ABSTRACT

The Intelligent Query Generator (IQA) is designed to bridge the gap between natural language and SQL queries, enabling seamless interaction with relational databases. This system is particularly valuable for users who lack SQL expertise but need to retrieve data efficiently. By leveraging Large Language Models (LLMs), IQA dynamically generates SQL queries based on user input, improving accessibility and ease of use.

A key component of the system is metadata extraction, which gathers essential information about the database, including table structures, relationships, constraints, and Data Definition Language (DDL) statements. This metadata ensures that the LLM has a well-informed understanding of the database schema, leading to more accurate and context-aware query generation. Additionally, dynamic prompt engineering is employed, where database-specific prompts are constructed to guide the model in producing precise SQL queries.

To enhance reliability, the system incorporates a validation phase both before & after the query generation where generated SQL queries are parsed and checked against the database schema. This step helps prevent syntax errors and ensures that only valid queries are executed. Security is also a central focus—non-SELECT queries are restricted to prevent unauthorized modifications, ensuring data integrity.

The query processing pipeline follows a structured approach. First, the user provides a natural language query, which is analysed to identify relevant tables and relationships. A corresponding SQL query is then generated using a dynamically tailored prompt, integrating extracted metadata. Before execution, the query undergoes validation and parsing to confirm correctness. Once verified, it is executed against the target database, and the results are formatted and presented to the user.

Unlike traditional query generators, the IQA adapts to any relational database schema without requiring manual configuration. The combination of metadata extraction and dynamic prompt construction allows for flexibility across diverse database environments, making it highly scalable. Furthermore, to refine query accuracy, similarity search techniques will be incorporated, allowing the system to reference existing queries and generate more precise SQL outputs.

By automating SQL generation and validation, this project significantly lowers the barrier for non-technical users to interact with databases. It offers a reliable,

efficient, and secure solution for converting natural language into structured SQL queries, improving data retrieval and decision-making. The IQA serves as a valuable tool for organizations seeking to enhance database accessibility while maintaining high accuracy and security standards.

Key Words:

- SQL
- Relational Databases
- Large Language Models (LLMs)
- Metadata Extraction
- Database Schema
- Data Definition Language (DDL)
- Dynamic Prompt Engineering
- Parsing
- Query Processing Pipeline
- Natural Language Query Interpretation
- Database Adaptability
- Similarity Search Techniques
- Automated SQL Execution
- Data Retrieval Efficiency

List of Symbols & Abbreviations used

API – Application Programming Interface
LLM – Large Language Model
DDL – Data Definition Language
SQL – Structured Query Language
RAG – Retrieval-Augmented Generation
NLP – Natural Language Processing
DB – Database
RAM – Random Access Memory
GPU – Graphics Processing Unit
REST – Representational State Transfer
JSON – JavaScript Object Notation
HTTP – Hypertext Transfer Protocol
POST – HTTP Method for sending data to a server
GET – HTTP Method for retrieving data from a server
Ollama – Framework for running local LLMs
Llama 3.1 (8B) – The specific model used for SQL generation
Embedding – Process of converting text into vector representation
Flask – Python-based web framework used for API development
MySQL – Relational database management system used for testing
Docker – Containerization tool used for simulating MySQL
Schema – The structure defining tables, columns, and relationships in a database
Prompt – Input text that guides an LLM to generate a response
Metadata – Data describing the structure and properties of other data

List of Figures

Table of contents

Introduction & Background

The ability to interact with databases using natural language is an evolving need, especially for users who lack technical expertise in SQL. Traditional database query systems require users to understand complex schemas, relationships, and query syntax, which can be a barrier to efficient data retrieval. To bridge this gap, the Intelligent Query Generator (IQA) leverages Large Language Models (LLMs) to dynamically generate SQL queries from natural language inputs.

This project explores how dynamic prompt engineering combined with metadata extraction can enhance query generation accuracy. By extracting database schema information, such as table structures and relationships, and integrating it into tailored prompts, the system ensures context-aware query formation. Unlike conventional static query generators, IQA dynamically adapts to any database structure, making it a scalable and adaptable solution.

The significance of this research lies in its potential to streamline data accessibility for non-technical users, reducing reliance on database administrators and minimizing human errors in query formulation. Furthermore, despite utilizing Llama 3.1 8B a significantly smaller model, the system effectively generates complex SQL queries, showcasing its efficiency even within a limited context window.

Business Process Flow

1. **Database Setup**

   - The user provides database credentials.
   - IQA extracts DDLs (schema definitions) of all tables.

2. **Metadata Extraction & Prompt Preparation**

   - Table descriptions are generated using an LLM based on the extracted DDLs.
   - These descriptions are used to create dynamic prompts relevant to the schema.
   - The system is now ready to process queries.

3. **Query Processing & Validation**

   - The user inputs a natural language query.
   - The query is validated before proceeding to generate the final prompt.
   - If validation is successful, a final prompt is generated using the extracted metadata and dynamic prompts.

4. **SQL Generation & Execution**

   - The LLM processes the prompt and generates an SQL query.
   - The generated SQL query undergoes another validation to prevent malicious queries.
   - If valid, the query is executed, and results are returned to the user.

Problem Statement

In many organizations, accessing structured data from relational databases requires SQL expertise. Non-technical users, such as business analysts and decision-makers, often rely on database administrators or developers to write SQL queries, leading to delays and inefficiencies.

Key Challenges:

- **SQL Expertise Barrier**: Users without SQL knowledge struggle to retrieve relevant data.
- **Human Dependency**: Frequent reliance on database administrators increases workload and response time.
- **Error-Prone Query Formulation**: Manually written queries may contain syntax or logical errors.
- **Scalability Issues**: Traditional query generation approaches require manual adjustments for different database schemas.
- **Inconsistent Query Performance**: Manually written queries may not be optimized for efficiency, leading to slow performance.
- **Difficult Query Debugging**: Users struggle to debug and refine queries without proper SQL knowledge.
- **Limited Reusability**: Manually written queries often serve a single purpose and are not easily adaptable for future queries.
- **Time-Consuming Query Writing:** Even for technical users, crafting complex queries for insights can be time-intensive.
- **Limited Adaptability to Schema Changes**: When the database schema changes, manually written queries need frequent updates.
- **Knowledge Transfer Challenges**: As employees leave or transition roles, SQL expertise is not always well-documented or easily transferable.

This project aims to automate SQL query generation using LLMs while ensuring accuracy, security, and adaptability across different relational database schemas. Through dynamic prompt engineering and metadata-driven query validation, our solution enables seamless interaction with databases, empowering non-technical users to retrieve insights efficiently.

A Step Towards Automatic REST API Generation

Beyond query generation, this approach serves as a foundation for automating REST API creation. By dynamically understanding database schemas and generating structured queries, the system can be extended to automatically expose API endpoints based on detected tables and relationships. This would eliminate the need for manual API development for CRUD operations, streamlining backend automation and enabling faster application development.

Objective of the Project

The primary objective of this project is to develop an Intelligent Query Agent (IQA) that enables seamless interaction between users and relational databases without requiring SQL expertise.

The system aims to:

- **Automate SQL Query Generation** – Convert natural language queries into SQL with high accuracy.
- **Improve Accessibility** – Enable non-technical users to retrieve data without relying on database administrators.
- **Enhance Query Validation** – Ensure generated SQL queries are syntactically and semantically correct before execution.
- **Increase Query Efficiency** – Dynamically optimize queries for performance and accuracy.
- **Adapt to Any Database Schema** – Automatically extract metadata and adjust prompt generation for different database structures.
- **Reduce Human Error** – Minimize syntax mistakes and logical errors in SQL queries.
- **Accelerate Data Retrieval** – Streamline the process of querying large datasets without manual intervention.

Uniqueness of the Project

The Intelligent Query Generator (IQA) stands out due to its innovative approach to SQL query generation and validation.

The key aspects of the project:

- **Dynamic Prompt Engineering** – Unlike static query generators, IQA dynamically constructs prompts using real-time metadata, improving SQL accuracy and relevance.

- **Metadata-Driven Adaptability** – The system extracts DDLs and generates table descriptions, allowing it to adapt to any relational database schema without manual configuration.

- **Two-Level Query Validation** – Ensures SQL safety by validating queries before and after generation, reducing errors and security risks.

- **Efficient Query Processing with Compact Models** – Demonstrates that even smaller models (Llama 3.1 8B) can generate complex SQL queries with limited context, making it feasible for real-world business applications.

- **Scalability Without Manual Adjustments** – Can be integrated with different databases without requiring human intervention or predefined query templates.

- **Laying the Foundation for Automatic API Generation** – By structuring validated SQL queries, this project serves as the first step toward fully automated REST API creation, reducing the manual effort needed for backend development.

Benefit to the Organization

The Intelligent Query Generator (IQA) project was inspired by real-world challenges faced at **SAP Concur**, where a **monolithic**, role-based configuration tool is being replaced with a modern microservices-based solution. This legacy tool interacts with **hundreds** of database tables and executes **complex**, multi-joined SQL queries, making it **difficult** for users without deep SQL **expertise** to extract meaningful insights.

Key Benefits to the Organization

1. Simplifying Data Retrieval in Complex Systems

The existing system at Concur relies on predefined queries that are often long and difficult to modify. IQA allows users to generate SQL dynamically using natural language, significantly reducing dependency on SQL experts.

2. Reducing Manual Query Writing Effort

Database administrators and developers currently spend significant time writing, debugging, and optimizing SQL queries. By automating query generation and validation, IQA improves efficiency and minimizes human errors.

3. Enhancing System Scalability

The traditional approach requires manual adjustments whenever the database schema evolves. IQA dynamically extracts metadata and generates prompts, allowing it to adapt to schema changes without additional manual effort.

4. Laying the Foundation for Automated API Generation

Beyond SQL generation, this approach is a first step toward automatic REST API creation. With further refinement, IQA could be extended to generate API endpoints dynamically, enabling seamless integration with external systems.

5. Driving Innovation with LLM-Powered Agentic Systems

The project leverages Llama 3.1 8B to generate SQL queries, demonstrating that even with a smaller context window, effective and complex queries can still be produced. This innovation opens doors for further LLM-powered automation in enterprise software.

Scope of Work

The primary focus of this project was to research and develop an Intelligent Query Generator (IQA) capable of dynamically generating SQL queries using LLMs. Given the complexity of modern database schemas, this phase concentrated on foundational elements necessary for automating query generation.

Key areas covered in this phase:

Research & Inspiration: Studied past & existing approaches, including Uber's recently revealed*[1]* Query GPT, their internal tool designed to automate SQL generation.

Database Schema Extraction: Developed mechanisms to dynamically extract DDL statements from databases, providing structural insights.

Metadata-Driven Prompt Engineering: Created prompt templates dynamically using extracted schema metadata to improve LLM-driven SQL generation.

Validation Mechanism: Implemented query validation to ensure correctness and prevent execution of invalid or potentially harmful queries.

Database Simulation: Used a controlled environment to simulate real-world database queries and evaluate system performance.

Exploration of Similarity Search

An additional research focus was embedding-based similarity search to enhance query accuracy. While full-scale implementation was not completed due to time constraints, initial tests were conducted using Llama 3.1's embedding endpoint (via Ollama). This step laid the groundwork for integrating query similarity search, which remains an ongoing enhancement for future iterations of IQA.

Despite the challenges, this phase successfully established the core framework for the dynamic generation, validation, and execution of SQL queries, ensuring a strong foundation for further advancements.

Solution Architecture

The Intelligent Query Generator (IQA) is designed with a modular architecture to ensure scalability, flexibility, and efficient query generation. The system is built around key components that handle metadata extraction, dynamic prompt generation, query validation, and execution. This section outlines the architectural components and their interactions.

Architectural Components

Ollama & Local LLM Deployment

Ollama is a lightweight framework for running large language models (LLMs) locally without relying on cloud-based APIs. In this project, Ollama is used to serve Llama 3.1 (8B), providing both query generation and embedding functionalities.

Install Ollama

1. Ollama can be installed via a dedicated binary for macOS and Windows, while Linux users can use the following command:
   curl -fsSL https://ollama.com/install.sh | sh *[2]*

2. Next in the terminal we run **ollama run llama3**
   This command will start the model, and we can then interact with it directly in your terminal. But in our case we serve it via a server.

3. To run the Ollama server in the background, we use the command **ollama serve**

We utilize two key Ollama endpoints:

- **/generate** – Handles SQL query generation from dynamically constructed prompts.
- **/embedding** – Converts text into vector representations (tested but not fully integrated).

By running Ollama on a local machine, we eliminate dependency on external APIs, ensuring better privacy, faster inference, and offline capabilities.

Application Components

1. Controllers (API Endpoints for Interaction)

- **db_metadata_controller** - Handles DDL extraction and metadata generation.
- **generate_prompt_controller** - Creates dynamic prompts using metadata and templates.
- **user_query_controller** - Processes natural language queries and returns results.

2. Services (Core Business Logic)

- **db_service** - Manages database connectivity and schema extraction.
- **mysql_service** - Executes SQL queries and handles validation.
- **insight_service** - Processes user queries and generates insights.
- **llm_service** - Calls Llama 3.1 8B for SQL generation and metadata enrichment.
- **prompt_loader** - Loads and manages prompt templates dynamically.
- **validation_service** - Ensures queries conform to schema constraints.

3. Prompt Templates (Dynamic Query Processing)

- **insight_query_prompt.txt** - Generates SQL queries based on user input.
- **validate_query_prompt.txt** - Ensures the generated SQL adheres to schema constraints.
- **table_descriptions.txt** - Stores LLM-generated descriptions for schema awareness.

LLM Service Code Snippet (llm_service.py)

```python
import requests

API_URL = "http://localhost:11434/api/generate"
HEADERS = {"Content-Type": "application/json"}

def call_llm(prompt, max_retries=3):
    """Generic function to call LLM API with retries."""
    body = {"model": "llama3.1", "prompt": prompt, "stream": False}

    for _ in range(max_retries):
        response = requests.post(API_URL, json=body, headers=HEADERS)
        if response.status_code == 200:
            return response.json().get("response", "")

    return ""  # Return empty string if all retries fail
```

This service acts as an interface between the application and the locally hosted Llama 3.1 model served via Ollama. It sends dynamically constructed prompts to the **/generate** endpoint and retrieves the generated SQL queries or insights.

- Uses HTTP POST requests to interact with the Ollama API.
- Implements a retry mechanism to handle temporary failures.
- Ensures that only the extracted response is returned, preventing unnecessary metadata from interfering with processing.

This abstraction allows seamless integration of LLM-generated SQL without exposing internal complexities to other parts of the system.

Prompt Templates

1. Generate Table Description Prompt

```
RETURN ONLY THE SUMMARY AND NOTHING ELSE.
Generate a concise summary (max 30 words) describing the purpose,
key attributes, and relationships of the following database table. Avoid
special characters.
Ensure clarity for semantic search.
RETURN ONLY THE SUMMARY AND NOTHING ELSE.

Table DDL:
{ddl}
```

- This prompt is designed to generate a concise summary of a given database table based on its **DDL (Data Definition Language)**. It extracts the table's purpose, key attributes, and relationships while ensuring clarity for semantic search. The strict instruction to "RETURN ONLY THE SUMMARY AND NOTHING ELSE" prevents unnecessary formatting or additional text, making it suitable for embedding-based retrieval and search optimization

2. Generate Query Validation Prompt

```
REPLY ONLY WITH ONE WORD
You are responsible for validating user queries based on the db schema.
Reply strictly with only 1 word "relevant" or "not_relevant"
based on the schema.

Database Schema:
{table_descriptions}

IMPORTANT: DELETE OR UPDATE QUERIES ARE NOT ALLOWED SO RETURN
"not_relevant".

Query: "{query}"
REPLY ONLY WITH ONE WORD
```

- This prompt is designed to validate the relevance and safety of a user's query before processing it further. It ensures that the generated SQL query aligns with the database schema and does not include unauthorized operations.

    - **Strict Response Format:** The LLM must return only one word—either "relevant" or "not_relevant".
    - **Schema Awareness:** Uses {table_descriptions} to validate whether the query fits within the database structure.
    - **Security Enforcement:** Explicitly blocks DELETE and UPDATE queries, automatically marking them as "not_relevant" to prevent unauthorized modifications.

    This validation step helps maintain database integrity and prevents accidental or malicious queries from being executed.

3. Generate Query Insights Prompt

```
RETURN ONLY THE JSON OUTPUT AND NOTHING ELSE.
Task:
Identify and return only the database tables necessary for generating a
correct SQL query based on the given database schema.

Note:
- The user query is in natural language, so table names may not match
exactly.
- Infer the most relevant tables based on their descriptions.

Database Schema:
{table_descriptions}

Query:
{query}

Expected output example
{{{{
    "identified_tables": ["table1", "table2"]
}}}}

RETURN ONLY THE JSON OUTPUT AND NOTHING ELSE.
```

This prompt is designed to identify the most relevant database tables required to generate a correct SQL query. It bridges the gap between natural language queries and database structures by leveraging table descriptions.

Key Points:

- **Natural Language Understanding:** The user query may not directly match table names, so the LLM infers relevant tables based on their descriptions.
- **Schema Awareness:** Uses {table_descriptions} to map the user's intent to actual database tables.
- **Strict JSON Output:** Ensures that the response follows a structured format, returning only an identified_tables list.
- **Pre-Processing for SQL Generation:** This step helps narrow down the scope of SQL query generation, reducing unnecessary computation and improving accuracy.

By extracting only the necessary tables, the system enhances efficiency and prevents unnecessary joins or incorrect table usage in query generation.

4. Generate SQL Final Prompt

```
RETURN ONLY THE SQL STATEMENT.

You are responsible for generating SQL queries based on the user query. The
user query
will be in natural language. For context, you will be provided with the
database schema
in the form of table names, descriptions, and DDLs of the relevant tables.

The SQL dialect used is {dbDialect}.
You will only return the SQL statement.

Database Schema:
{table_descriptions}

DDLs:
{ddls}

Query:
{query}

If the query does not match any known tables, return:
"ERROR: No matching tables found for the given query."

RETURN ONLY THE SQL STATEMENT OR THE ERROR MESSAGE.
```

This prompt is the last step in the SQL generation pipeline, responsible for converting natural language queries into SQL statements.

Key Points:

- **Schema & DDL Awareness:** Uses {table_descriptions} and {ddls} to ensure the generated SQL aligns with the database structure.
- **SQL Dialect Specification:** Adapts to different database engines using {dbDialect} (e.g., MySQL, PostgreSQL).
- **Strict Output Control:** Ensures only the SQL statement is returned, eliminating unnecessary text.
- **Error Handling:** If no matching tables are found, it returns a predefined error message instead of an invalid SQL statement.

This step guarantees that the generated query is both syntactically correct and contextually relevant before execution.

Architectural Diagrams (End-to-End Flow)

Below are the key high level architectural diagrams representing the flow of the Intelligent Query Generator (IQA) system. These diagrams illustrate how the system extracts metadata, generates dynamic prompts, processes natural language queries, validates, generates and executes SQL securely.

1. **Database schema extraction and dynamic prompt generation workflow**

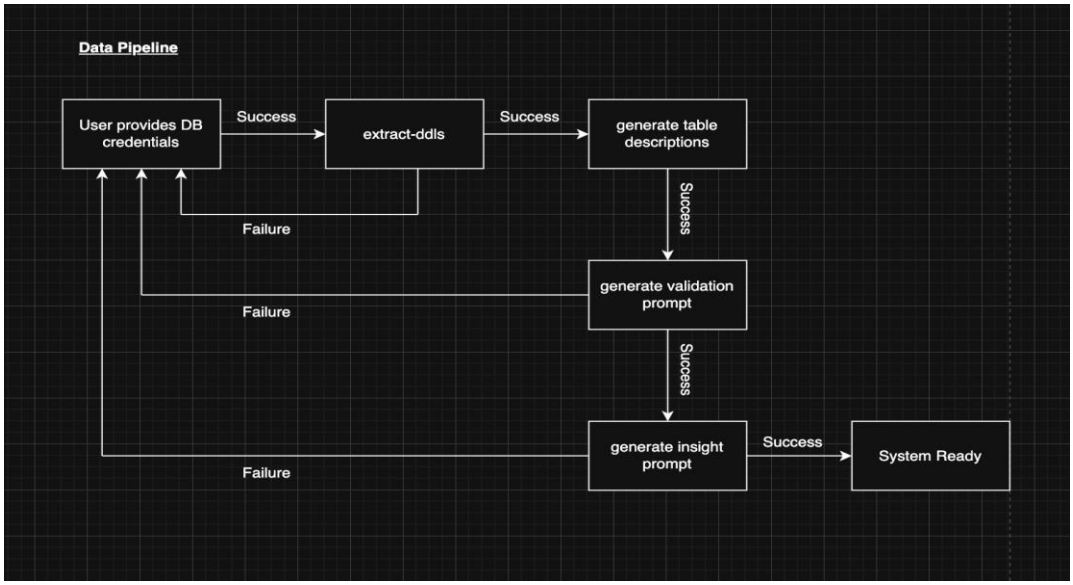

Figure 1.1 - Database metadata extraction & generating dynamic prompts

2. **User query validation and SQL generation workflow**

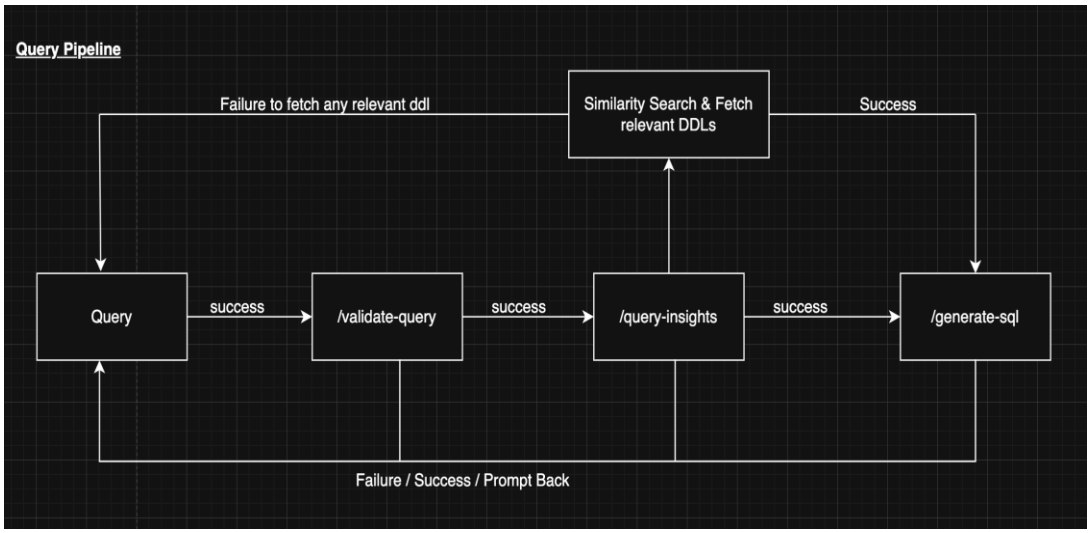

Figure 1.2 – User query validation & SQL generation

Note: The prompt back mechanism & similarity search is scheduled in next iteration of IQA

Resources Needed for the Project

To successfully develop and evaluate the Intelligent Query Generator (IQA) system, the following resources were utilized:

1. People Involved

- Supervisor – Provided guidance on research direction and methodology.
- SAP Concur Team – Offered insights and feedback on real-world business use cases.

2. Hardware Resources

- Device Used – MacBook Pro (M2 chipset)
- System Requirements:
  - Minimum **16GB RAM** recommended
  - GPU (Apple silicon or NVIDIA)
  - Adequate disk space for models and database storage

3. Software & Tools

- Backend Framework - **Flask**
- API Platform - **Postman**
- LLM Model - **Llama 3.1 8B**
- LLM Serving Tool - **Ollama**
- Database - **MySQL**
- Database Simulation & Seeding
  a. **Docker**
  b. Custom Seeding Scripts
- Vector Database (Explored but not integrated yet) - **FAISS**
- Version Control – **GitHub***[3]*
- Python Libraries Used:
  a. **sqlalchemy**
  b. **sqlparse**

These resources collectively enabled the successful development and testing of the system.

Project Plan & Deliverables

High-Level Steps in the Project:

1. **Research & Feasibility Analysis**

- Study existing LLM-based SQL generators.
- Analyse Uber's QueryGPT as an inspiration.
- Explore similarity search techniques for improving SQL accuracy.

2. **Database Schema Extraction & Metadata Generation**

- Use MySQL DDL extraction to retrieve schema details.
- Store DDLs for all tables dynamically.
- Generate table descriptions using LLM (Llama 3.1)

3. **Prompt Engineering & Query Generation**

- Develop prompt templates for query insights and validation.
- Dynamically construct prompts based on extracted metadata.
- Ensure prompts provide context-aware SQL generation.

4. **Query Validation & Execution**

- Implement SQL parsing and validation mechanisms.
- Restrict non-SELECT queries for security.
- Execute validated queries on the simulated MySQL database.

5.**Testing & Refinement**

- Evaluate SQL accuracy against complex queries involving multiple JOINS, GROUP BY, CTE (Common Table Execution)
- Compare generated queries with manually written ones.
- Optimize prompt tuning for improved query generation.

6. **Embedding & Similarity Search (Post Mid-Sem Work)**

- Test embedding generation using Llama 3.1 via Ollama.
- Explore indexing and retrieval methods for similarity search.
- Improve SQL accuracy by referencing past queries.

Deliverables:

- Flask-based backend for SQL query generation.
- MySQL DDL extraction scripts and metadata storage.
- Dynamic prompt engineering framework for SQL generation.
- Query validation and execution pipeline with security measures.
- Dockerized MySQL environment with seed scripts for testing.
- Research insights & performance evaluations on query accuracy.

Work Accomplished So Far

The project follows a multi-step process, divided into two major workflows:

1. Data Pipeline (System Initialization – One Time Setup)

This phase prepares the system by extracting database schema information and generating metadata for efficient query generation.

- **Database Connection Setup:** The user provides MySQL credentials.
- **DDL Extraction:** The system fetches schema definitions (DDLs) for all tables and stores them in the /**ddls** directory.
- **Table Description Generation:** Using Llama 3.1 (8B) via Ollama, the system generates descriptions for each table based on the extracted DDLs and stores them in **metadata/tables/table_descriptions.txt** .

Dynamic Prompt Engineering:

Why is dynamic prompt engineering needed?

- Unlike static prompts, which are hardcoded and inflexible, dynamic prompts adapt to different database schemas without requiring manual modifications.
- This ensures that the prompt is always relevant to the schema at hand, leading to more accurate SQL generation.
- It helps in context-aware SQL formulation, ensuring that queries align with table relationships and column constraints.

How does it work?

The extracted metadata is used to dynamically generate:

- **Insights Query Prompt**: Guides the LLM in understanding the user's intent.
- **Validation Query Prompt:** Ensures generated SQL queries are accurate and secure before execution.
- **System Ready for Query Processing:** Once metadata and prompts are set up, the system is ready for real-time queries.

2. User Workflow (Query Processing in Real Time)

Once the system is initialized, the user can begin interacting with it. This process

- **User Input:** The user enters a natural language query.
- **Input Validation:** The input is checked for ambiguity, and necessary metadata is retrieved.
- **Dynamic Prompt Construction:**
  - The final prompt is generated using schema descriptions and stored metadata.
  - The prompt is structured to provide the LLM with precise contextual information, allowing it to generate accurate SQL queries.
- **SQL Query Generation:** Llama 3.1 processes the final prompt and generates an SQL query.
- **Generated SQL Validation:** The generated SQL undergoes additional validation to prevent malicious queries or invalid syntax.
- **Query Execution & Result Retrieval:** If valid, the query is executed against the database, and the results are returned to the user.

API Requests & Responses

Overview of API Flows

Our system consists of two primary flows:

1. **Data Pipeline Execution Flow** – This prepares the necessary metadata before processing user queries.

   The steps are:

   a. Extracting database schema (DDL)
   b. Generating table descriptions
   c. Creating validation prompts
   d. Creating query insight prompts

2. **User Query Processing Flow** – This processes user queries and generates the final SQL statement.
   The steps are:

   a. Validating the user query
   b. Identifying relevant tables
   c. Generating the SQL query

Data Pipeline APIs

1. Extract DDLs

Endpoint: POST **/extract-ddls**

Request:

```
{
   "host": "localhost",
   "port": "3306",
   "database": "testdb",
   "username": "something",
   "password": "something"
}
```

Response:

```
{
   "message": "DDL extraction completed successfully"
}
```

2. Generate Table Description Metadata

Endpoint: POST /**generate-table-desc-metadata**

Response:

```
{
   "message": "Table description metadata generated successfully"
}
```

3. **Generate Query Validation Prompt**

Endpoint: POST **/generate-validation-prompt**

Response:

```
{
   "message": "Validation prompt generated successfully."
}
```

4. **Generate Query Insights Prompt**

Endpoint: POST **/generate-insight-prompt**

Response:

```
{
   "message": "Insight prompt generated successfully."
}
```

User Query APIs

1. Validate Query

Endpoint: POST **/validate-query**

Request:

```
{
   "query": "Mark all pending payments older than 7 days as ‘FAILED"
}
```

Response:

```
{
   "result": " not_relevant "
}
```

2. Identify Relevant Tables for Query

Endpoint: POST **/query-insights**

Request:

```
{
   "query": "Mark all pending payments older than 7 days as ‘FAILED"
}
```

Response:

```
{
   “result”: {
         “identified_tables”: [
               “orders”,
               “order_items”,
               “payments”
               ]
         }
}
```

3. Generate SQL Query

Endpoint: POST **/generate-sql**

Request:

```
{
   "query": "Find me all orders that are more than 50k"
}
```

Response :

```
{
   result": "SELECT * FROM `orders` WHERE `total_amount` > 50000"
}
```

Key Features Implemented

This project incorporates multiple advanced components to enable efficient and accurate natural language to SQL conversion while maintaining flexibility and scalability.

1. **Flask-Based APIs**

- Developed a RESTful API using Flask to handle database schema extraction, prompt generation, and SQL query validation.
- Modularized API structure with clearly defined responsibilities for schema retrieval, LLM interactions, and query validation.

2. **Three-Layer Architecture**

- Controllers: Handle API request routing and interaction logic.
- Services: Implement business logic such as schema extraction, query validation, LLM calls, and prompt management.
- Dynamic Prompt Generation: Uses custom templates to generate context-aware prompts based on database metadata.

3. **Dockerized MySQL Simulation**

- A local MySQL database is simulated using Docker, allowing for realistic testing environments without external dependencies.
- Custom seed scripts ensure predefined datasets for testing different SQL query scenarios.

4. **Llama 3.1 (8B) Integration via Ollama**

- Runs Llama 3.1 locally using Ollama, eliminating reliance on cloud-based LLMs for better privacy, lower latency, and offline capabilities.
- Uses Ollama's /generate API for SQL generation and /embedding API (tested but not fully integrated) for similarity-based retrieval.

5. **End-to-End SQL Query Validation Pipeline**

- Pre-SQL Validation: Ensures the user query is relevant before SQL generation.
- Post-SQL Validation: Checks if the generated SQL adheres to database constraints and avoids unsafe operations.
- Custom validation rules to reject DELETE/UPDATE queries for security.

These features ensure the project is scalable, efficient, and robust, capable of handling real-world SQL automation scenarios.

Key Challenges Faced During the Project

- Data Pipeline Complexity – Extracting and structuring relevant database metadata dynamically while ensuring minimal latency was challenging.

- Model Constraints – Llama 3.1 (8B) has limitations in context retention and processing long queries accurately, affecting complex SQL generation.

- Query Relevance Filtering – Identifying whether a user query was relevant to the given schema while avoiding false positives was difficult.

- SQL Dialect Handling – Generating SQL that aligns with different database dialects required careful schema processing and validation.

- Scalability for Large Schemas – Handling databases with extensive tables and relationships introduced performance bottlenecks in metadata extraction and query generation.

- Ensuring Query Safety – Filtering out potentially harmful queries (e.g., DELETE, UPDATE) without affecting legitimate analytical queries was a challenge.

- Embedding Effectiveness – Vector-based similarity search for schema awareness was explored but required optimization for accuracy.

Potential Risks and Mitigation Plan

1. **Model Hallucination & Misinterpretation** – The LLM may generate incorrect SQL even with validation.

    - **Current State:** We validate queries post-generation but do not yet have a confidence scoring mechanism.
    - **Mitigation:** Introduce a feedback loop where incorrect queries can be flagged and fine-tuned over time.

2. **Scalability & Performance** – Running Llama 3.1 locally works for development but may struggle with real-time workloads.

    - **Current State:** We rely on Ollama for local inference without optimizations.
    - **Mitigation:** If deployed, consider using optimized LLM hosting (e.g., ONNX runtime, Triton Inference Server) or cloud-based alternatives.

3. **Security Risks Beyond Query Restriction** – While we prevent write operations, other risks remain.

    - **Current State:** Only SELECT queries are allowed, but there's no user authentication or role-based access.
    - **Mitigation:** Implement API authentication and role-based query permissions for multi-user access.

4. **Handling Large Schemas & Complex Queries** – Current schema extraction works, but efficiency drops with larger databases.

    - **Current State:** We extract and process schema metadata in-memory, which may slow down for large-scale databases.
    - **Mitigation:** Optimize metadata retrieval using indexing, caching, or precomputed schema embeddings.

5. **Production-Ready Model Serving** – Ollama is fine for local testing but isn't designed for large-scale deployments.

    - **Current State:** We haven't set up a scalable inference backend.
    - **Mitigation:** Evaluate a dedicated inference engine or cloud-hosted LLM API if scaling is required.

Plan for Remainder of the Project

The next phase of the project focuses on refining core functionalities, enhancing search, performance, improving security, and expanding testing. Several key areas need attention before finalization.

Upcoming Tasks

Optimization & Performance Tuning:

- Implement similarity search for better query-table matching.
- Improve SQL generation and validation speed.
- Test efficiency across different database schemas.
- Security & Robustness Enhancements:
- Implement authentication for API endpoints.
- Restrict API access to prevent misuse.
- Enhance query logging for auditing and debugging.

Model Upgrades & Experimentation:

- Explore newer LLM models (e.g., larger or optimized variants of Llama).
- Experiment with different embedding models.
- Compare performance across different architectures.
- Testing & Schema Validation:
- Validate the system against multiple database schemas.
- Expand test cases with complex queries inspired by real-world enterprise use cases.
- Assess query reliability across different SQL dialects.

Finally, the project will focus on completing technical documentation and final report writing, ensuring that all architectural choices, implementation strategies, and results are well-documented. If feasible, alternative deployment strategies (e.g., cloud-based vs. local deployment) will be explored for scalability.

Appendices

Appendix A: Example Database Schema

Below is a sample schema used for testing the system:

Tables:

- orders (id, customer_id, total_amount, order_date)
- order_items (id, order_id, product_id, quantity, price)
- payments (id, order_id, amount, payment_status, payment_date)

These tables represent a relational structure where orders are linked to order_items, and payments are associated with orders to track transactions.

Appendix B: Sample User Queries

- "List all orders above 50k"
- "Show me the products that got the most 5 star reviews"
- "Drop the table payments"

Appendix C: Example Prompt Templates

- Generate Table Description Prompt – Extracts concise descriptions of tables based on their DDL.
- Query Validation Prompt – Ensures queries align with the schema and checks for restricted operations.
- Query Insights Prompt – Identifies relevant tables based on natural language input.
- SQL Generation Prompt – Converts user queries into executable SQL statements.

References

[1] - https://www.uber.com/en-IN/blog/query-gpt/

[2] - https://ollama.com/download/linux

[3] - https://github.com/samrat-19/iqa-core

Bibliography

1. Anthropic's Blog Post About Agents

2. Anthropic's guide on semantic search & RAG

3. Ollama Documentation