

|  |
| --- |
| Sutra Management Consultancies  SQL Agent  Approach Document |

**DOCUMENT VERSION 1**

**13/06/2025**

**AUTHORS**

|  |  |
| --- | --- |
| **Name** | **Role** |
| Dishant Savaliya | Analyst Intern |

**DOCUMENT HISTORY**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Document Revision Description** | **Document Author** |
| 13/06/2025 | 1 | SQL Agent | Dishant Savaliya |

**APPROVALS**

|  |  |  |  |
| --- | --- | --- | --- |
| **Approval Date** | **Approved Version** | **Approver Role** | **Approver** |
|  |  |  |  |

Contents

[Executive Summary 3](#_Toc200724422)

[Problem Statement 3](#_Toc200724423)

[Solution overview 3](#_Toc200724424)

[High Level Architecture 3](#_Toc200724425)

[Detailed Description 4](#_Toc200724426)

[Additional Information 6](#_Toc200724427)

# Executive Summary

This document outlines my approach to developing a natural language-based query agent that generates answers by querying data stored in databases.

# Problem Statement

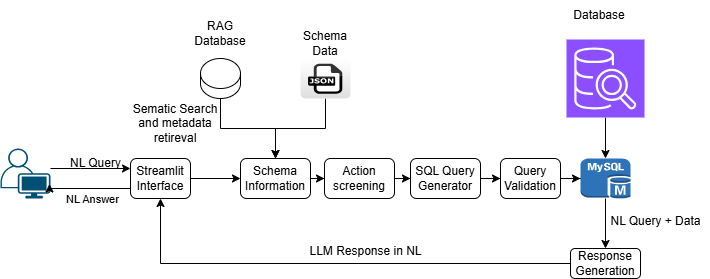
Business users often struggle to access data stored in relational databases due to their lack of SQL knowledge. This creates dependencies on technical teams and delays in data-driven decision-making. A natural language-based query system can simplify data access, but poses challenges in understanding user intent, mapping it to database schema, and generating accurate SQL—especially when multiple tables or joins are involved. Additionally, the system must enforce strict access controls. Certain tables or actions (like UPDATE or DELETE) should be restricted based on user roles, ensuring sensitive data remains secure. Handling these functional and security requirements is essential for building a reliable and safe NL-to-SQL-to-NL solution.

# Solution overview

The proposed solution is a natural language-based query agent that allows users to interact with databases using plain English questions. The system takes a user query, identifies the relevant database and schema, and uses a large language model (LLM) to convert the input into a valid SQL statement. It incorporates schema metadata and role-based access policies to ensure that only authorized users can access specific tables or perform certain actions.

A screening layer classifies the user's intent and prevents harmful operations like data modification or access to restricted tables. The generated SQL is validated for syntax, structure, and policy compliance before execution. Once the query is approved and executed, the final output is returned to the user in a readable format, answering their question based on the underlying data.

# High Level Architecture



# Detailed Description

1. Streamlit Interface (User Interaction Layer)

This is the main interface where users interact with the system through natural language queries. Built using Streamlit, it offers a user-friendly UI with input text boxes and output result panels.

Responsibilities:

* Captures user query in plain English.
* Displays the final answer, which may be in tabular or textual format.
* Routes the query to backend services for processing.

2. RAG Database (Metadata & Semantic Retrieval Layer)

The RAG (Retrieval-Augmented Generation) database is used to support semantic search across schema metadata about databases. It enhances the model’s understanding of the domain by retrieving relevant tables for a given user query.

Responsibilities:

* Stores schema documentation, sample rows, descriptions, relationships, and business logic mappings.
* Performs semantic search to retrieve the most relevant context for a query using vector similarity techniques (e.g., using sentence-transformers).
* Supplies metadata to the query generator to inform prompt engineering.

Implementation:

* Use FAISS to store and retrieve document embeddings. (Which are open source)
* Documents: schema descriptions, table relations, business rules, FAQs.
* Embeddings generated using models like sentence-transformers or OpenAI embeddings.

3. Schema Data (Structured Metadata in JSON Format)

This component holds the raw schema information in a structured JSON format. This schema is used for lookup and verification at various stages of query generation and validation.

Responsibilities:

* Stores table names, column names, data types, keys, and constraints.
* Includes policy information like column-level access restrictions.
* Enables lookup of schema information for query generation and action screening.

Implementation:

* Export DB schema into structured JSON.
* Store table names, columns, types, and foreign keys.

4. Schema Information Module

This module integrates data from both the RAG semantic retrieval and the static schema metadata to extract relevant context for a given user query.

Responsibilities:

* Identifies which tables and columns are relevant.
* Extracts foreign key relationships to determine if joins are needed.
* Prepares schema-aware prompt context for the LLM.

Implementation:

* Python functions that extract required schema subset using table/entity names detected from the NL query.
* Merges retrieved metadata from RAG and static schema.

5. Action Screening Module (Security & Intent Classification)

Before generating SQL, this module classifies the user’s intent and enforces security policies and access control.

Responsibilities:

* Classifies intent (SELECT, INSERT, UPDATE, DELETE, etc.).
* Rejects disallowed actions such as schema modifications or deletion commands.
* Checks if the user has permission to access particular tables/columns.
* Uses rules or fine-tuned classification models for intent recognition.

Example: If a user attempts to “delete all inactive customers,” this module blocks the request and returns a secure error.

Implementation:

* Rule-based filters and intent classifier.
* Maintains a list of disallowed tables and operations.

6. SQL Query Generator (LLM-Based NL2SQL)

This is the core of the system, where an LLM (like T5, Codex, GPT-4, or OSS models like SQLCoder) is prompted with the user query and relevant schema context to produce the corresponding SQL.

Responsibilities:

* Takes user input, schema, and retrieved metadata.
* Constructs prompt in the format
* Outputs an optimized SQL query
* Supports multi-table joins and nested conditions based on input complexity.

Implementation:

* Prompt construction using schema + user query.
* Use Llama instruct 3.2 3B

7. Query Validation Module

Before sending the generated SQL for execution, it passes through this validation checkpoint to ensure:

* SQL syntax correctness (using parsers or dry runs).
* Avoids risky or unsupported SQL constructs.

8. Query Execution (Database Layer)

Once validated, the query is executed on the appropriate database.

Responsibilities:

* Selects the target database connection based on metadata.
* Executes query using a DB driver.
* Collects result set and passes it to the response module.

9. Response Generation (LLM + Formatting)

Submit natural query and data retrieved from database as prompt to LLM for a natural language response.

Responsibilities:

* Converts tabular data into a human-readable explanation (e.g., “There are 12 active sales orders in Q2”).
* Summarizes or groups results as per query.
* Returns the final response to Streamlit for rendering.

# Additional Information

* User will have provision to modify prompt in configuration settings via streamlit screen.
* Database access control via streamlit screen.
* Login function will also provide for restricted access to interface.
* Sample questions will be provided for reference in chat screen.
* Answer will be word-by-word pattern in chat interface.