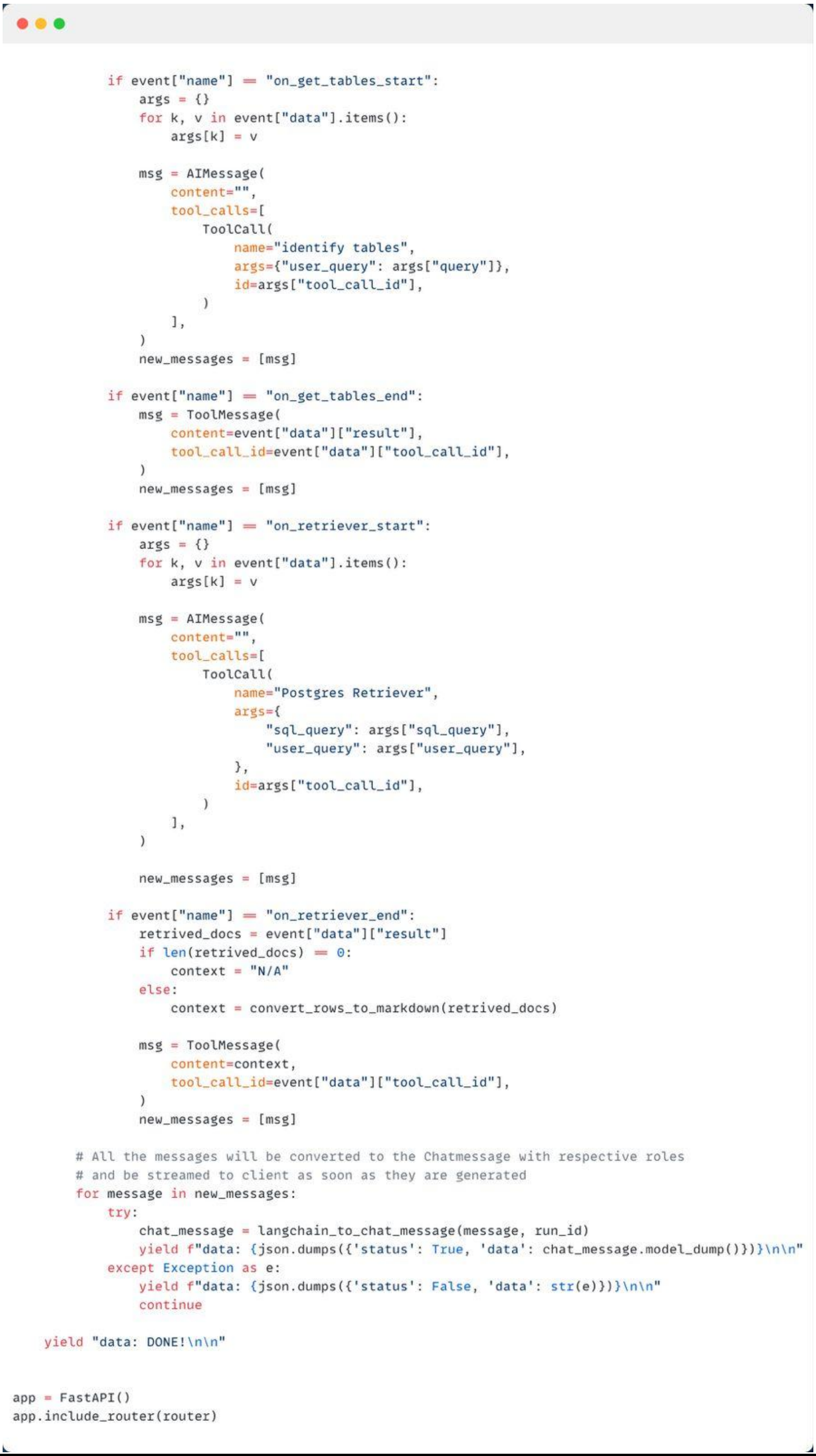


```python
from typing import List

from openai import AzureOpenAI, OpenAI

from config import settings
from embedder.base import BaseEmbedder, batch_list


class OpenAIEmbedder(BaseEmbedder):
    def __init__(self, model: str) → None:
        self.client = OpenAI()
        self.model = model

    def embed_texts(self, texts: List[str]) → List[List[float]]:
        embedding_response = self.client.embeddings.create(
            input=texts, model=self.model
        )

        embeddings = []
        for embedding in embedding_response.data:
            embeddings.append(embedding.embedding)

        return embeddings

    def embed_chunks(self, chunks: List[str]) → List[List[float]]:
        embeddings = []

        chunks = [i if i else "N/A" for i in chunks]

        input_batches = list(batch_list(chunks))

        for batch in input_batches:
            embeddings.extend(self.embed_texts(batch))

        return embeddings

    def get_dim(self):
        return 1536
```

4. Service



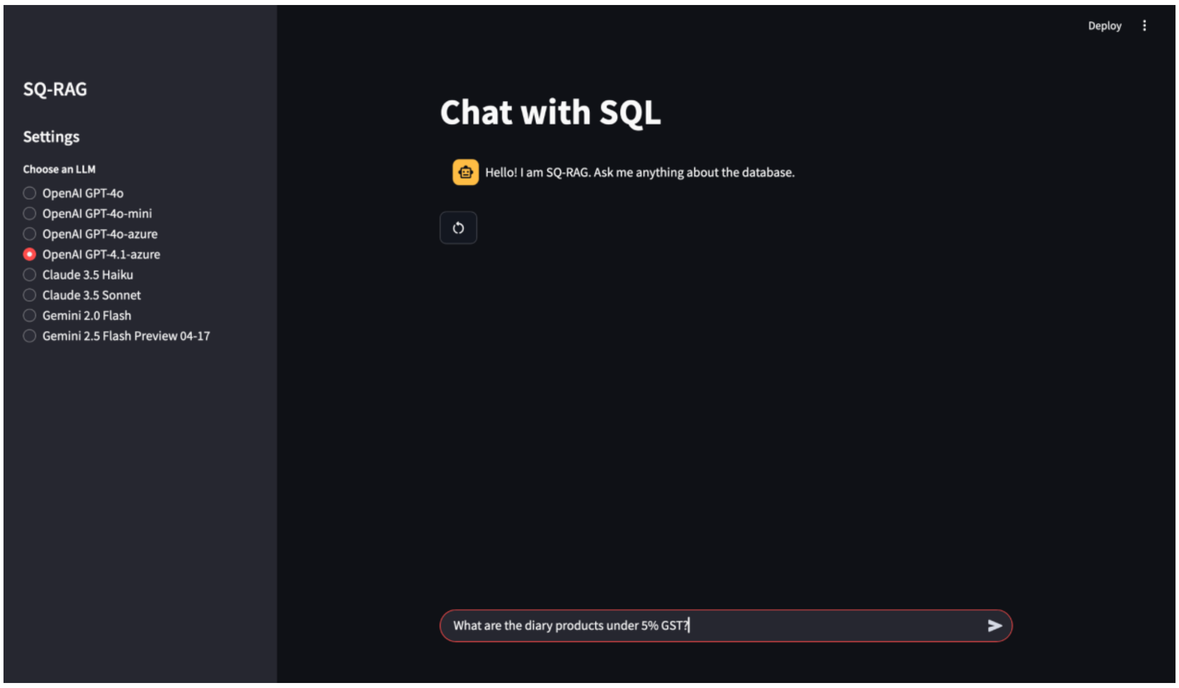
```python
import json
from typing import Any, AsyncGenerator
from uuid import uuid4

from fastapi import APIRouter, FastAPI
from fastapi.responses import StreamingResponse
from langchain_core.messages import AIMessage, HumanMessage, ToolCall, ToolMessage
from langchain_core.runnables import RunnableConfig
from langgraph.graph.state import CompiledStateGraph

from agents.pg_agent import pg_rag
from agents.utils import convert_rows_to_markdown
from models.models import StreamInput, UserInput
from server.utils import langchain_to_chat_message

router = APIRouter()


def parse_input(user_input: UserInput) → tuple[dict[str, Any], str]:
    run_id = str(uuid4())
    thread_id = user_input.thread_id or run_id
    kwargs = {
        "input": {"messages": [HumanMessage(content=user_input.message)]},
        "config": RunnableConfig(
            configurable={"thread_id": thread_id, "model": user_input.model},
            run_id=run_id,
        ),
    }
    return kwargs, run_id


@router.get("/")
async def root():
    return {"message": "Hello World"}


# Endpoint streams a response to the client
@router.post("/stream")
async def agent_stream(user_input: StreamInput) → StreamingResponse:
    return StreamingResponse(
        message_generator(user_input), media_type="text/event-stream"
    )


async def message_generator(
    user_input: StreamInput,
) → AsyncGenerator[str, None]:
    agent: CompiledStateGraph = pg_rag
    # The config that will help keep track of the agents phases and events through the graph
    kwargs, run_id = parse_input(user_input)

    async for event in agent.astream_events(**kwargs, version="v2"):
        if not event:
            continue

        # Every tool event has a start and ending, the start of the event will be a AIMessage
        # with toolcalls and the end of the event will be a tool message
        new_messages = []
        if (
            event["event"] == "on_chain_end"
            and any(t.startswith("graph:step:") for t in event.get("tags", []))
            and "messages" in event["data"]["output"]
        ):
            new_messages = event["data"]["output"]["messages"]

        # Custom events like on_get_core_subject_start, on_get_tables_start used in langgraph
        if event["event"] == "on_custom_event" and "custom_data_dispatch" in event.get(
            "tags", []
        ):
            if event["name"] == "on_get_core_subject_start":
                msg = AIMessage(
                    content="",
                    tool_calls=[
                        ToolCall(
                            name="Get core subject",
                            args={"input": event["data"]["input"]},
                            id=event["data"]["tool_call_id"],
                        )
                    ],
                )
                new_messages = [msg]

            if event["name"] == "on_get_core_subject_end":
                msg = ToolMessage(
                    content=event["data"]["result"],
                    tool_call_id=event["data"]["tool_call_id"],
                )
                new_messages = [msg]

            if event["name"] == "on_retriever_error":
                msg = ToolMessage(
                    content=event["data"]["error"],
                    tool_call_id=event["data"]["tool_call_id"],
                )
                new_messages = [msg]
```



```
            if event["name"] == "on_get_tables_start":
                args = {}
                for k, v in event["data"].items():
                    args[k] = v

                msg = AIMessage(
                    content="",
                    tool_calls=[
                        ToolCall(
                            name="identify tables",
                            args={"user_query": args["query"]},
                            id=args["tool_call_id"],
                        )
                    ],
                )
                new_messages = [msg]

            if event["name"] == "on_get_tables_end":
                msg = ToolMessage(
                    content=event["data"]["result"],
                    tool_call_id=event["data"]["tool_call_id"],
                )
                new_messages = [msg]

            if event["name"] == "on_retriever_start":
                args = {}
                for k, v in event["data"].items():
                    args[k] = v

                msg = AIMessage(
                    content="",
                    tool_calls=[
                        ToolCall(
                            name="Postgres Retriever",
                            args={
                                "sql_query": args["sql_query"],
                                "user_query": args["user_query"],
                            },
                            id=args["tool_call_id"],
                        )
                    ],
                )

                new_messages = [msg]

            if event["name"] == "on_retriever_end":
                retrived_docs = event["data"]["result"]
                if len(retrived_docs) == 0:
                    context = "N/A"
                else:
                    context = convert_rows_to_markdown(retrived_docs)

                msg = ToolMessage(
                    content=context,
                    tool_call_id=event["data"]["tool_call_id"],
                )
                new_messages = [msg]

        # All the messages will be converted to the Chatmessage with respective roles
        # and be streamed to client as soon as they are generated
        for message in new_messages:
            try:
                chat_message = langchain_to_chat_message(message, run_id)
                yield f"data: {json.dumps({'status': True, 'data': chat_message.model_dump()})}\n\n"
            except Exception as e:
                yield f"data: {json.dumps({'status': False, 'data': str(e)})}\n\n"
                continue

    yield "data: DONE!\n\n"


app = FastAPI()
app.include_router(router)
```

5.3 Screenshots

5.3.1 Frontend UI



SQ-RAG

Settings

Choose an LLM

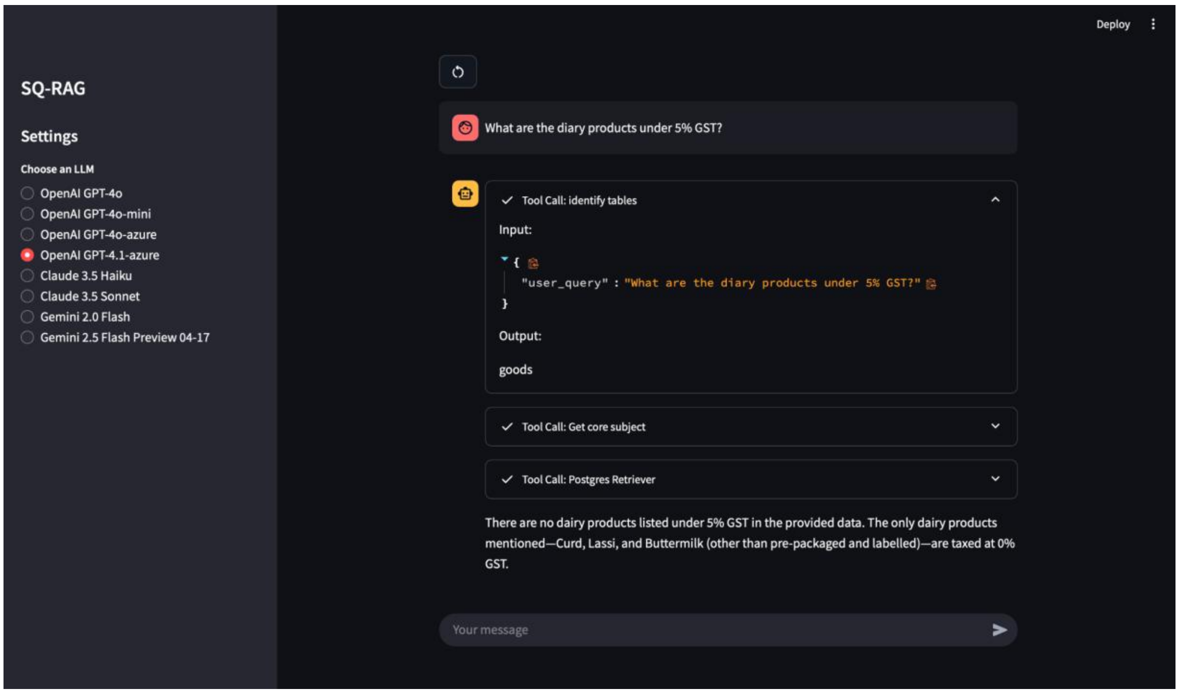
- OpenAI GPT-4o
- OpenAI GPT-4o-mini
- OpenAI GPT-4o-azure
- [✓ OpenAI GPT-4.1-azure]
- Claude 3.5 Haiku
- Claude 3.5 Sonnet
- Gemini 2.0 Flash
- Gemini 2.5 Flash Preview 04-17

Chat with SQL

Hello! I am SQ-RAG. Ask me anything about the database.

What are the diary products under 5% GST?

5.3.2 Tool Calls -



SQ-RAG

Settings

Choose an LLM

- OpenAI GPT-4o
- OpenAI GPT-4o-mini
- OpenAI GPT-4o-azure
- [✓ OpenAI GPT-4.1-azure]
- Claude 3.5 Haiku
- Claude 3.5 Sonnet
- Gemini 2.0 Flash
- Gemini 2.5 Flash Preview 04-17

What are the diary products under 5% GST?

✓ Tool Call: identify tables

Input:

```
{
  "user_query" : "What are the diary products under 5% GST?"
}
```

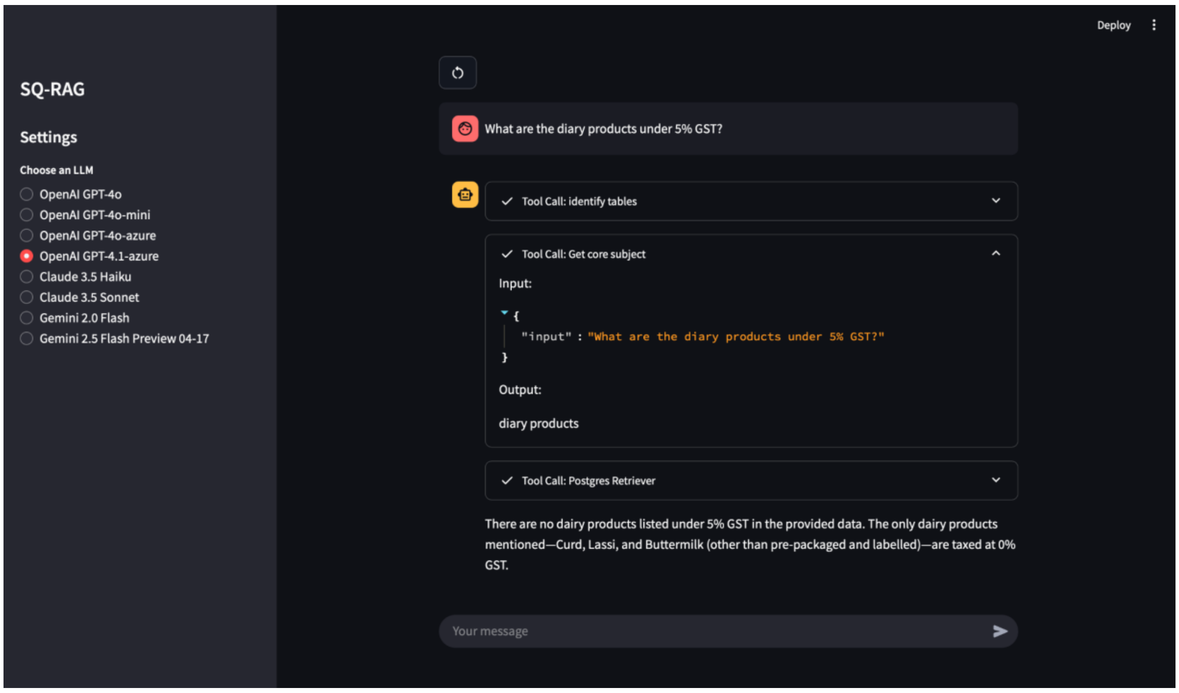
Output:

goods

✓ Tool Call: Get core subject

✓ Tool Call: Postgres Retriever

There are no dairy products listed under 5% GST in the provided data. The only dairy products mentioned—Curd, Lassi, and Buttermilk (other than pre-packaged and labelled)—are taxed at 0% GST.



Choose an LLM: OpenAI GPT-4.1-azure

What are the diary products under 5% GST?

✓ Tool Call: identify tables

✓ Tool Call: Get core subject

Input:

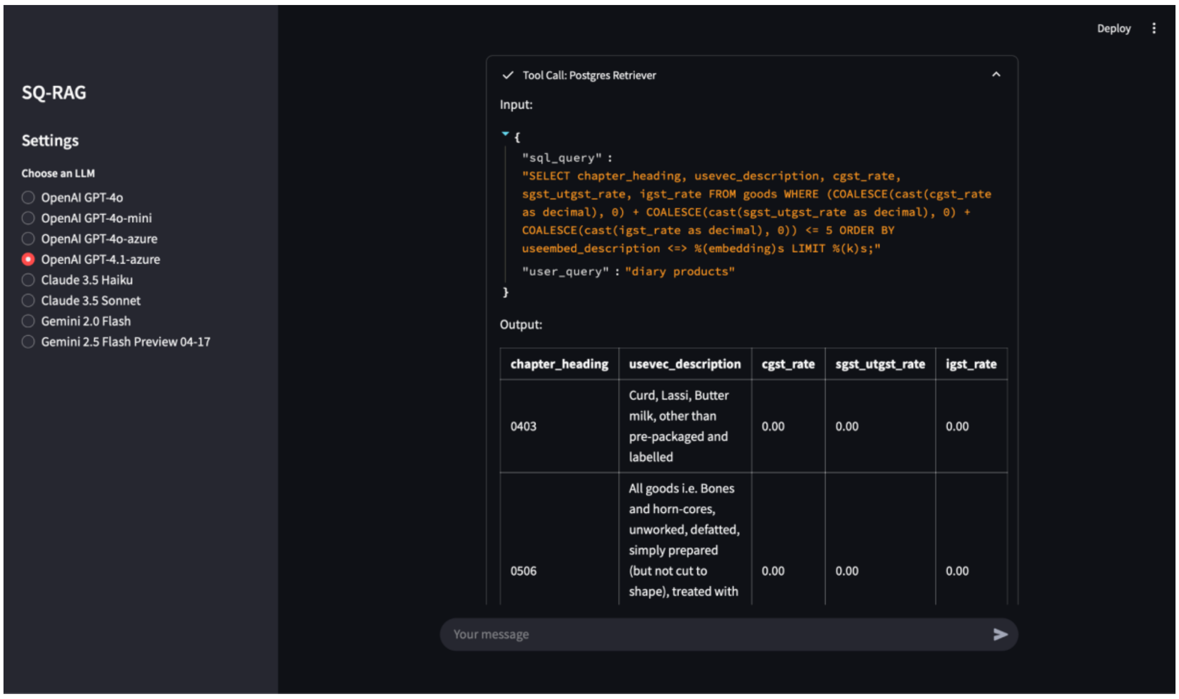
```
{
  "input" : "What are the diary products under 5% GST?"
}
```

Output:

diary products

✓ Tool Call: Postgres Retriever

There are no dairy products listed under 5% GST in the provided data. The only dairy products mentioned—Curd, Lassi, and Buttermilk (other than pre-packaged and labelled)—are taxed at 0% GST.



Choose an LLM: OpenAI GPT-4.1-azure

✓ Tool Call: Postgres Retriever

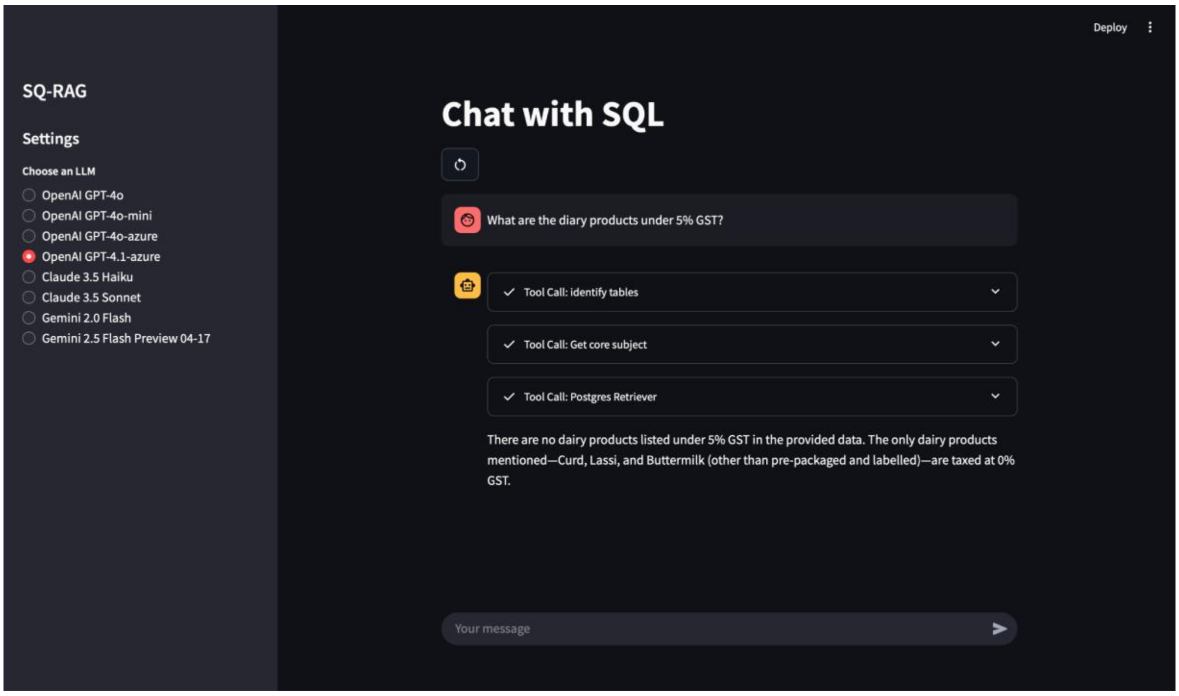
Input:

```
{
  "sql_query" :
  "SELECT chapter_heading, usevec_description, cgst_rate,
  sgst_utgst_rate, igst_rate FROM goods WHERE (COALESCE(cast(cgst_rate
  as decimal), 0) + COALESCE(cast(sgst_utgst_rate as decimal), 0) +
  COALESCE(cast(igst_rate as decimal), 0)) <= 5 ORDER BY
  useembed_description <=> %(embedding)s LIMIT %(k)s;"
  "user_query" : "diary products"
}
```

Output:

chapter_heading	usevec_description	cgst_rate	sgst_utgst_rate	igst_rate
0403	Curd, Lassi, Butter milk, other than pre-packaged and labelled	0.00	0.00	0.00
0506	All goods i.e. Bones and horn-cores, unworked, defatted, simply prepared (but not cut to shape), treated with	0.00	0.00	0.00

5.3.2 GENERATION OF QUERY WITH SEMANTIC SEARCH



Chat with SQL

Choose an LLM: ○ OpenAI GPT-4o ○ OpenAI GPT-4o-mini ○ OpenAI GPT-4o-azure [✓ OpenAI GPT-4.1-azure] ○ Claude 3.5 Haiku ○ Claude 3.5 Sonnet ○ Gemini 2.0 Flash ○ Gemini 2.5 Flash Preview 04-17

What are the diary products under 5% GST?

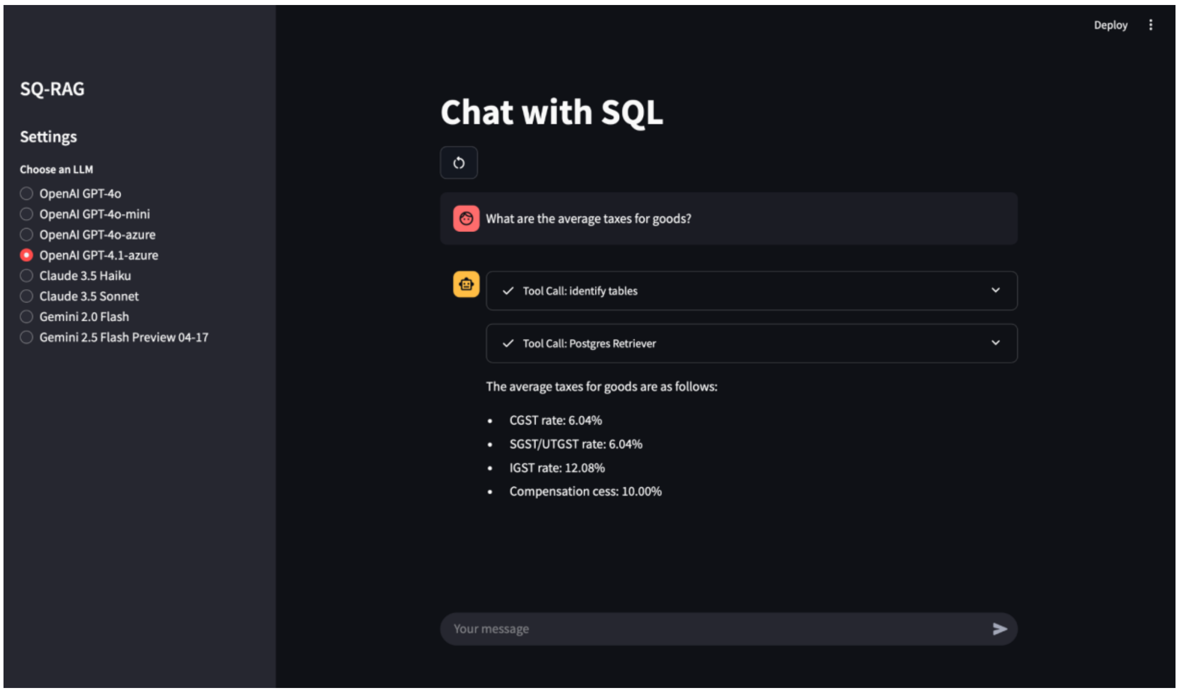
✓ Tool Call: identify tables

✓ Tool Call: Get core subject

✓ Tool Call: Postgres Retriever

There are no dairy products listed under 5% GST in the provided data. The only dairy products mentioned—Curd, Lassi, and Buttermilk (other than pre-packaged and labelled)—are taxed at 0% GST.

5.2.4 GENERATION OF STANDARD SQL QUERY



Chat with SQL

Choose an LLM: ○ OpenAI GPT-4o ○ OpenAI GPT-4o-mini ○ OpenAI GPT-4o-azure [✓ OpenAI GPT-4.1-azure] ○ Claude 3.5 Haiku ○ Claude 3.5 Sonnet ○ Gemini 2.0 Flash ○ Gemini 2.5 Flash Preview 04-17

What are the average taxes for goods?

✓ Tool Call: identify tables

✓ Tool Call: Postgres Retriever

The average taxes for goods are as follows:

- CGST rate: 6.04%
- SGST/UTGST rate: 6.04%
- IGST rate: 12.08%
- Compensation cess: 10.00%

CHAPTER - 6

6. SOFTWARE TESTING

6.1 Introduction

Software testing for the SQL RAG project is essential to ensure that the system reliably and accurately generates SQL queries from natural language inputs. The testing phase validates that users can input queries in plain language, receive correct and optimized SQL statements, and experience smooth interactions with the database. It covers query accuracy, semantic search performance, and the responsiveness of the overall system.

6.1.1 Testing Objectives

The primary objectives of testing the SQL RAG system are:

- To verify that the application accurately translates user natural language queries into valid and optimized SQL statements.
- To ensure that the hybrid search mechanism (semantic + keyword-based) retrieves the correct schema and query context.
- To confirm that the LangGraph agents handle query refinement, retries, and error management effectively.
- To validate that the system handles ambiguous or incomplete inputs gracefully, providing meaningful feedback to users.
- To ensure secure handling of user queries and that no sensitive data is stored or exposed during the interaction process.

6.1.2 Testing Strategies

The testing strategies for SQL RAG include:

- **Unit Testing:** Testing individual modules such as the semantic search, keyword retrieval, SQL generation, and embedding processes separately.

- **Integration Testing:** Verifying the seamless interaction between components—natural language processing, semantic search (pgvector), LangGraph agents, and database execution.
- **System Testing:** Performing full end-to-end testing covering input processing, query retrieval, SQL generation, and result display via the frontend.
- **Performance Testing:** Measuring the response time of the system to ensure fast query generation and minimal latency, especially under load.
- **User Acceptance Testing (UAT):** Ensuring the system meets user expectations regarding accuracy, usability, and performance in real-world scenarios.

6.1.3 System Evaluation

System evaluation assesses SQL RAG's ability to meet the requirements of non-technical and technical users who interact with databases. Key evaluation criteria include:

- **Functionality:** The system must generate accurate SQL queries that align with user intent and database schema.
- **Usability:** The interface should be intuitive, allowing users to input queries easily and interpret results without SQL expertise.
- **Performance:** The system should maintain low latency, providing query results within a predefined acceptable timeframe (e.g., under 2 seconds).
- **Reliability:** SQL RAG should operate consistently even under varying loads and handle errors gracefully.
- **Security:** The system must process queries securely, ensuring that database access is protected and unauthorized actions are prevented.

6.1.4 Testing New System

Testing the new SQL RAG system involves verifying that all core features function as expected in the live environment. This includes:

- **Functionality Testing:** Ensuring that natural language input is accurately processed and translated into valid SQL queries, and that results are retrieved successfully from the database.
- **API Integration Testing:** Verifying that the interaction between the semantic search module (pgvector), LangGraph agents, and PostgreSQL database operates smoothly and reliably.
- **Compatibility Testing:** Testing across different browsers and platforms (desktop and mobile) to ensure a seamless experience through the Streamlit interface.
- **Exception Handling:** Ensuring the system gracefully manages issues such as ambiguous queries, unsupported question types, or database connectivity errors, providing clear user feedback.

6.2 Test Cases

Test Case 1: Valid Natural Language Query

Description: Verify that the system accurately processes a valid natural language query and returns the correct SQL statement.

Expected Outcome: The system generates an optimized SQL query and retrieves the corresponding results from the database.

Test Case 2: Ambiguous or Incomplete Query

Description: Test system behavior when the user provides an incomplete or unclear natural language query.

Expected Outcome: The system prompts the user for clarification or provides a fallback message indicating the issue.

Test Case 3: SQL Generation and Execution Time

Description: Measure the time taken from natural language input to SQL execution and result display.

Expected Outcome: The system generates and executes the SQL query within the acceptable time frame (e.g., under 2 seconds).

Test Case 4: Semantic vs. Keyword Search Accuracy

Description: Test whether the system retrieves the correct schema information and generates the appropriate SQL query using both semantic and keyword-based search methods.

Expected Outcome: The system correctly identifies relevant tables/columns and constructs a valid query using the hybrid search approach.

Test Case 5: Error Handling for Invalid Schema Reference

Description: Simulate a user query that references a non-existent table or column.

Expected Outcome: The system returns an informative error message indicating that the table or column is not found.

Test Case 6: Network Failure During API Call

Description: Simulate a network interruption during the retrieval or SQL generation process.

Expected Outcome: The system notifies the user of the network issue and prompts a retry.

Test Case 7: Privacy Check for Query Logs

Description: Verify that user queries and generated SQL statements are processed in real-time without being stored in any logs or databases.

Expected Outcome: The system confirms real-time processing with no persistent storage of sensitive queries.

CONCLUSION

The SQL RAG project successfully integrates retrieval-augmented generation (RAG) with modern database systems to create an intelligent SQL query assistant. By leveraging a hybrid search pipeline that combines semantic and keyword-based retrieval, the system effectively translates natural language queries into optimized SQL statements. It bridges the gap between technical and non-technical users, enabling seamless interaction with relational databases without requiring prior SQL expertise.

The backend, built using FastAPI, integrates with PostgreSQL and the pgvector extension to manage vector embeddings for semantic search, while LangGraph-based agents dynamically handle query generation, refinement, and error correction. The frontend, developed with Streamlit, provides a clean and intuitive user interface for submitting queries and viewing results in real-time.

SQL RAG's architecture emphasizes modularity, allowing easy integration into existing database infrastructures. Its intelligent agent-driven workflow ensures that generated queries are context-aware, precise, and optimized for performance. This project demonstrates how combining large language models with retrieval techniques can streamline data access, improve decision-making processes, and democratize database usage across industries. With robust performance, accuracy, and user-friendliness, SQL RAG lays a strong foundation for further innovations in AI-driven database management and business intelligence solutions.

.

FUTURE ENHANCEMENTS

- **Multi-Database Engine Support:** Extending SQL RAG to support other popular databases such as MySQL, Oracle, and Microsoft SQL Server would broaden its usability. This enhancement ensures cross-platform compatibility, allowing organizations with diverse tech stacks to adopt the system seamlessly.
- **Advanced Query Optimization:** Integrating query execution planners and optimization modules that not only generate SQL but also refine and benchmark performance could significantly enhance result speed and reduce resource consumption.
- **Enterprise Analytics Integration:** Adding native integration with BI tools (like Tableau, Power BI) and analytics dashboards would empower users to visualize query results instantly. This will make SQL RAG not just a query generator but a full data exploration assistant.
- **Schema Auto-Discovery and Documentation:** Implementing a feature that automatically maps and documents the database schema can improve transparency and usability, especially for non-technical users. This module would keep schema snapshots updated and provide insights during query formulation.
- **Role-Based Access & Security Enhancements:** Incorporating detailed role-based access control (RBAC), along with OAuth2 and JWT integrations, will ensure SQL RAG meets enterprise-grade security requirements, allowing safe access to sensitive data.
- **Offline Embedding and Search:** Building an offline mode that caches schema, embeddings, and past queries can allow users to work in disconnected environments (e.g., during system outages or remote fieldwork), with synchronization capabilities once reconnected.
- **Multi-Language Natural Query Support:** Expanding the system to accept natural language queries in multiple languages (e.g., Hindi, Spanish, Mandarin) would make SQL

RAG accessible to global teams, increasing inclusivity and usability.

- **Enhanced Error Handling & Self-Healing Agents:** Developing self-healing LangGraph agents that can detect, explain, and auto-correct common query errors or schema mismatches will make the system more robust and user-friendly.
- **AI-Assisted Schema Design:** A future module could suggest schema improvements based on common query patterns, optimizing the database structure for better performance and easier querying.

REFERENCES

- **Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks**

  Authors: Patrick Lewis, Ethan Perez, Aleksandra Piktus, and others

  Summary: Introduces the RAG framework, which combines retrieval and generation for knowledge-intensive tasks—forming the backbone of SQL RAG's enhanced SQL generation.

  Link: https://arxiv.org/abs/2005.11401

- **Language Models are Few-Shot Learners (GPT-3)**

  Authors: Tom B. Brown, Benjamin Mann, Nick Ryder, and others

  Summary: Demonstrates how large language models like GPT-3 can generate accurate SQL queries from natural language input using a few-shot learning core to SQL RAG's query generation.

  Link: https://arxiv.org/abs/2005.14165

- **LangChain: Building LLM-Powered Applications**

  Authors: Harrison Chase

  Summary: Introduces LangChain, a framework for building modular AI pipelines, critical for implementing the agentic behavior and dynamic retrieval logic in SQL RAG.

  Link: https://python.langchain.com/docs/

- **Hybrid Semantic Search: Unveiling User Intent Beyond Keywords**

  Authors: Ahluwalia, S., Sharma, R., et al.

  Summary: Discusses combining semantic vector search with keyword-based techniques to improve relevance—an approach mirrored in SQL RAGs hybrid search pipeline.

  Link: https://arxiv.org/abs/2401.10177

- **Text-to-SQL Benchmarking: Spider Dataset**

  Authors: Tao Yu, Rui Zhang, and others

Summary: Presents the Spider dataset for evaluating complex and cross-domain text-to-SQL tasks, which helps benchmark SQL RAG's performance in SQL query generation.

Link: https://yale-lily.github.io/spider

- **SQLova: Table-Aware Neural Approaches to Text-to-SQL**

  Authors: Hyesung Kim, Minjoon Seo, and others

  Summary: Proposes a neural architecture that translates natural language into SQL, inspiring aspects of SQL RAG's pipeline for handling diverse query structures.

  Link: https://arxiv.org/abs/1902.01069

- **SQLova: Open-Source Vector Search for PostgreSQL**

  Authors: PGVector Community

  Summary: Describes pg vector, a PostgreSQL extension for vector similarity search, enabling SQL RAG to perform semantic retrieval of schema and query examples.

  Link: https://github.com/pgvector/pgvector

BIBILIOGRAPHY

- **Lewis, P., Perez, E., Piktus, A., et al. (2020).** Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *arXiv preprint arXiv:2005.11401.*

  This paper introduces the RAG framework, the foundation of SQL RAG's retrieval-augmented SQL generation.

  https://arxiv.org/abs/2005.11401
- **Brown, T.B., Mann, B., Ryder, N., et al. (2020).** Language Models are Few-Shot Learners. *arXiv preprint arXiv:2005.14165.*

  Demonstrates how large language models like GPT-3 can accurately generate SQL queries from natural language using few-shot learning techniques.

  https://arxiv.org/abs/2005.14165
- **Ahluwalia, S., Sharma, R., et al. (2024).** Hybrid Semantic Search: Unveiling User Intent Beyond Keywords. *arXiv preprint arXiv:2401.10177.*

  Explains hybrid search combining semantic vector embeddings and keyword matching—essential to SQL RAG's hybrid search engine.

  https://arxiv.org/abs/2401.10177
- **Chase, H. (2023).** LangChain: Building LLM-Powered Applications. *LangChain Documentation.*

  Provides a modular framework for building LLM pipelines and agentic logic, critical for SQL RAG's architecture.

  https://python.langchain.com/docs/
- **Yu, T., Zhang, R., Ernie, S., et al. (2018).** Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Text-to-SQL Task. *Proceedings of EMNLP.*

  Describes the Spider dataset used to benchmark SQL RAG's SQL generation capabilities.

  https://yale-lily.github.io/spider

- **Kim, H., Seo, M., Zhang, J., et al. (2019).** SQLova: Table-Aware Neural Approaches to Text-to-SQL Generation. *arXiv preprint arXiv:1902.01069.*

  Presents SQLova's architecture, which influences SQL RAG's text-to-SQL pipeline for accurate query generation.

  https://arxiv.org/abs/1902.01069
- **PGVector Contributors. (n.d.).** pg vector: Open-Source Vector Search for PostgreSQL. *GitHub Repository.*

  Describes pg vector, a PostgreSQL extension for vector similarity search, crucial for SQL RAG's semantic retrieval.

  https://github.com/pgvector/pgvector
- **OpenAI API Documentation. (n.d.).** OpenAI: Text Generation and Embedding APIs. *OpenAI Documentation.*

  Provides API documentation for text generation and embeddings, powering SQL RAG's query generation and retrieval.

  https://platform.openai.com/docs/