Enhancing AI-Driven Query Generator by Bridging Natural Language and Cloud Data Base

Michael Harditya[1] , Salma Dewi Taufiqoh[2], Vemby Somadias[3], Kenya Damayanti Priyatna[4], Riri Fitri Sari[5]

[1,2,3,4,5]Electrical Engineering Department, Faculty of Engineering, University of Indonesia
riri@ui.ac.id

Abstract. Accessing data from modern cloud databases remains a significant challenge for users without expertise in Structured Query Language (SQL), creating a bottleneck in data-driven decision-making. This paper introduces Querier, a system that bridges this gap by leveraging a Large Language Model (LLM), specifically GPT-3.5, to translate high-level natural language prompts into efficient, executable SQL queries. The system is designed for seamless integration with cloud data warehouses like Google BigQuery, handling the end-to-end workflow from user input to result visualization. Performance evaluation demonstrates the system's high reliability, successfully processing all test prompts of varying complexity, with an average end-to-end response time of under ten seconds. The results validate the effectiveness of using LLMs to enhance AI-driven query generation, providing an accessible and efficient bridge between human language and complex cloud data, thereby empowering a broader range of users to perform data analysis.

Keywords: SQL, Query Generation, LLM, GPT-3.5.

1 Introduction

1.1 A Subsection Sample

Query Generation is a fundamental process in data analysis for efficiently extracting information from large and complex datasets. This process is crucial as it provides the foundation for evidence-based practices, knowledge discovery, and decision support across various domains[1], [2]. The most dominant query language used in the industry, especially in the era of Big Data Analytics, is SQL[3], [4]. SQL is designed for structured databases, making the information extraction process more accessible and understandable[5], [6].

Despite its popularity, the implementation of SQL in enterprise environments faces several significant challenges. First, manually writing SQL queries is often inefficient and difficult, particularly for users with limited technical skills [7], [8]. This phenomenon is exacerbated by the industry trend of hiring underqualified human resources or

S. C. Asih et al. (eds.), *Proceedings of the 19th International Conference on Quality in Research (QiR 2025)*, Advances in Engineering Research 303,
https://doi.org/10.2991/978-94-6239-717-0_12

assigning non-technical staff to handle databases due to the high demand for data analysts [9]. Second, natural language prompts from users are often varied and ambiguous, with different levels of specificity and granularity [10]. Third, inefficiency in query writing has a direct impact on operational costs. Cloud database platforms like Snowflake, Google BigQuery, and Azure calculate charges based on the computational volume of executed queries ('computed cost') [11]. Suboptimal queries can lead to unnecessary budget inflation.

To address these challenges, advancements in Natural Language Processing (NLP) offer a promising solution. NLP is a branch of artificial intelligence that enables machines to understand, interpret, and generate human language [12], [13]. The implementation of NLP through Large Language Models (LLMs) has been proven to successfully translate natural language instructions into programming code, as demonstrated by popular AI assistants like 'GitHub Copilot' and 'Stack Overflow' [14], [15]. This technology can be leveraged to bridge the gap between data analysis needs and users' technical limitations[16]. However, a gap remains in creating practical, end-to-end systems that seamlessly integrate these models with modern cloud databases for general business users.

To address this gap, this paper introduces Querier, an AI-driven system designed to provide a seamless and efficient bridge between natural language and cloud databases. The primary contributions of this work are: (1) the design and implementation of an end-to-end architecture that integrates a user interface, an LLM-based generation engine, and a cloud database; (2) a methodology for dynamic prompt engineering using database schema to enhance query accuracy; and (3) a performance evaluation that validates the system's efficiency and reliability.

The remainder of this paper is structured as follows. Section 2 reviews related work in the field of query generation. Section 3 details the methodology and system architecture of Querier. Section 4 presents the results of our functional and performance testing. Finally, Section 5 discusses the implications of the findings, and Section 6 concludes the paper with directions for future work.

2 Materials and Methods

2.1 Materials

Subsequent paragraphs, however, are indented. This section reviews the literature on query Generation software, focusing on three main aspects: (1) the motivation and challenges of Query Generation, (2) the methods and techniques for generating and optimizing SQL queries from natural language prompts, and (3) the approaches and challenges for integrating query Generation software with cloud-based data services. These aspects are relevant and essential for our paper, as we aim to propose a novel query Generation software, named Querier, that utilizes natural language processing

techniques and reinforcement learning to generate SQL queries from user prompts and seamlessly integrates with Google BigQuery for query execution.

Methods and Techniques for Generating SQL Queries from Natural Language Prompts. Previous research has explored various methods for generating structured queries from natural language, evolving from rule-based systems to sophisticated deep learning models. Recent advancements, particularly those leveraging Large Language Models (LLMs), have significantly improved the accuracy and complexity of text-to-SQL translation.

A notable approach is **DIN-SQL** [8], which employs a "Decomposed In-Context Learning" strategy. Instead of tackling a complex text-to-SQL problem in one step, it breaks the problem down into smaller, more manageable sub-problems. This method uses an LLM with minimal examples to guide it through a multi-step process, which includes schema linking, query classification, and decomposition, before generating the final SQL query. It also incorporates a self-correction mechanism to refine the generated query, enhancing its accuracy.

Another advanced system, **MAC-SQL** [17], is designed to handle multi-turn interactions, where users can refine their queries through conversation. It maintains the context of the dialogue by classifying the relationship of a new question to the previous ones. MAC-SQL then selectively incorporates historical information and schemas to generate the appropriate SQL query, making it effective for complex, iterative data exploration.

CodeS [18], on the other hand, is a text-to-SQL model focused on supervised fine-tuning of large pre-trained language models like T5. It was trained on the Spider and other text-to-SQL datasets, achieving high execution accuracy without relying on in-context learning. Its performance demonstrates the power of fine-tuning LLMs on specific tasks to achieve state-of-the-art results.

More recent developments include systems that integrate directly with database operations. **DBCopilot** [19] utilizes an LLM to generate not only SQL queries but also associated Python code for data visualization. It operates within a closed-loop environment that retrieves database schema, generates queries, executes them, and even performs self-correction by analyzing execution results and applying feedback for subsequent query generation.

Finally, addressing safety and correctness, **SafeSQL** [20] introduces a multi-step generation process that focuses on producing reliable and executable queries. It uses an LLM to first select the relevant tables and then generate the query column-by-column, performing validation at each step. This methodical approach is designed to reduce errors and ensure that the final query is both syntactically correct and semantically aligned with the user's intent. The previous works use different combinations of methods and techniques to generate and optimize SQL queries from natural language prompts, as shown in Table 1.

Table 1. Comparison of Methodologies from 4 Different Papers

Paper	NLP	CTE	Predicate-oriented Clustering
Decomposed In-Context Learning of Text-to-SQL with Self-Correction (DIN-SQL) [8]	Yes	No	No
Multi-Agent Collaborative Framework for Text-to-SQL (MAC-SQL) [17]	Yes	No	No
CodeS [18]	Yes	No	No
DBCopilot [19]	Yes	No	No
Self-Augmentation in-context learning with Fine-grained Example selection for Text-to-SQL (SafeSQL) [20]	Yes	No	No

Approaches and Challenges for Integrating Query Generation Software with Cloud-based Data Services. While the integration of query generation software with cloud-based data services like Google BigQuery offers significant advantages such as the scalability to meet growing data demands [21] and access to diverse datasets for more insightful analysis [22] it also introduces a unique set of challenges. These include ensuring interoperability between the generation software and various cloud protocols [23], addressing stringent security and privacy requirements through mechanisms like encryption and authentication [24], and validating the reliability of the generated results [25].

Many existing studies on query generation [18], [19], [26] do not fully address these end-to-end integration challenges, often because their systems are designed for privately accessed databases in specific research or clinical contexts. This leaves a gap for a solution that can tackle the complexities of a modern cloud environment. Addressing these challenges requires a technology capable of handling both linguistic ambiguity and complex system integration, a role for which Large Language Models (LLMs) are exceptionally well-suited[27], [28].

Generative Pre-trained Transformers (GPT) are a class of LLMs that have demonstrated powerful capabilities in various language and code-generation tasks. These capabilities were famously demonstrated by GPT-3 [29], and successor models like GPT-3.5 are further optimized to understand context and generate human-like text or, in this case, structured code. In this approach, the database schema (metadata) is programmatically provided to the model as context. This is followed by the user's natural language prompt, which serves as the specific task instruction. The LLM then synthesizes this information to generate a syntactically correct SQL query that is tailored to the specific

database structure. This technique forms the core of the methodology used in this research to create an integrated, end-to-end system for cloud databases.

2.2 Methods

This section details the design of Querier, a system developed to translate natural language prompts into executable SQL queries. The system's architecture, illustrated in Fig. 1, is a three-part application built with Python. It utilizes Streamlit for the user interface, the OpenAI API for the core query generation engine, and Google BigQuery as the target database. For testing, we used public dataset available inside Google BigQuery titled "wine_quality". It consists of 13 features and 1143 rows without any missing values detected.

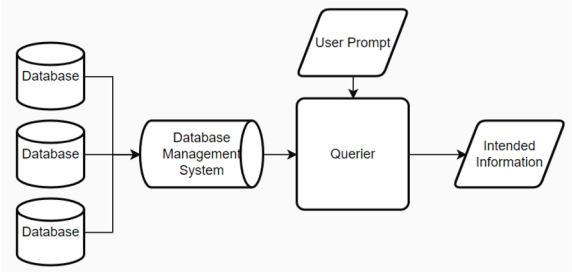

Fig. 1. Querier Architecture

The operational workflow is initiated when a user submits a natural language prompt through the Streamlit UI. To provide context for query generation, the backend engine first retrieves the database schema from BigQuery. This schema is then combined with the user's request using dynamic prompt engineering to construct a detailed prompt for the GPT-3.5-turbo model. The model generates a corresponding SQL query, which is extracted from the API response. After user validation, this query is executed directly on the BigQuery warehouse, and the final results are passed back to the frontend for display.

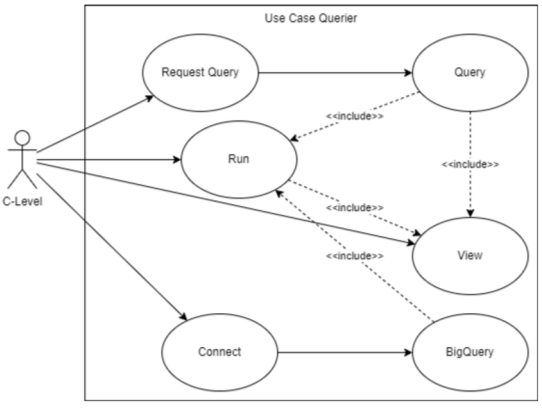

Fig. 2. Querier Use Case Diagram

The primary user interactions are illustrated in the Use Case Diagram (Fig. 2). This diagram depicts the main actor, the User, and their key capabilities within the system, which include submitting a natural language prompt, validating the system-generated SQL query, and initiating its execution against the database.

The Object Diagram in Fig. 3 presents the static relationships between the core software components. It shows the StreamlitUI object responsible for handling user input/output, the TextGenerationAI object that encapsulates the API call, and the main Querier object that orchestrates the workflow between the UI and the BigQuery database object. This query is then passed to the main Querier object, which orchestrates the workflow. After receiving confirmation from the user via the StreamlitUI, the Querier object sends the SQL query to the BigQuery object for execution. The resulting data is returned through the Querier object and ultimately displayed by the StreamlitUI.

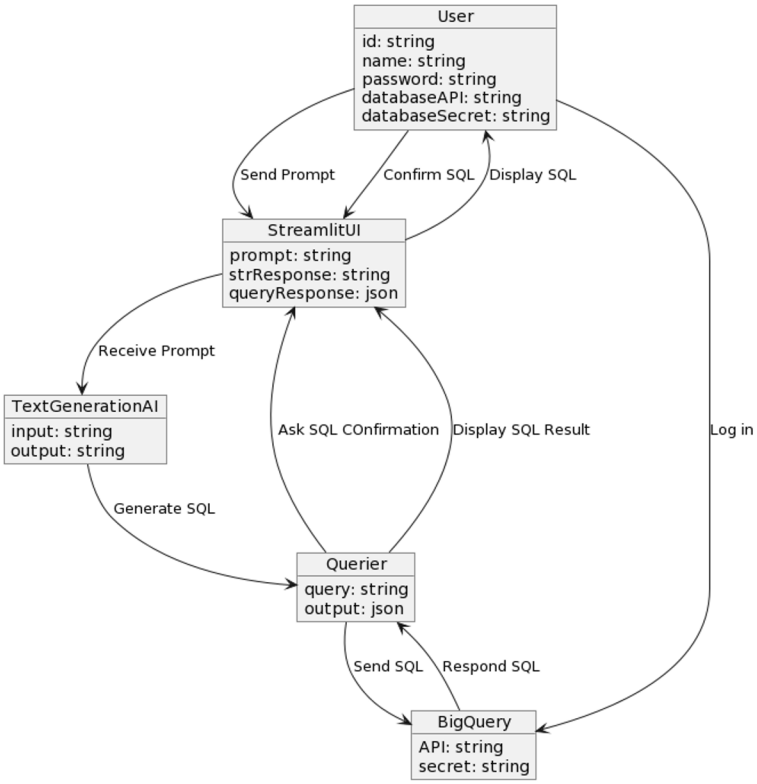

Fig. 3 Querier Object Diagram

The sequence of interactions between the system's four primary components the User, the Querier application, Google BigQuery, and the OpenAI server is detailed in Fig. 4. This diagram shows the chronological flow of messages. Querier functions as the central controller, mediating interactions between the user and the external services.

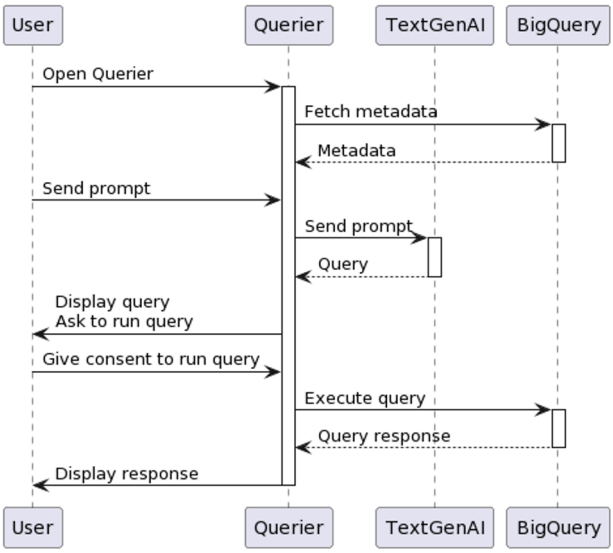

Fig. 4. Querier Sequence Diagram

Two main interaction sequences occur between Querier and Google BigQuery. The first is the initial fetching of database metadata, which is triggered automatically upon application startup. This metadata is then cached within Querier to be used as context for the OpenAI API. The second interaction is the execution of the SQL query, which is initiated only after the user approves the query generated by the LLM. The interaction between Querier and the OpenAI service (represented as TextGenAI) is singular in purpose: to generate the SQL query. A prompt, composed of the database context and the user's request, is sent to the service, and the generated query is returned in the response.

3 Results

3.1 User Scenario Test

The User Scenario Test aims to emulate real-world user interactions. The tests were performed on a laptop with Python libraries and dependencies to validate the application's usability under standard user scenarios. Two tests are conducted in this section, shown in Table 2, mainly to check the application's usability. The output from these tests shows that the application worked properly

Table 2. User Scenario Test Result

No	Test Case ID	Description	Output
1	US-QG-1	Users can enter a prompt into the app	query successfully generated and shown in the chat box

2	US-QG-2	Users can choose whether to run the query or not	Both Run and No option works well

Fig. 5 shows the graphical user interface (GUI) of Querier during a typical operational sequence. A prompt submitted by the user (red profile icon) results in a generated SQL query displayed by the assistant (yellow profile icon). Following user approval for execution, the query result is rendered at the top of the chat interface.

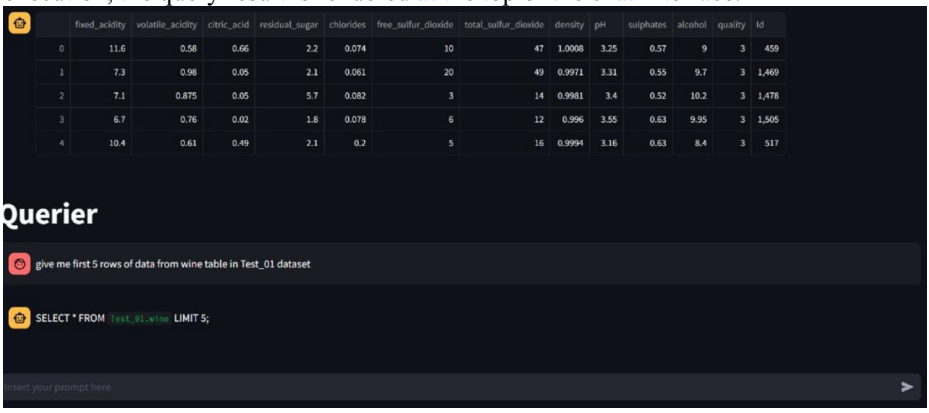

Fig. 5. Querier working GUI

A second set of tests was conducted to specifically validate individual components of the user interface. These tests, detailed in Table 3, assessed the functionality of the prompt input field, the query display area, and the query execution button. The observed outputs for each test case confirm that all tested interface elements respond correctly to user interaction.

Table 3. Interface Testing Result

No	Test Case ID	Description	Output
1	IT-1	Test Prompt input.	The prompt successfully printed on the User's chat bubble inside the chat interaction box.
2	IT-2	Test Display of Query Generation Result	After the user's input is displayed, the assistant's chat bubble successfully appears below the user's chat bubble with printed query results.
3	IT-3	Test Run Query Button	After choosing the 'Run' button, the user receives a query execution result.

3.2 Component-Level Testing

In addition to the end-to-end functional validation, tests were conducted to verify the core backend components responsible for query generation and database communication. The primary functions of the query generation engine were tested, including metadata retrieval from the database, the parsing of user prompts, and the handling of the OpenAI API call. The test cases and outcomes, confirming the successful operation of each component, are detailed in Table 4.

Table 4. Query Generation Testing Result

No	Test Case ID	Description	Output
1	QG-1	Read metadata from the database	The metadata of the database was successfully fetched.
2	QG-2	Read the user prompt and parse it to fit the true prompt	The prompt result is successfully parsed into a query.
3	QG-3	Send the parsed prompt to the API	API Response shows that the query is successfully executed.

The results in Table 4 confirm that each component of the query generation engine, from metadata retrieval to API communication, operated successfully without errors. The tests specifically verified the system's capability to correctly fetch the database schema, properly structure the user's prompt for the API call, and receive a valid SQL query in response. This successful sequence validates that the foundational logic for translating a user's intent into an executable query is functioning as designed.

Furthermore, the connectivity between the Querier application and the Google BigQuery database was specifically tested. These tests validated the system's ability to establish a connection, execute a generated query against the database, and receive the resulting data.

Table 5. Database Connection Testing Result

No	Test Case ID	Description	Output
1	DC-1	Connect to Database	Connected and retrieved database metadata
2	DC-2	Send generated query into BigQuery	Query successfully executed

3	DC-3	Receive results from an executed query	Successfully received query execution result

As shown in Table 5, all stages of database interaction were completed successfully, confirming the system's robust connectivity with Google BigQuery. The tests validated not only the initial handshake to establish a connection but also the subsequent steps of sending a generated SQL query for execution and correctly receiving the resulting data set. This successful outcome demonstrates that the data communication pipeline, which is critical for the application's core function, is reliable and operates without issue.

To assess the operational efficiency of the system, a single-user performance test was conducted in a controlled environment. The evaluation measured the time required for the system to process a range of prompts with varying levels of complexity, from simple data retrieval to queries involving multiple aggregations. The five distinct prompts used for this evaluation are detailed in Table 6

Table 6. Prompts used in the User Performance Test

No	Context	Prompt
1	Simple Prompt	give me the first five rows of data from the wine table in the Test_01 dataset
2	Prompt with Filter	show me the first ten customer surnames from the 'Test_01.customer_churn' table in the Test_01 dataset that have complained
3	Prompt with 1 Aggregation	count the frequency of active customers from the 'Test_01.customer_churn' table in the Test_01 dataset
4	Prompt with 3 Aggregation	find the mean, min, and max of points earned by all customers in the 'Test_01.customer_churn' table in the Test_01 dataset and return only the query for the Google BigQuery database
5	Prompt with Operand	Find the top 3 wines with the highest acidity by adding fixed and volatile acidity from the Test_01.wine table in the Test_01 dataset

Each of the five prompts was executed twice to calculate an average processing time for different stages of the workflow. The results of this performance test are presented in Table 7. The measured stages include the time for the OpenAI API to generate a query, the time for Google BigQuery to execute that query, and the time required to load the final result into the user interface.

Table 7. User Performance Test Result per Prompts

No	Context	Usability Case				
		1	2	3	4	5
1	Simple Prompt	≈0.0s	1.49s	2	1.35s	1.35s
2	Prompt with Filter	≈0.0s	2.15s	2	1.39s	1.40s
3	Prompt with 1 Aggregation	≈0.0s	2.23s	2	1.47s	1.57s
4	Prompt with 3 Aggregation	≈0.0s	3.67s	2	1.39s	1.39s
5	Prompt with Operand	≈0.0s	3.08s	2	1.28s	1.28s
		≈0.0s	2.52s	10	1.38s	1.39s

The performance data presented in Table 7 reveals several key insights into the system's operational efficiency. A direct correlation is observed between the complexity of the user's natural language prompt and the API Latency required for query generation. For instance, the 'Simple Prompt' was processed by the API in just 1.49 seconds, whereas the significantly more complex 'Prompt with 3 Aggregations' required 3.67 seconds, more than double the time. Conversely, the Query Execution Time within Google BigQuery showed remarkable consistency across all tests, remaining stable around the average of 1.38 seconds, regardless of the query's complexity.

To provide a more generalized overview of the system's performance, a summary of key metrics is presented in Table 8. For metrics involving network communication, such as API latency and query execution time, the results have been normalized by the measured internet speed of the local test environment. This normalization, expressed in seconds per megabit per second (s/MBps), was performed to account for connection speed as a significant external factor and to offer a more standardized performance baseline.

Table 8. Usability Test Metric Result

No	Context	Usability Case	Score
1	Query Generation	Time to parse the prompt	≈0.0s
2	Query Generation	Time to send a prompt and receive query based on Internet speed (per 1 MBps)	0.41s / 1 MBps

3	Query Generation	A successful run of the query out of 10 trials	10 out of 10
4	Query Execution	Successful query execution based on internet speed (per 1 MBps)	0.22s / 1 MBps
5	Query Execution	Time to load/view table from Google BigQuery Database	1.39s

Table 8 aggregates the performance data into a concise set of benchmarks that define the system's capabilities under the tested conditions. Notably, the system achieved a 100% success rate across all 10 trials, indicating high reliability for the core functionalities. The summary highlights an average API latency of 2.52 seconds and an average database execution time of 1.38 seconds. The normalized figures, such as the API latency of 0.41 s/MBps, provide a useful metric for estimating performance expectations under varying network conditions.

4 Results

The evaluation confirmed that all test prompts were successfully translated into executable queries and highlighted that API latency, rather than database execution time, is the primary performance variable. A particularly significant finding was the distinction between API latency and execution time. This suggests that for systems of this nature, optimization efforts should prioritize the AI interaction layer such as advanced prompt engineering or model selection rather than traditional database tuning. The system's overall speed, delivering results in under ten seconds for all test cases, further confirms its suitability for interactive data analysis, where user responsiveness is a critical factor.

When contextualized within the existing literature, these results are noteworthy. Compared to the trending query generators from Table 1, Querier offers superior flexibility in handling a diverse range of natural language instructions. Unlike highly specialized systems like DIN-SQL [8], MAC-SQL [17], and CodeS [18], Querier showcases the power of modern LLMs to achieve strong results in a more general-purpose context, relying on comprehensive schema context rather than deep, domain-specific programming. Beyond these academic contributions, the practical implications of this work are substantial. By effectively lowering the technical barrier to database interaction, Querier serves as a functional model for tools that can democratize data access within an organization, enabling domain experts without SQL skills to conduct self-service analytics and accelerate decision-making. Despite these promising results, it is important to acknowledge several limitations. The evaluation relied on a limited set of custom prompts and did not utilize standardized benchmarks like FIBEN [30], making direct performance comparisons challenging. Furthermore, testing was confined to a

single-user environment and a single database platform (Google BigQuery), so its performance under concurrent loads and its adaptability to other database technologies remain unevaluated.

5 Conclusion

This research was motivated by the need to simplify database interaction for users with limited SQL proficiency. To address this, the Querier system was designed and developed as an end-to-end solution that translates natural language prompts into executable SQL queries using a Large Language Model. The comprehensive evaluation conducted in this study confirmed the viability and effectiveness of this approach, demonstrating that Querier is both functionally reliable and performant.

The key findings indicate that the system successfully handles all tested user scenarios and that its core components operate as designed. Performance evaluations revealed that Querier can process complex requests efficiently, with the primary performance variable being the AI's inference latency rather than the database execution time. The primary contribution of this work is a validated proof-of-concept that demonstrates how modern LLMs can be securely and effectively integrated with cloud databases to democratize data access, providing a practical blueprint for similar applications.

A key limitation of the present study is the use of custom tests rather than standardized benchmarks, which makes direct performance comparison with other systems challenging. Therefore, a critical direction for future work is to evaluate Querier against established benchmarks, such as SPLODGE for query execution performance or the FIBEN dataset for natural language to SQL accuracy. Furthermore, future research should expand the scope of testing to include a wider variety of ambiguous prompts, assess performance under concurrent user loads, and extend support to other database systems to enhance the system's versatility and robustness.

Acknowledgments. The authors would like to express their sincere gratitude to their supervisor, Prof. Riri Fitri Sari, for her invaluable guidance, support, and supervision throughout this research project. This research is supported in part by Kemdiktisaintek under WCU-Equity Grant with Grant Number PKS-36/UN2.R3/HKP.05.00/2025

Disclosure of Interests. The authors declare no conflict of interest regarding the publication of this paper. All authors have seen and approved the final manuscript.

References

1. A. Abdelkawi, H. Zafar, M. Maleshkova, and J. Lehmann, “Complex Query Augmentation for Question Answering over Knowledge Graphs,” in *On the Move to Meaningful Internet Systems: OTM 2019 Conferences*, vol. 11877, H. Panetto, C. Debruyne, M. Hepp, D. Lewis, C. A. Ardagna, and R. Meersman, Eds., in Lecture Notes in Computer Science, vol. 11877. , Cham: Springer International Publishing, 2019, pp. 571–587. doi: 10.1007/978-3-030-33246-4_36.

2. "Database System Concepts - 7th edition." Accessed: Aug. 08, 2025. [Online]. Available: https://db-book.com/
3. A. Neema, "Transformation in Data Storing Technique- Big Data: A Literature Review," *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. 9, no. 9, pp. 752–762, Sep. 2021, doi: 10.22214/ijraset.2021.38038.
4. M. Poess, T. Rabl, and H.-A. Jacobsen, "Analysis of TPC-DS: the first standard benchmark for SQL-based big data systems," in *Proceedings of the 2017 Symposium on Cloud Computing*, Santa Clara California: ACM, Sep. 2017, pp. 573–585. doi: 10.1145/3127479.3128603.
5. W. Ali, M. U. Shafique, M. A. Majeed, and A. Raza, "Comparison between SQL and NoSQL Databases and Their Relationship with Big Data Analytics," *Asian J. Res. Comput. Sci.*, pp. 1–10, Oct. 2019, doi: 10.9734/ajrcos/2019/v4i230108.
6. D. C. Jamison, "Structured Query Language (SQL) Fundamentals," *Curr. Protoc. Bioinforma.*, vol. 00, no. 1, Jan. 2003, doi: 10.1002/0471250953.bi0902s00.
7. J. Peng and K. Han, "Survey of Pre-trained Models for Natural Language Processing," in *2021 International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB)*, Dec. 2021, pp. 277–280. doi: 10.1109/ICEIB53692.2021.9686420.
8. M. Pourreza and D. Rafiei, "DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction," 2023, *arXiv*. doi: 10.48550/ARXIV.2304.11015.
9. J. J. Cuadrado-Gallego and Y. Demchenko, *Data Analytics: A Theoretical and Practical View from the EDISON Project*. Cham: Springer International Publishing, 2023. doi: 10.1007/978-3-031-39129-3.
10. A. Bhaskar, T. Tomar, A. Sathe, and S. Sarawagi, "Benchmarking and Improving Text-to-SQL Generation under Ambiguity," Oct. 20, 2023, *arXiv*: arXiv:2310.13659. doi: 10.48550/arXiv.2310.13659.
11. R. V. Alvarez, L. Mariño-Ramírez, and D. Landsman, "Transcriptome annotation in the cloud: complexity, best practices, and cost," *GigaScience*, vol. 10, no. 2, p. giaa163, Jan. 2021, doi: 10.1093/gigascience/giaa163.
12. H. Cunningham, "A definition and short history of Language Engineering," *Nat. Lang. Eng.*, vol. 5, no. 1, pp. 1–16, Mar. 1999, doi: 10.1017/S1351324999002144.
13. D. Jurafsky and J. H. Martin, Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2. ed. [Nachdr.]. Upper Saddle River, NJ: Prentice Hall, 2009.
14. S. Barke, M. B. James, and N. Polikarpova, "Grounded Copilot: How Programmers Interact with Code-Generating Models," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, pp. 85–111, Apr. 2023, doi: 10.1145/3586030.
15. A. A. Sawant and P. Devanbu, "Naturally!: How Breakthroughs in Natural Language Processing Can Dramatically Help Developers," *IEEE Softw.*, vol. 38, no. 5, pp. 118–123, Sep. 2021, doi: 10.1109/MS.2021.3086338.
16. "Natural language interfaces to databases | The Knowledge Engineering Review | Cambridge Core." Accessed: Aug. 08, 2025. [Online]. Available: https://www.cambridge.org/core/journals/knowledge-engineering-review/article/abs/natural-language-interfaces-to-databases/955216384EDF110EE89CB401C4A8C555
17. B. Wang *et al*, "MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL," Mar. 18, 2025, *arXiv:* arXiv:2312.11242v6, doi: 10.48550/arXiv.2312.11242.
18. H. Li *et al.*, "CodeS: Towards Building Open-source Language Models for Text-to-SQL," Feb. 26, 2024, *arXiv:* arXiv:2402.16347, doi: 10.48550/arXiv.2402.16347.
19. T. Wang *et al*, " DBCopilot: Natural Language Querying over Massive Databases via Schema Routing", Feb. 22, 2025, *arXiv:* arXiv:2312.03463, doi: 10.48550/arXiv.2312.03463.

20. J. Lee *et al*, " SAFE-SQL: Self-Augmented In-Context Learning with Fine-grained Example Selection for Text-to-SQL", May 22, 2025, *arXiv:* arXiv:2502.11438v2, doi: 10.48550/arXiv.2502.11438.
21. W. Al Shehri, "Cloud Database Database as a Service," *Int. J. Database Manag. Syst.*, vol. 5, no. 2, pp. 1–12, Apr. 2013, doi: 10.5121/ijdms.2013.5201.
22. R. Gyorodi, M. I. Pavel, C. Gyorodi, and D. Zmaranda, "Performance of OnPrem Versus Azure SQL Server: A Case Study," *IEEE Access*, vol. 7, pp. 15894–15902, 2019, doi: 10.1109/ACCESS.2019.2893333.
23. S. Pal, "SaaS Level based Middleware Database Integrator Platform," *Int. J. Adv. Comput. Sci. Appl.*, vol. 8, no. 5, 2017, doi: 10.14569/IJACSA.2017.080552.
24. D. Agrawal and A. E. Abbadi, "A Wake-up Call: Managing Data in an Untrusted World".
25. W. Lan *et al.*, "UNITE: A Unified Benchmark for Text-to-SQL Evaluation," Jul. 14, 2023, *arXiv*: arXiv:2305.16265. doi: 10.48550/arXiv.2305.16265.
26. X. Dong *et al.*, "C3: Zero-shot Text-to-SQL with ChatGPT," Jul. 14, 2023, *arXiv*: arXiv:2307.07306. doi: 10.48550/arXiv.2307.07306.
27. M. Chen *et al.*, "Evaluating Large Language Models Trained on Code," Jul. 14, 2021, *arXiv*: arXiv:2107.03374. doi: 10.48550/arXiv.2107.03374.
28. X. L. Li and P. Liang, "Prefix-Tuning: Optimizing Continuous Prompts for Generation," Jan. 01, 2021, *arXiv*: arXiv:2101.00190. doi: 10.48550/arXiv.2101.00190.
29. T. Brown *et al.*, "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2020, pp. 1877–1901. Accessed: Aug. 08, 2025. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html
30. J. Sen *et al.*, "ATHENA++: natural language querying for complex nested SQL queries," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 2747–2759, Aug. 2020, doi: 10.14778/3407790.3407858.