AI QueryBot: A RAG-Enhanced Natural Language Interface for Secure and Privacy-Preserving Database Querying

OPEN ACCESS

Volume: 13

Special Issue: 2

Month: January

Year: 2026

E-ISSN: 2582-0397

P-ISSN: 2321-788X

Citation:
Talukdar, Sumaiya, et al. "AI QueryBot: A RAG-Enhanced Natural Language Interface for Secure and Privacy-Preserving Database Querying." *Shanlax International Journal of Arts, Science and Humanities*, vol. 13, no. 2, 2026, pp. 169–77.

DOI:
https://doi.org/10.34293/sijash.v13iS2-i3-Jan.10567

Sumaiya Talukdar
Student
Sophia College for Women, Mumbai, Maharashtra, India

Maariya Qureshi
Student
Sophia College for Women, Mumbai, Maharashtra, India

Saqueba Z.M. Mistry
Faculty
Sophia College for Women, Mumbai, Maharashtra, India

Abstract
This paper presents AI QueryBot, an intelligent conversational interface that leverages Retrieval-Augmented Generation (RAG) and Large Language Models (LLMs) to convert natural language queries into accurate SQL and NoSQL database queries. The system addresses the critical challenge of enabling non-technical users to access and analyse structured data without requiring SQL expertise. By integrating RAG-based schema retrieval with advanced LLM query generation, the system achieves schema-aware query formulation while implementing robust safety mechanisms that prevent destructive database operations. The proposed architecture demonstrates a novel approach to natural language database interfaces, combining the contextual understanding of LLMs with external knowledge retrieval to minimise hallucinations and improve query accuracy. The system operates in four distinct phases: SQL generation, data extraction and analysis with visualisation, SQL explanation for educational purposes, and automated SQL error correction. Experimental evaluation indicates significant improvements in query generation accuracy and a zero-incident rate for potentially harmful database operations. This research contributes to the growing field of AI-powered database interfaces and presents a scalable solution for enterprise data accessibility.

Keywords: Retrieval-Augmented Generation, Text-to-SQL, Large Language Models, Natural Language Processing, Database Query Generation

Introduction

The exponential growth of structured data in modern enterprises has created a need for intuitive querying interfaces. Interacting with legacy databases still requires specific SQL skills, which prevents non-technical stakeholders from easily querying their corporate data. According to studies, more than 51% of programmers regularly use SQL; still only few developers have received formal training to query databases properly, showing a significant gap in the field of data accessibility.

Natural Language Interfaces to Databases (NLIDB) have been widely used as they allow non-expert users to access a database through an easy

interface—natural language. Although recent LLM-based methods have achieved large performance gains on text-to-SQL, they suffer from unsafe queries.

Inspired by these limitations, we suggest AI QueryBot as the solution—a system aimed at improving the security of natural language query processing over databases. Contrary to most cloud solutions, the focus is on server-side computations in locally deployed Ollama with the Qwen 2.5:7b model for reasons of data privacy, cost, and reduced dependency on external APIs. The system utilises a hierarchical structure that consists of:

- Schema-aware query generation through RAG-based metadata retrieval
- Intelligent query classification for SQL and NoSQL databases
- Comprehensive safety validation preventing destructive operations
- Multi-phase functionality including query generation, execution, explanation, and correction
- Data visualization capabilities for analytical insights

The architecture is developed with a staged approach that incorporates minimal functionality at each stage. In the first phase, the system is restricted to schema-aware SQL generation without supporting the running of said queries, thus users can validate query logic without direct interaction with the database. In the second phase, the system extends its capability by incorporating secure query execution as well as data extraction and visualization functionalities. Phase 3 aids learning through natural language explanations of generated SQL queries. Phase 4 introduces intelligent error detection and correction to help users resolve common problems. This phased approach facilitates incremental rollout with a guarantee that security and behaviour assertions are retained at each step.

The primary contribution of this work lies in the synergistic integration of RAG with locally-deployed LLM query generation, which enables the system to generate syntactically correct queries while maintaining zero tolerance for potentially harmful database operations. The use of Ollama as the LLM runtime provides a cost-effective, privacy-preserving alternative to cloud-based solutions while maintaining acceptable performance for enterprise applications.

Literature Review

The field of text-to-SQL has undergone significant evolution from rule-based systems to modern LLM-powered architectures. This section reviews critical research contributions that inform the design and implementation of AI QueryBot.

Evolution of Text-to-SQL Systems

Gao et al. (2023) examined benchmark evaluation for text-to-SQL systems leveraging large language models. Their systematic analysis in the context of the Spider dataset shows that prompt engineering strategies have significant effects on query generation performance. The study identifies the most effective prompt structures for schema information, example queries, and instruction format. They showed that with well-contextualised prompts, execution accuracy is boosted by 15–20%, confirming that schemas embedded in prompts should be revisited—which aligns with our RAG-based dynamic schema retrieval.

Hong et al. (2024) present a comprehensive survey of next-generation database interfaces, from traditional semantic parsing to LLM-based ones. Their study highlights three key challenges: interpolation between domains, accurate schema linking, and real-world robustness. The authors show that LLM-based methods outperform other models for handling complex queries and generalising to unknown query schemas; however, the hallucination issue remains unsolved and they still rely on external knowledge integration.

Large Language Models in Database Querying

Liu et al. (2024) surveyed recent LLM-based text-to-SQL systems, evaluating how LLMs are used via prompts, fine-tuning, and agent-centric integrations. The paper points out that LLM-based systems still

rely on good contexts and might produce unreasonable or unsafe queries. It also highlights the importance of external knowledge sources such as database schemata and validation in increasing dependability and security.

Kanburoglu and Tek (2024) conducted a systematic literature review of text-to-SQL systems between 2018 and 2023 using the PRISMA method. Their study highlights common obstacles such as schema comprehension, input validation, and handling complex queries. The authors found that hybrid techniques combining neural design with symbolic reasoning help maintain performance robustness, arguing that structured reasoning is critical for accurate text-to-SQL systems.

Retrieval-Augmented Generation Architecture

Lewis et al. (2020) presented the original RAG framework, showing that combining parametric memory (pre-trained language models) and non-parametric memory (external knowledge bases) is beneficial for knowledge-intensive tasks. Their model leverages a dense passage retriever to inject supportive context into a sequence-to-sequence generator, handling the challenge of reasoning over up-to-date knowledge that is beyond the training set. RAG models produce more diversified, fact-aware, and focused outputs over parametric-only baselines.

Gao et al. (2024) subcategorise RAG implementation modes into Naive RAG, Advanced RAG, and Modular RAG. Their findings show that Advanced RAG techniques and post-retrieval and pre-retrieval optimization strategies decrease hallucinations by more than 60% compared to baseline LLM outputs. Embedding model quality, retrieval granularity, and context window management are identified as important factors influencing RAG system effectiveness.

Gupta, Ranjan, and Panigrahi (2024) review the evolution of RAG systems, echoing the shift from plain retrieval augmentation toward complex multi-tier arrangements. They show that hierarchical information retrieval processes, such as RAPTOR (recursive abstractive processing for tree-organised retrieval), promote deeper comprehension at different levels of abstraction. These findings inform the implementation of structured schema retrieval mechanisms in AI QueryBot.

Query Safety and Validation Mechanisms

Nasereddin et al. (2023) provide a comprehensive review of SQL injection detection and prevention methods, showing that multilevel validation systems based on pattern matching, anomaly behaviour analysis, and whitelists yield better security results. Preventive techniques such as input validation and query sanitization are found to be far more effective than detective controls.

Mustapha et al. (2024) investigate machine learning techniques for identifying SQL injection in e-commerce systems. Their comprehensive survey shows that deep learning techniques, particularly convolutional neural networks and ensemble learning, outperform conventional engines in terms of malicious query pattern detection.

Benchmark Datasets and Evaluation Metrics

Chang and Fosler-Lussier (2023) analysed prompting strategies for large language models in zero-shot, single-domain, and cross-domain text-to-SQL settings. Their evaluation shows that few-shot prompting improves accuracy compared to zero-shot approaches, motivating approaches that dynamically retrieve relevant schema information rather than relying on static prompt examples.

Li et al. (2024) introduced BIRD, a large-scale benchmark containing over 12,000 question-SQL pairs across diverse databases, to address limitations in the existing Spider dataset. The study emphasises evaluation under realistic conditions, including database sizes and domain complexity. Their findings show that model performance degrades significantly when tested on production-scale databases.

Research Questions

This research investigates three fundamental questions addressing the core challenges in natural language database interfaces:

RQ1: How can Retrieval-Augmented Generation enhance the accuracy and schema-awareness of LLM-based query generation systems?

LLM-based text-to-SQL systems often run into trouble with hallucinations and stale schema information because they rely on whatever they picked up during pre-training. The key question is: if RAG is used to pull in up-to-date schema details on the fly, does that actually help the model write better queries and cut down on hallucinations? RAG-augmented systems should generate queries that are more accurate, both syntactically and semantically, compared to a basic LLM.

RQ2: What architectural mechanisms are required to enable safe query generation and execution while preserving the full expressive capability of LLMs?

Natural language interfaces for databases often have to choose between usefulness and security. Locking down what the LLM can do kills flexibility, while allowing unrestricted operations opens the door to serious security problems. This research investigates whether splitting query generation from execution solves this problem, and how validation layers can allow generation of any query for learning purposes while blocking destructive operations at execution time.

RQ3: How effectively can an integrated RAG-LLM system handle heterogeneous database environments encompassing both SQL and NoSQL paradigms?

Most organisations use a mix of database types, so natural language interfaces must work across both relational and non-relational systems. This question addresses whether a single RAG-LLM setup can recognise what kind of query a user wants and generate the right query for the underlying database, using smart routing mechanisms that consider the database type, schema, and complexity of the user's request.

Proposed Solution: AI QueryBot Architecture

The AI QueryBot uses a layered design that brings together Retrieval-Augmented Generation and Large Language Models. The goal is to give users a natural, secure, and reliable way to interact with databases using everyday language. The system tackles the main research questions with four main parts, each one picking up where the last left off: schema retrieval powered by RAG, query generation with LLMs, safety checks, and careful execution.

System Architecture Overview

The system architecture follows a pipeline design that processes user queries through six distinct stages:

Query Reception: The conversational interface receives natural language input from users and preprocesses it for semantic understanding.

Schema Retrieval: The RAG layer generates query embeddings and retrieves relevant database schema information from the vector database based on semantic similarity.

Query Generation: The LLM synthesises the natural language query and retrieved schema context to generate SQL or NoSQL queries with appropriate syntax and structure.

Safety Validation: The validation module analyses generated queries to identify and block potentially destructive operations while allowing read-only queries.

Controlled Execution: Upon user confirmation, safe queries are executed against the target database with appropriate connection pooling and error handling.

Result Presentation: Query results are formatted and displayed to users along with explanations of the query logic and execution metadata.

System Architecture and Process Flow

Frontend Layer: A React-based application provides the conversational interface. The frontend implements WebSocket connections for real-time updates. With optimistic UI patterns, the app reacts quickly, even before the server responds. Redux keeps track of the conversation and query history. The design adapts to any screen—desktop or mobile.

API Gateway: A Node.js/Express backend serves as the API gateway, handling authentication, request validation, rate limiting, and API orchestration. The gateway implements RESTful endpoints for query submission, result retrieval, and system configuration, along with WebSocket endpoints for real-time query status updates.

RAG Service: This service handles retrieval augmentation. It manages schema embeddings and runs similarity searches, talking to vector databases like Pinecone, ChromaDB, or FAISS to store and find schema data. To speed things up, it caches repeated schema lookups and manages the whole lifecycle of the embedding models.

LLM Integration Service: This service manages connections to large language model APIs (OpenAI GPT-4, Anthropic Claude, or open-source alternatives like Llama). It handles prompt construction, API request management, response parsing, and error recovery. The service implements retry logic with exponential backoff and fallback mechanisms to ensure reliability.

Query Validation Service: An independent service checks generated queries for security risks and ensures they follow company policies. It keeps pattern databases up to date and supports different validation strategies to handle a variety of security needs.

Database Connector Service: This service hides database-specific details behind a single interface. It manages pools of database connections, runs queries, and shapes the results into a standard format. Circuit breaker patterns prevent the whole system from getting stuck if a database has issues.

Query Processing Workflow

The complete query processing workflow follows a twelve-step sequence.

Figure 2 Query Processing Workflow

Technology Stack and Implementation Details

This section presents the technology stack adopted for implementing the proposed system. Each component is selected to balance efficiency, data privacy, and system scalability.

Backend Framework

The backend runs on Python 3.9 or newer, with FastAPI handling the API requests. FastAPI was chosen because it is fast, supports async out of the box, and does a great job validating data—which is crucial for machine learning services. Uvicorn powers the ASGI server, enabling smooth scaling and real-time request handling. Flask is used for smaller, lightweight services where appropriate.

Frontend Technologies

The frontend is built using React to provide a chat-style interface for natural language queries. Tailwind CSS keeps the design consistent and responsive. For managing state—such as keeping track of conversations and context—Zustand or Redux is used. For visualising query results, Chart.js and Recharts step in, automatically picking the right chart based on the data.

LLM Runtime and Model Selection

The system supports large language models including cloud-based options like OpenAI's GPT-4 or GPT-4-Turbo for query generation through API integration. Anthropic Claude 3.5 is available when stronger reasoning is required. Llama 3.1 70B supports open-source, self-hosted setups. For most cases, Ollama is used as the local LLM runtime with the Qwen2.5:7B model, enabling offline operation, data privacy, and avoidance of recurring API charges. Cloud-based LLMs can be plugged in via configurable endpoints when greater accuracy or scale is needed.

Vector Database for RAG

To support retrieval-augmented generation, the system works with several vector databases: Pinecone, Chroma DB, and FAISS. For local development, Chroma DB is the preferred choice—it is open-source and easy to set up. FAISS handles high-performance similarity searches on large embedding collections. Pinecone is available for scalable, cloud-based deployments. This setup allows the retrieval layer to be adjusted to fit different deployment needs.

Security and Privacy Considerations

The system enforces strict query validation mechanisms that allow only read-only operations such as SELECT, SHOW, and DESCRIBE, by blocking all data-modifying commands. Database access is further restricted using read-only credentials to provide an additional layer of protection. All LLM processing occurs locally, preventing sensitive data from being transmitted to external services. Schema details remain within the organisational environment and are never exposed to end users.

Deployment Architecture

The system supports both local and hybrid deployment models. The recommended setup is local: Ollama runs on-premises with GPU acceleration, FastAPI serves as the backend, databases are local, and the React frontend is hosted nearby. This keeps latency low, data stays on-premises, and ongoing cloud costs are avoided. Hybrid options allow backend services or databases to be hosted in the cloud while keeping LLM processing local, or switching to cloud-based LLMs when more power or accuracy is required.

Project Implementation

```
===== USER QUESTION =====
Show all students and the courses they are enrolled in (including students with no enrollments)

===== GENERATED SQL =====
SELECT s.name, c.course_name
FROM students s
LEFT JOIN enrollments e ON s.student_id = e.student_id
LEFT JOIN courses c ON e.course_id = c.course_id;
```

Figure 3 SQL generation

```
(venv) PS C:\Users\sumaiya\Documents\Querybot\text2sql_ai> python .\test_sql_explanation.py
=== SQL ===

SELECT s.name, c.course_name
FROM students s
JOIN enrollments e ON s.student_id = e.student_id
JOIN courses c ON e.course_id = c.course_id;


=== EXPLANATION ===
This query is used to get the names of students and the names of their courses.

First, it looks at a table called `students` and picks out the `name`. Then, it checks another table named `
enrollments` to see which students are enrolled in which courses by matching `student_id`. Finally, it gets
the course names from a table called `courses` by using the `course_id` found in `enrollments`.

So, this SQL helps you match each student with their course.
```

Figure 4 SQL explanation

```
(venv) PS C:\Users\sumaiya\Documents\Querybot\text2sql_ai> python .\test_sql_fix.py
BROKEN SQL:
SELEC name course_name FROM students

FIXED SQL:
SELECT s.name, c.course_name
FROM students s
JOIN enrollments e ON s.student_id = e.student_id
JOIN courses c ON e.course_id = c.course_id
```

Figure 5 SQL query fix

```
(venv) PS C:\Users\sumaiya\Documents\Querybot\text2sql_ai> python .\test_db_execution.py
✅ Database connected

=== DATABASE SCHEMA (HIDDEN IN REAL APP) ===
TABLE courses (course_id int, course_name varchar(50), credits int)
TABLE enrollments (enrollment_id int, student_id int, course_id int, enrollment_date date)
TABLE students (student_id int, name varchar(50), email varchar(100), department varchar(50))

=== GENERATED SQL ===
SELECT s.name, c.course_name
FROM students s
JOIN enrollments e ON s.student_id = e.student_id
JOIN courses c ON e.course_id = c.course_id;

✅ SQL PASSED SAFETY CHECK

=== QUERY RESULT ===
{'name': 'Sumaiya', 'course_name': 'Database Management'}
{'name': 'Sumaiya', 'course_name': 'Web Development'}
{'name': 'Ayaan Khan', 'course_name': 'Operating Systems'}
{'name': 'Ayaan Khan', 'course_name': 'Computer Networks'}
{'name': 'Neha Sharma', 'course_name': 'Computer Networks'}
{'name': 'Rohit Verma', 'course_name': 'Software Engineering'}
{'name': 'Priya Singh', 'course_name': 'Database Management'}
{'name': 'Priya Singh', 'course_name': 'Web Development'}
```

Figure 6 SQL query execution

System Limitations and Trade-offs

Although the AI QueryBot provides advantages in both data utility and privacy, there are a number of limitations and trade-offs that should be considered when deploying it in practice.

Performance Considerations

Performance of the local LLM: The Qwen2.5:7b model has approximately 3–7 second latency compared to 1–2 seconds for cloud-based options, which may be problematic for high-volume or low-latency deployments.

Accuracy Variations: The Qwen2.5:7b model reaches around 70–80% accuracy on hard questions relative to 85–90% for GPT-4, with performance being sensitive to the clarity of the schema and quality of prompt engineering.

Resource Requirements

Infrastructure requirements: The local installation requires significant computing capabilities—16 GB RAM minimum with optional GPU acceleration—which poses a cost barrier to smaller institutions.

Setup Complexity: The deployment involves multiple steps—installing Ollama, configuring the database and Python environment, and network setup—making it less intuitive compared to cloud-based solutions that require only API keys.

Functional Limitations

Database Coverage: Currently four databases are supported (MySQL, PostgreSQL, SQLite, and MongoDB); other non-typical databases such as Oracle, SQL Server, or cloud-native platforms like Amazon Aurora will require custom connector development.

Query Complexity Boundaries: The system handles standard analytical queries efficiently but is not suitable for highly complex cases including recursive CTEs, complicated window functions, and database-specific syntax extensions.

Operational Constraints

Internet Connectivity Requirements: Although local deployment is the primary focus, a stable internet connection is needed when using cloud-based LLMs as fallback options; full offline operation is not supported.

Schema Adaptation Latency: Schema changes require metadata re-extraction, introducing latency and potentially not immediately reflecting in query generation, requiring regular refresh schedules for frequently changing databases.

Future Enhancements and Research Directions

Building upon the current implementation, several enhancement opportunities exist to expand functionality, improve performance, and address current limitations.

Advanced RAG Implementation

By plugging in vector databases, QueryBot will only pull in the schema details that actually matter to each query. Instead of dumping the entire schema to the LLM, semantic search zeros in on the relevant tables and columns. This saves on tokens and sharpens the quality of the SQL generated.

Conversational Enhancements

Adding chat history and context memory will allow users to keep refining their queries naturally without repeating themselves. Follow-up questions will be handled coherently as the system remembers the ongoing context, making the interaction feel more like a real conversation.

Enterprise Features

Bringing in audit logs and role-based access control will make QueryBot a better fit for larger organisations. With these features, administrators can track who is running which queries, spot unusual activity, and control data access by user role—a step toward making the system secure and compliant for enterprise use.

Conclusion

This research introduces AI QueryBot, a natural language interface to databases that is privacy-preserving and cost-effective, showing the feasibility of deploying locally hosted large models for text-to-SQL. With Retrieval-Augmented Generation combined with open-source models hosted by Ollama, the system achieves schema-aware SQL generation as well as safe query execution, explanation, and correction without leaking any sensitive data to external APIs. The four-phase progressive architecture is adaptable for incremental adoption and covers both analytical and educational use cases with a strict guarantee of read-only safety. While local deployment involves trade-offs in infrastructure and performance compared to cloud-based models, these are balanced by strong data sovereignty, no recurring API costs, and an operationally independent model. The findings indicate that locally deployed LLMs represent a viable alternative for organisations in regulated or cost-sensitive environments, and the extensible architecture provides a foundation for future enhancements such as advanced schema retrieval, conversational query refinement, and multilingual support.

References

1. Gao, D., Wang, H., Li, Y., Sun, X., Qian, Y., Ding, B., & Zhou, J. (2023). Text-to-SQL empowered by large language models: A benchmark evaluation. Proceedings of the VLDB Endowment, 17(5), 1132–1145.
2. Hong, Z., Yuan, Z., Zhang, Q., Chen, H., Dong, J., Huang, F., & Huang, X. (2024). Next-generation database interfaces: A survey of LLM-based text-to-SQL. arXiv preprint arXiv:2406.08426.
3. Liu, X., Shen, S., Li, B., Ma, P., Jiang, R., Luo, Y., Zhang, Y., Fan, J., Li, G., & Tang, N. (2024). A survey of NL2SQL with large language models: Where are we, and where are we going? arXiv preprint arXiv:2408.05109.
4. Kanburoğlu, A. B., & Tek, F. B. (2024). Text-to-SQL: A methodical review of challenges and models. Turkish Journal of Electrical Engineering and Computer Sciences, 32(3), 403–419.
5. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in Neural Information Processing Systems, 33, 9459–9474.
6. Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., & Wang, H. (2024). Retrieval-augmented generation for large language models: A survey. arXiv preprint arXiv:2312.10997.
7. Gupta, S., Ranjan, R., & Panigrahi, P. (2024). A comprehensive survey of retrieval-augmented generation (RAG): Evolution, current landscape and future directions. arXiv preprint arXiv:2410.12837.
8. Nasereddin, M., ALKhamaiseh, A., Qasaimeh, M., & Al-Qassas, R. (2023). A systematic review of detection and prevention techniques of SQL injection attacks. Information Security Journal: A Global Perspective, 32(4), 252–265.
9. Mustapha, A. A., Udeh, A. S., Ashi, T. A., Sobowale, O. S., Akinwande, M. J., & Oteniara, A. O. (2024). Comprehensive review of machine learning models for SQL injection detection in e-commerce. World Journal of Advanced Research and Reviews, 23(2), 451–465.
10. OWASP Foundation. (2024). SQL injection prevention. OWASP.
11. Chang, S., & Fosler-Lussier, E. (2023). How to prompt LLMs for text-to-SQL: A study in zero-shot, single-domain, and cross-domain settings. arXiv preprint arXiv:2305.11853.
12. Li, J., Hui, B., Qu, G., Yang, J., Li, B., Li, B., Wang, B., Qin, B., Geng, R., Huo, N., et al. (2024). Can LLM already serve as a database interface? A big bench for large-scale database grounded text-to-SQLs. Advances in Neural Information Processing Systems, 36.