

# **DSCI 551 - ChatDB 43 Project Report**

## **Team Members:**

Alka Anson - 1232943458

Weona Lazarus - 5318528720

**GitHub/Drive Code Link:** [ChatDB 43 Code](#)

## **1. INTRODUCTION**

Data querying remains a technical barrier for many business users. This project addresses that challenge by implementing a Natural Language Interface (NLI) that translates user-entered English questions into executable MongoDB queries using Google's Gemini Flash API, a large language model (LLM). This system enables intuitive interaction with structured databases through natural language, making data access more inclusive and efficient.

We use a modular architecture that allows users to input queries via a Streamlit frontend, then route them through a prompt-generation engine that injects the necessary database schema, sends the request to Gemini Flash, and executes the resulting query on the appropriate MongoDB or MySQL database.

The chatbot is connected to three comprehensive and widely-used database schemas:

1. FIFA – A rich dataset encompassing various football-related statistics and records,
2. AdventureWorks – A sample business database provided by Microsoft, commonly used for demonstrating relational database concepts
3. Bike Store – A retail database focused on bicycle sales, inventory, and customer management.

## **2. PLANNED IMPLEMENTATION**

The initial project proposal for **ChatDB 43** aimed to design and develop a Natural Language Interface (NLI) capable of translating plain natural language queries into executable SQL and NoSQL database commands. The primary objective was to create a user-friendly tool that would allow non-technical users to interact with both structured and semi-structured data through a simple web interface.

### **Initial Proposal Highlights**

#### **Objective:**

To build a chatbot that leverages a Large Language Model (LLM) to convert user queries into SQL (for MySQL) and CQL (for Apache Cassandra), enabling database interaction through natural language.

#### **Scope:**

- Targeted **two types of databases**:
  - MySQL for structured relational data

- Apache Cassandra for unstructured/semi-structured data
- Intended support for a broad range of query operations:
  - **SQL**: SELECT, JOIN, WHERE, GROUP BY, HAVING
  - **CQL**: FIND, AGGREGATE, LOOKUP

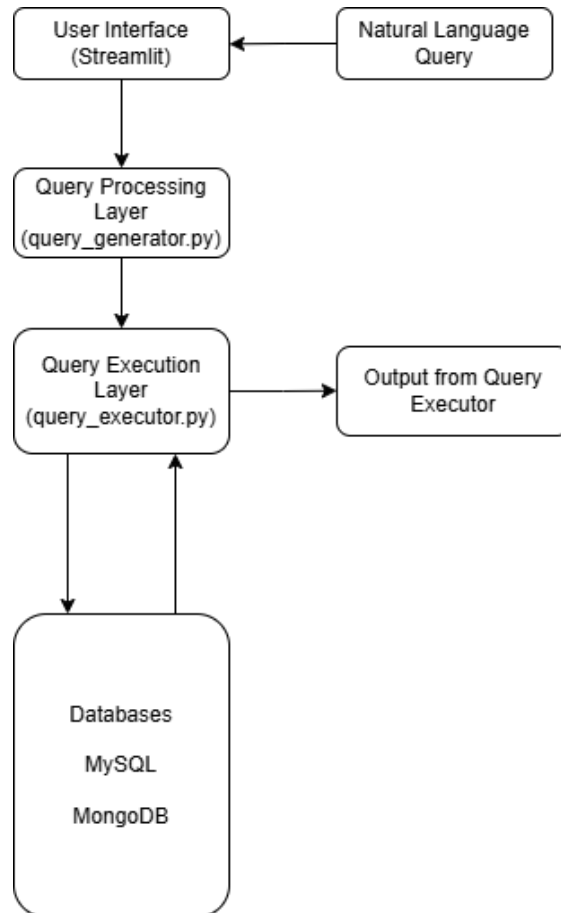
#### Architecture & Design Considerations:

- Planned frontend using **React** for a responsive user interface.
- Backend services via **FastAPI**, enabling rapid development of REST endpoints.
- Integration of **Meta's LLaMA model** for natural language understanding.
- Use of **Retrieval-Augmented Generation (RAG)** to provide schema context and enhance the accuracy of query generation.
- RESTful communication between frontend and backend components for modularity.

| Feature/Component        | Initial Proposal                         | Final Implementation                         |
|--------------------------|--|--|
| <b>Database Support</b>  | MySQL & Apache Cassandra                 | MySQL & MongoDB (No Cassandra used)          |
| <b>Backend Framework</b> | FastAPI for REST APIs                    | No FastAPI; logic handled entirely in Python |
| <b>Frontend</b>          | React-based interface                    | Replaced with Streamlit for simplicity       |
| <b>LLM Model</b>         | Meta LLaMA + RAG for query understanding | Switched to Gemini Flash 1.5 via Gemini API  |
| <b>Deployment</b>        | Modular API endpoints and microservices  | Single-page Streamlit application            |
| <b>Architecture</b>      | REST APIs for service separation         | No REST APIs; direct method invocations      |

### 3. ARCHITECTURE DESIGN

The architecture of the system is modular, consisting of three primary layers: the user interface (UI), the query processing layer, and the query execution layer. The flow, as shown in the provided diagram, ensures a logical and maintainable separation of concerns:



#### Flow Diagram Description:

- **Natural Language Query:**  
The user enters a question in English, such as “Show all customers from Germany.” This input is captured through the web-based interface.
- **User Interface (Streamlit):**  
This is the front-end layer where the user interacts with the system. Streamlit captures the query and displays the result. It provides feedback on invalid queries and errors.
- **Query Processing Layer (query\_generator.py):**  
This component builds a prompt for the Gemini Flash LLM. It detects the relevant schema based on the input, injects schema fields and relationships, and formats the complete prompt. It then sends this prompt to Gemini Flash via API and retrieves the model-generated query.

- **Query Execution Layer (query\_executor.py):**  
This layer is responsible for executing the generated MongoDB or SQL query. It uses PyMongo for MongoDB and PyMySQL for SQL execution. The result is parsed, converted to a readable format (e.g., DataFrame), and sent back to the UI.
- **Databases (MySQL and MongoDB):**  
The application supports both relational (MySQL) and document-based (MongoDB) databases. Schema definitions for these are either manually written or loaded into the schemas.py module for use in prompt construction.
- **Output from Query Executor:**  
The query result is returned to the Streamlit interface. This can be visualized as a table and potentially extended to include charts and exportable files.

This architecture separates concerns clearly into input handling, generation logic, execution engine, and output rendering.

## **4. IMPLEMENTATION**

### **4.1 Functionalities**

- **Natural Language to Query Conversion:**  
Translates plain English prompts like “List players who scored in FIFA” into syntactically correct SQL or MongoDB queries.
- **Multi-Schema Support:**  
Supports tailored prompts and schema-specific instructions for:
  - FIFA (players, matches, goals)
  - AdventureWorks (products, sales)
  - Bike Store (orders, order\_items)
- **Schema Exploration:**  
Responds to queries like:
  - “What tables are in AdventureWorks?”
  - “What columns does the 'goals' table have?”
- **MongoDB Support:**  
Generates aggregation queries using \$match, \$group, \$project, \$sort, \$limit, and date filters. Handles nested fields and dynamic collections with schema-aware prompts.
- **Query Constructs and Operators:**  
Handles SQL and MongoDB equivalents for:  
SELECT, JOIN, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT, OFFSET, COUNT(), SUM(), and more.
- **Schema-Altering Commands:**  
Supports ALTER operations in SQL and equivalent MongoDB updates.  
Examples:
  - “Add column 'birthplace' to players”
  - “Remove field 'discount' from products”

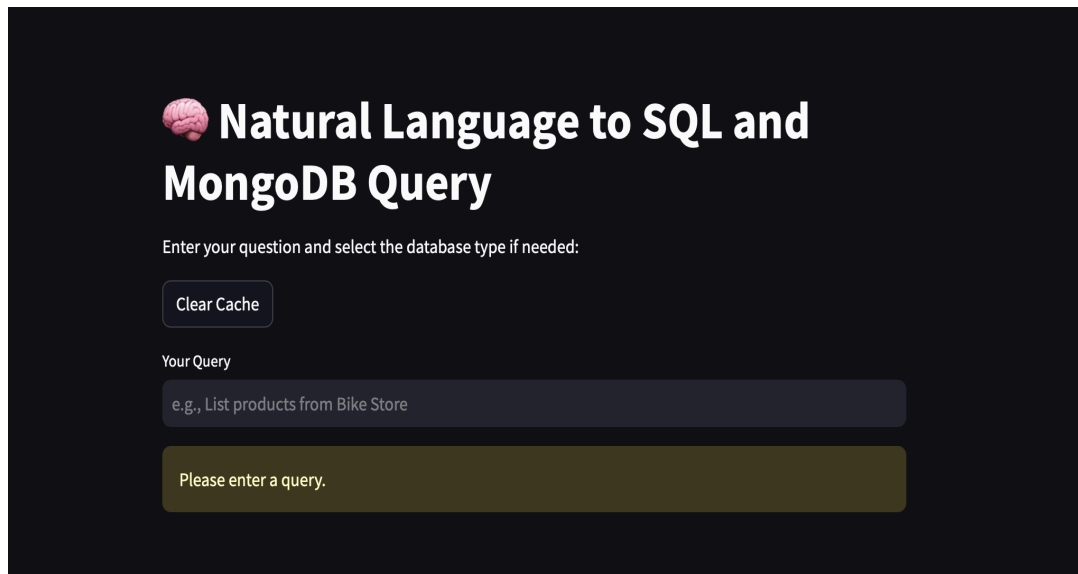
- **Error Handling:**  
Prevents execution of malformed or hallucinated queries by validating against schema.  
Returns empty string for invalid queries (e.g., “reseller” in AdventureWorks).
- **Caching with Streamlit:**  
Uses `@st.cache_data` to store repeated LLM outputs and improve response time.
- **Date and Time Filtering:**  
Implements date filters like `WHERE YEAR(order_date) = 2021` in SQL and equivalent `$match` clauses in MongoDB.
- **Result Display:**
  - SQL results shown in tabular format using pandas
  - MongoDB results displayed as formatted JSON

## 4.2 Tech Stack

- **Frontend:** Streamlit for UI
- **Backend:** Python
- **LLM API:** Gemini Flash via [google.generativeai](#) (Model used: gemini-1.5-flash-8b)
- **Databases:** MongoDB and MySQL
- **Supporting Libraries:**
  - [pymongo](#), [pymysql](#): database execution
  - [json](#), [json5](#): parsing configuration
  - [pandas](#): displaying result sets
  - [re](#), [os](#), [ast](#): utility and parsing logic

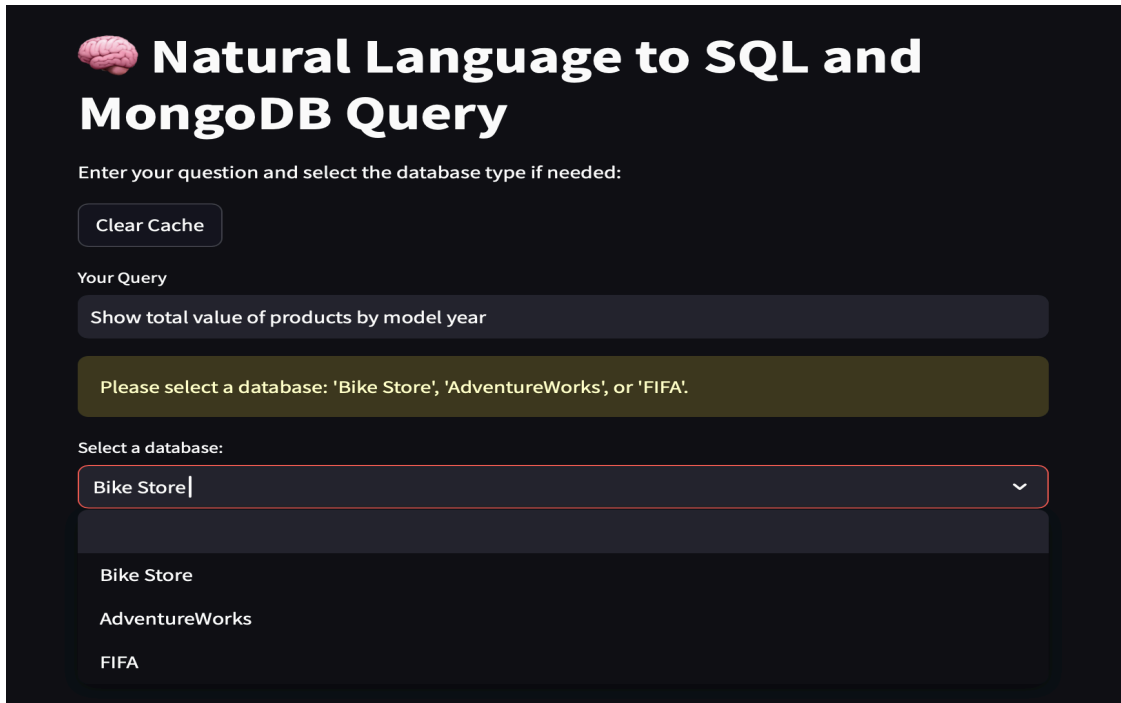
## 4.3 Implementation Screenshots


- Streamlit input interface:



The screenshot shows a web interface with a dark background. At the top, there is a brain icon followed by the title "Natural Language to SQL and MongoDB Query" in large white text. Below the title, a subtitle reads "Enter your question and select the database type if needed:". There is a "Clear Cache" button. Under the heading "Your Query", there is a text input field containing the example text "e.g., List products from Bike Store". Below this is a larger, empty text input field with a placeholder text "Please enter a query.".

- Selection of Dataset after user enters the natural language query:



 **Natural Language to SQL and MongoDB Query**

Enter your question and select the database type if needed:

Clear Cache

Your Query

Show total value of products by model year

Please select a database: 'Bike Store', 'AdventureWorks', or 'FIFA'.

Select a database:

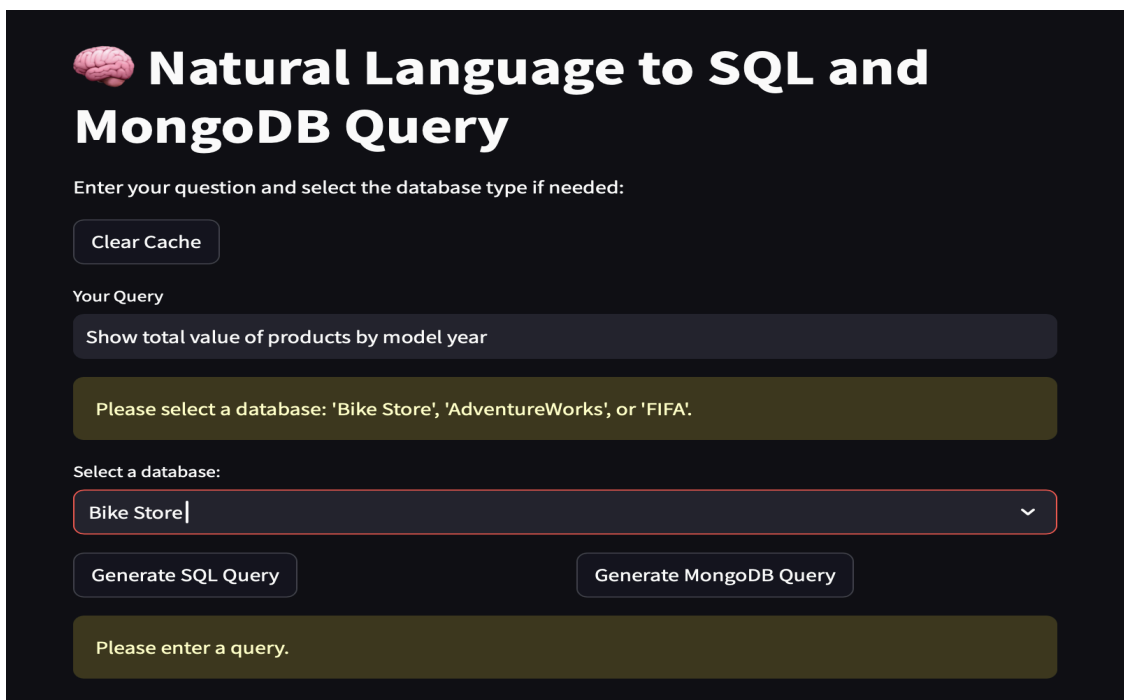
Bike Store


Bike Store

AdventureWorks

FIFA

- Option to select the database:



 **Natural Language to SQL and MongoDB Query**

Enter your question and select the database type if needed:

Clear Cache

Your Query

Show total value of products by model year

Please select a database: 'Bike Store', 'AdventureWorks', or 'FIFA'.

Select a database:

Bike Store

Generate SQL Query Generate MongoDB Query

Please enter a query.

- Query output for SQL Database:

Enter your question and select the database type if needed:

Clear Cache

Your Query

Show total value of products by model year

Please select a database: 'Bike Store', 'AdventureWorks', or 'FIFA'.

Select a database:

Bike Store

Generate SQL Query

Generate MongoDB Query

```
SELECT model_year, SUM(list_price) AS total_value
FROM products
GROUP BY model_year
```

Results:

|   | model_year | total_value |
|---|------------|-------------|
| 0 | 2,016      | 25,487.78   |
| 1 | 2,017      | 108,794.15  |
| 2 | 2,018      | 338,327.97  |
| 3 | 2,019      | 15,499.94   |

- Query output for MongoDB:

Enter your question and select the database type if needed:

Clear Cache

Your Query

Show total value of products by model year

Please select a database: 'Bike Store', 'AdventureWorks', or 'FIFA'.

Select a database:

Bike Store

Generate SQL Query

Generate MongoDB Query

db.products.aggregate([{"\$group": {"\_id":

## **5. LEARNING OUTCOMES**

This project offered rich technical exposure in several areas:

- **LLMs and Prompt Engineering:** Learned how to create effective instruction-style prompts for Gemini Flash and how schema context influences output quality.
- **APIs vs Local Models:** Understood the limitations of local inference due to hardware (e.g., LLaMA, Mistral) and the advantages of using APIs like Gemini.
- **Schema Mapping:** Gained experience writing flexible schema mapping logic to inject relevant collection or table details dynamically.
- **MongoDB Aggregation:** Gained deeper understanding of aggregation pipelines, \$match, \$group, and \$project.
- **End-to-End Deployment:** Learned how to deploy and test applications using Streamlit and virtual environments.

## **6. CHALLENGES FACED**

- **Local LLM Constraints:** Initial experimentation with running local LLMs using llama.cpp proved infeasible due to lack of storage space as well as GPU acceleration and high RAM/VRAM usage.
- **Gemini API Consistency:** The API occasionally generated queries with mismatched braces or incorrect syntax, necessitating post-processing with re.sub and brace checking.
- **Query Validity:** LLM occasionally hallucinated fields not present in the schema, requiring post-validation. To mitigate this, we implemented schema-injected prompt construction, where the relevant field names, data types, and collection structures were programmatically inserted into the prompt. This helped ground the LLM's generation to the actual schema and significantly reduced hallucinations in column or attribute names.
- **Performance:** Initial uncached queries were slow due to API calls, addressed with @st.cache\_data, but large datasets may still pose issues.
- **Schema Ambiguity:** For databases like AdventureWorks, mapping ambiguous or nested structures posed challenges.

## **7. INDIVIDUAL CONTRIBUTION**

This was a two-person collaborative project with well-defined responsibilities for both structured (SQL) and unstructured (MongoDB) database workflows. Tasks were split based on database expertise, LLM integration, and application development.

- **Weona:**
  - Led SQL query generation, schema injection, and validation for AdventureWorks, Bike Store, and FIFA datasets using pymysql for execution.
  - Designed and tested SQL prompts for SELECT, JOIN, WHERE, GROUP BY, and aggregate clauses.
  - Built and refined the Streamlit interface to capture user queries and display results.



- Co-implemented LLM fine-tuning for SQL-focused prompts to improve accuracy and reduce hallucinations.
- **Alka:**
  - Focused on MongoDB query generation using PyMongo, including prompt construction for collections like players, matches, and goals in the FIFA schema.
  - Designed and validated MongoDB aggregation pipelines using \$match, \$lookup, \$project, \$group, and \$sort, tailored to different schema contexts.
  - Investigated various LLMs (e.g., Gemini Flash, LLaMA, Mistral) for feasibility and deployment strategies, including API vs. local hosting.
  - Co-implemented fine-tuning methods to adapt the LLM to MongoDB syntax and edge cases.
- **Shared Responsibilities:**
  - Designed and implemented the full architecture, including prompt pipelines and schema-aware query generation.
  - Integrated Gemini Flash API, managed environment setup, and handled error mitigation in model outputs.
  - Developed and tested Python modules for query parsing, execution, and post-processing.
  - Tried and validated a variety of natural language queries across all supported schemas (FIFA, AdventureWorks, Bike Store) to ensure robust model performance.
  - Troubleshooted and resolved issues such as invalid query formats, LLM hallucinations, and schema mismatches.
  - Integrated Streamlit caching and wrote comprehensive documentation for reproducibility.

## **8. CONCLUSION**

This project showcased a practical application of LLMs in the field of data accessibility. The natural language interface significantly reduces friction in database querying, especially for non-technical users. By integrating Gemini Flash API with schema-aware prompting and a modular execution pipeline, this system achieved reliable performance for translating user intent into structured database queries. The design supports extensibility for multiple databases, making it adaptable for enterprise-level dashboards, internal BI tools, and customer service platforms.

## **9. FUTURE SCOPE**

To enhance the project, the following key areas are identified for future development:

- **Schema Expansion:** Include additional tables (e.g., reseller in AdventureWorks) and fields to support more complex queries with two joins.
- **Real-Time Schema Generation:** Automatically detect and register new tables or collections created in the database, enabling real-time schema updates and prompt injection.
- **Performance Optimization:** Implement batch processing or local NLP models to reduce API dependency and improve speed.
- **Result Visualization:** Incorporate charting capabilities (e.g., Matplotlib or Altair in Streamlit) to visualize query results for better interpretability.