International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Enhancing Database Accessibility: A Llama-2 Powered Interface for Intelligent Database Querying

Neelam Pawar[1], Isha Waghulde[2], Soham Nimbalkar[3], Prathamesh Suryawanshi[4], Manish Sharma[5]

[1,2,3,4]Student, Artificial Intelligence & Data Science , Dy Patil College Of Engineering
[5]Guide, Artificial Intelligence & Data Science , Dy Patil College Of Engineering

Abstract
This research work introduces an AI-powered Natural Language to SQL (NL2SQL) interface to make database querying easy for end-users. Our system, which uses a fine-tuned LLaMA-2 model, generates SQL queries dynamically, with better adaptability and accuracy than rule-based systems. Based on a curated dataset from Hugging Face's synthetic text-to-Sql corpus, our system maintains robust performance on various query structures. Implemented using Python, PyTorch, and Flask, and deployed using Gunicorn and Tensor Dock, the system exhibits high accuracy in converting natural language to executable SQL. Experimental results demonstrate improvements in query precision, execution efficiency, and deployment scalability. Future development will aim to improve contextual comprehension and real-time query performance.

Keywords: Natural Language to SQL (NL2SQL), LLaMA-2, Database Querying, AI-driven SQL Generation, Deep Learning, Text-to-SQL, Query Optimization

Introduction

In the data-driven world of today, it is still a major challenge to access and interpret structured data, especially for non-technical users who are not familiar with database query languages such as SQL. Although databases are the foundation of decision-making in all industries, the intricacy of Structured Query Language (SQL) acts as a hindrance for most people who want to gain valuable insights from data. [1] . To fill this void, we introduce an Artificial Intelligence (AI) Natural Language to SQL (NL2SQL) interface that allows users to access databases through easy, natural language queries. Our system utilizes Large Language Models (LLMs), namely a fine-tuned LLaMA-2, to process user intent and automatically create correct SQL queries. In contrast to conventional strategies based on predefined templates or rule-based conversions, our deep learning-based model becomes flexible to support multiple query forms, enhancing usability. [2], [5]

The power to view and process structured data is vital for decision-making in various industries. Nevertheless, working with relational databases usually necessitates knowledge of Structured Query Language (SQL), which represents a significant hindrance for non-technical users like business analysts, researchers, and domain experts. Although current databases can deal with large and complex datasets,

their usefulness is hampered by the technical obstacle of SQL. This intricacy tends to deter useful data insights from being maximally used by a larger population. To address this issue, our project proposes a state-of-the-art AI-based NL2SQL system that allows users to query databases with easy natural language queries. By integrating natural language processing (NLP) with deep learning, we convert loose or imprecise natural language into accurate SQL queries. The core of our system is a fine-tuned LLaMA-2 model, optimized with LoRA and QLoRA for memory-aware learning, and trained over an optimized dataset for varied SQL tasks.

Our proposed work relies on a well-filtered dataset obtained from Hugging Face, with the initial pool being 100,000 text-to-SQL pairs. Using data augmentation strategies like paraphrasing, we narrowed down the dataset to 41,000 high-quality query mappings. The back-end is supported using Python, PyTorch, and Hugging Face Transformers, with Flask and SQLAlchemy ensuring smooth database operations. For production deployment, we make use of Gunicorn for web server management and Tensor Dock for scalable GPU hosting.

This research will democratize data access by enabling business analysts, decision-makers, and researchers to access complicated information without having any SQL expertise beforehand. By fusing Natural Language Processing (NLP) and deep learning, we make the database more accessible and usable and enable data-driven decision-making for everyone.

Literature Survey

Design of Natural Language to SQL (NL2SQL) tools has remained an active line of research during recent years in view of mounting pressures to simplify and make available the structured database in an understandable, user-friendly fashion for lay persons. Since domains are gradually growing data-dependent, being in a position to discern valuable facts and insights out of relational databases with no specific training in Structured Query Language (SQL) becomes vital. But the syntactic complexity of SQL and the rigid schema of database structures pose a considerable obstacle for non-programming users. Efforts to overcome this have seen researchers pursue a number of methods for translating natural language questions into SQL queries. Initial methods employed rule-based approaches, whereby hand-coded syntax rules and templates were invoked to process user input and generate SQL queries. Successful in constrained settings, such methods did not scale, were not flexible, and were inefficient at handling differing query structures.

With the advancement of machine learning (ML) and deep learning, particularly Large Language Models (LLMs) like GPT, BERT, and LLaMA-2, impressive developments have been made in NL2SQL systems to facilitate more flexible and dynamic query generation. NL2SQL systems have progressed from initial rule-based structures to current deep learning frameworks, greatly increasing database access for users who are not technical. Initially, rule-based approaches utilizing hand-coded syntactic rules and pre-defined SQL templates were utilized by researchers. A typical instance can be found in [1], where authors utilized the Natural Language Toolkit (NLTK) to analyze user input and translate it into SQL queries. While such systems worked fine in controlled settings, they were inflexible to accommodate various query formulations and did not generalize beyond pre-defined templates.

To overcome these challenges, machine learning (ML) solutions were proposed. In [3], supervised learning techniques were utilized to acquire mappings from natural language queries to SQL queries. This method enhanced flexibility and minimized the requirement of rule engineering by hand. Nevertheless, the models needed large amounts of labeled training data and still had difficulty

generalizing across various schemas. Moreover, dealing with sophisticated queries with aggregations, joins, and nested structures was still a challenge. With the advent of transformer-based architectures and pretraining at scale, the performance of NL2SQL systems was greatly enhanced. As demonstrated in [4], Large Language Models (LLMs) like BERT, T5, and LLaMA-2 attained significant success in producing correct SQL queries by learning context-sensitive natural language to structured output mappings. Fine-tuning these models on dataset-specific splits greatly enhanced their performance. Yet, as observed in [5], fine-tuning poses computational cost and domain adaptation challenges. Robust performance depends on dataset diversity and quality.

The computational requirement of LLMs is another significant hurdle. The authors in [6] investigated efficient fine-tuning methods like parameter offloading, gradient checkpointing, and quantization. Such techniques allow for model training and inference on commodity hard-ware, making LLM-based systems more accessible to the masses. Still, such approaches compromise model performance at the cost of lower hardware specifications slightly. From a systems viewpoint, optimization of architecture is essential for real-time use. In [7], the authors introduced a scalable natural language query processing system that utilizes schema-aware indexing and query caching to minimize latency. Such enhancements increase user experience but necessitate system resource management and user intent disambiguation mechanisms to ensure relevance of queries.

At a larger level [8] , outlines a vision of how NL2SQL interfaces make data accessible to everyone by allowing non-technical users to access structured information through natural language. Such systems have immense potential for industries such as healthcare, education, and finance. However, their success depends on the model's capacity to produce secure, executable, and semantically correct queries. Additionally, there are ongoing research efforts on zero-shot and few-shot learning methods to enhance model flexibility with minimum fine-tuning [10].

A thorough overview in [11] reviews LLM-based Text-to-SQL systems, the ways in which sequence-to-sequence models, schema linking, and benchmark data such as Spider and WikiSQL have influenced the present state of development. The overview identifies existing problems like handling vague queries, generalization over domains, and generating logically accurate queries consistent with user intent. Machine learning methods brought forth supervised learning to enhance query flexibility. Statistical techniques were employed in [3] to translate natural language into SQL. This was more flexible but needed large annotated data sets and still had trouble with unknown database schemas and multi-table queries.

The use of transformer-based architectures represented a paradigm shift in NL2SQL. As illustrated in [4], Large Language Models (LLMs) such as BERT, T5, and LLaMA-2 showed excellent performance in producing syntactically and semantically accurate SQL. These models learned contextual mappings and outperformed previous methods significantly. Fine-tuning such models, however, presents challenges regarding domain adaptation and computational expense, as explained in [5]. Performance is highly reliant on dataset diversity and model generalization abilities.

NL2SQL systems have progressed a lot since then from rule-based to advanced deep learning models. Early systems employed hand-crafted templates and context-free grammars (CFGs) to map user queries to formal SQL statements. While effective for straightforward queries, these methods struggled with complex SQL structures such as nested queries, joins, and aggregations. To ad-dress these limitations, machine learning (ML) models were suggested by researchers that enabled query translation automatically through learning patterns from labeled datasets. Sequence-to-sequence models based on

recur-rent neural networks (RNNs) and transformers provided improvements in natural language query comprehension and dynamic SQL query generation.

In Summary, NL2SQL systems have therefore evolved from inflexible, rule-based systems to extremely flexible and precise LLM-based systems. Rule-based systems were not scalable, ML-based models learned better but did not have deep contextual knowledge, and transformer-based LLMs deliver state-of-the-art results with compromises in computation and resource utilization. Future work needs to tackle issues like model efficiency, real-time query generation, better schema generalization, and better semantic understanding to make NL2SQL systems practical, secure, and widely deployable.

Methodology

A. System Overview

The overview of our proposed system presents the design, functionality, and importance of the system pro-posed for transforming natural language queries into SQL queries with the help of large language models (LLMs). Our proposed system is designed to fill the gap be-tween users who do not possess technical skills in SQL and the requirement of accessing structured data from relational databases effectively. Historically, approaches to data querying involved strong proficiency with SQL syntax, database structures, and query optimization, acts which might impede non-technical users from making use of querying systems. Our proposed system solves the difficulty by means of natural language processing and deep learning that helps map human-query expressions to SQL queries that could be executed on the machine.

B. System Architecture

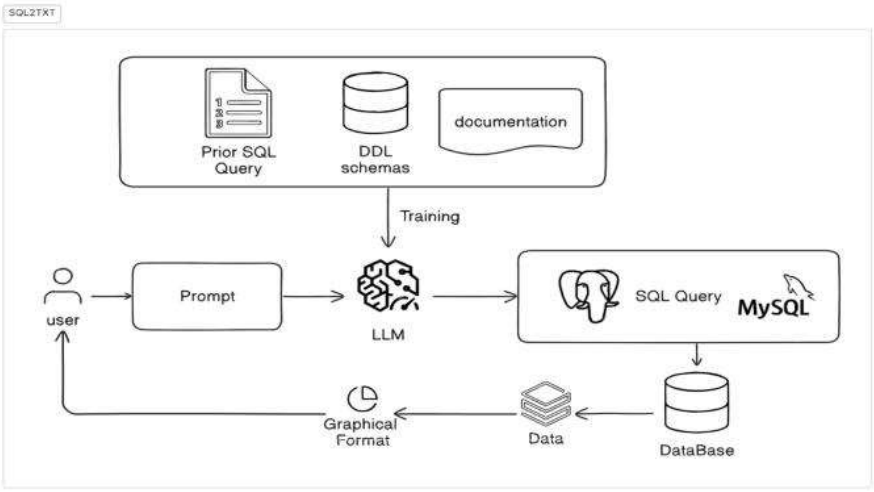

Fig. 1. System Architecture

C. Input Processing

Input processing is an essential part of our suggested system that guarantees natural language queries are properly interpreted and prepared for precise SQL query generation. Input processing entails several steps such as handling the user query, schema extraction, context generation, and prompt generation for the large language model (LLM). Our proposed system uses the Clinton dataset, which is a set of structured database schemas along with natural language queries and their equivalent SQL queries. The dataset gives us a large set of examples where every entry includes an instruction (natural language

query), a table schema, and the resulting SQL output. Through the use of this dataset, our system enhances its capability to comprehend many different ways through which users will query databases.

User Query Handling: The processing of input starts with the input from a user of a natural language question. The system can handle various types of questions, such as simple retrieval requests (e.g., "Display all the sales department employees") and complex analytical questions (e.g., "What is the average salary for employees recruited during the last five years?"). Due to the inherent nature of human language as flexible, the system is required to normalize and preprocess the input for standardization in query interpretation.

Ambiguity resolution is one of the major challenges of handling user queries. The user could pose the same query differently, employ synonyms, or give partial instructions. For example, the question "How many times he finished fourth" must be correctly mapped to the corresponding position column in the table.

To accommodate this, the system applies elementary text preprocessing involving tokenization, stopword filtering, and synonym mapping. Besides, named entity recognition (NER) is also applied to find important entities like names, dates, and places in the query.

Schema Extraction and Context Generation: Upon processing of the user query, the system fetches the pertinent database schema from the Clinton dataset. The schema provides metadata regarding the database table structure, such as column names, data types, and relation-ships. Suppose a user query involves competition results of an athlete. Schema extraction is necessary since it enables the system to realize what columns are applicable for the query and how they must be addressed within SQL statements. The system chooses the proper schema dynamically depending on the context of the query so that it never creates SQL queries that reference non-existent tables or columns.

Prompt Construction for LLM: After processing the user query and schema details, the system builds a structured prompt to be forwarded to the LLM. The prompt is in a standardized format to provide consistency in generating SQL queries. The basic structure of a prompt consists of:

Instruction: Natural language query of the user.

Schema Information: Structured schema of the concerned database.

Example SQL Queries (if necessary): Earlier queries and their SQL equivalents to direct the model.

Expected Output Format: Instructions on how the SQL query must be framed.

For example, if the user query is: "Tell me how many times he came in 4th."

The system frames the following prompt:

Here is an SQL table schema accompanied by an instruction that outlines a task. Based on the provided schema, write an SQL query that completes the instruction.

Schema:

```
CREATE TABLE table_name id NUMBER,
year NUMBER, competition TEXT, venue TEXT, position
TEXT, notes TEXT
);
```

Instruction: Tell me the number of times he placed 4th.

Output: SELECT COUNT(*) FROM table_name WHERE "position" = 4;

This orderly strategy ensures the LLM contains everything it needs to create an acceptable SQL query. The insertion of the schema eliminates hallucinations when the model generates SQL commands with

invalid column names or syntactic errors. Also, inserting past examples of the Clinton set enhances query quality through contextual learning.

Processing Edge Cases in Inputs: As it processes user input, the system has to deal with a number of possible issues:

Spelling Errors & Variations: The system uses spell-check and synonym expansion to correctly match user words with database column names.

Ambiguous Queries: In case a query is not specific enough, the system can ask the user for clarification (e.g., "Did you mean results for all years or a particular year?").

Complex Queries with Aggregation: When queries have computations, the system makes sure the SQL query is accompanied by relevant functions (e.g., COUNT, SUM, AVG).

Missing Data Handling: When a query is based on missing or null values, conditional filtering methods are used by the system to tackle such scenarios nicely.

Optimization for Large-Scale Datasets: Since the Clinton dataset comprises a huge set of records, performance optimization while generating queries is required. Efficiency is maintained through:

Query Simplification: The model tries to reduce unnecessary complexity in SQL statements.

Caching Mechanisms: Often repeated queries are cached to eliminate redundant processing.

D. Model training and fine tuning

Training and fine-tuning for our natural language to SQL generation model, we used LLaMA 2, a high-performance large language model (LLM) specifically tuned for natural language processing. The model was fine-tuned and trained with LoRA (Low-Rank Adaptation) and QLoRA (Quantized LoRA) to strike a balance between efficiency and accuracy, greatly minimizing computational overhead while ensuring strong performance. Our method ensures that the model generalizes well across a wide range of SQL query structures and can adjust to different query formulations. In order to im-prove model training, we have incorporated several upgrades. The use of LoRA and QLoRA facilitated parameter-efficient fine-tuning, which was essential for processing large datasets without high memory usage. Regular model evaluation runs were conducted during training to analyze the balance between model complexity and efficiency. These steps ensured that the model converged well while maintaining optimal performance. To promote usability, we integrated domain adaptation by aligning domain-specific SQL queries, for example, in finance, healthcare, and e-commerce domains, to allow the model to generate accurate queries that fit particular datasets Synthetic data creation was also employed to add to the dataset, so that the model saw a huge variety of SQL patterns. This improved the model's ability to generalize across various use cases and reduced dependence on labeled data.

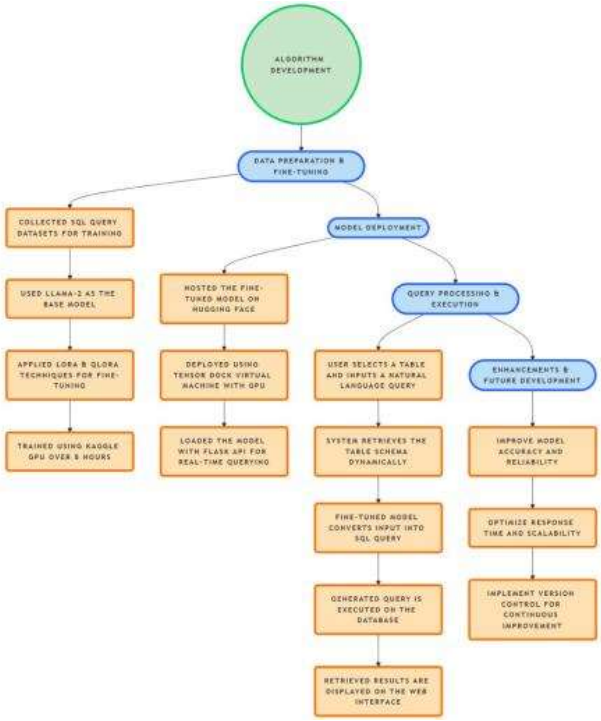

Fig. 2. Algorithm Development

E. Validation and testing

Upon training and fine-tuning the model, the SQL queries generated are of utmost importance in terms of accuracy and correctness. The generated queries should be syntactically correct as well as logically accurate, so as to provide results when run against a real database environment. Verification and execution of the generated SQL queries are distributed across various crucial steps:

- ***Syntactic Validation:*** The initial step of query validation is to check whether the SQL query generated by the model is correctly syntactically formed. This is achieved through parsing the query with an SQL parser or running the query in a mock or real database environment where the syntax violations will be trapped. Any errors, like misnamed tables, missing or incorrectly placed keywords, or incorrect operators, are indicated for additional tuning.
- ***Logical Validation:*** After the query has successfully gone through syntactic checking, the second challenge lies in verifying that it is semantically correct and yields the correct results. Logical checking guarantees that the resultant SQL query is semantically valid, i.e., it accurately mirrors the intention behind the initial natural language query. This process verifies whether the query conforms to the underlying database schema and if it yields meaningful output.
- ***Target Database Execution:*** To complete the verification, automatically created SQL queries need to be run against a simulation or target database to check if they work appropriately against the database schema. This phase checks whether the queries produce the correct data, handle edge cases, and execute adequately within the given database system. The executing is normally done in a testing environment where the queries are run against a testing database that imitates actual operations but with partial data.
- ***Error Correction and Post-Processing:*** At the time of execution, there could be some queries that would contain errors or yield unexpected output. For the resolution of such issues, an error correction process is employed. This facility detects common runtime errors, such as division by zero, op-

erations involving an invalid data type, or unsuccessful joins, and offers corrections or modifications to the query syntax. In addition, post-processing rules are also applied to improve query quality, for example, including suitable indexing hints, JOIN condition optimization, or subquery reordering for better performance.

F. Performance Evaluation

Evaluation Metrics: We have used several performance metrics to give a thorough evaluation of the effectiveness of the natural language to SQL generation model. These metrics were chosen to evaluate the model's correctness, efficiency, and robustness for real-world SQL generation problems.

Execution Accuracy: Measures if the generated SQL query returns the appropriate results when run on a database.

Execution Accuracy (ExecAcc) is defined as:

$$ExecAcc = \frac{Number\ Of\ Queries\ With\ Correct\ Execution\ Results}{Total\ Number\ Of\ Queries}$$

F1 Score: Measures precision and recall trade-off in SQL query generation. The F1 Score is the harmonic mean of Precision and Recall:

$$F1\ Score = 2 \times \frac{Precision\ -\ Recall}{Precision\ +\ Recall}$$

Where:

Precision is the fraction of relevant instances among the retrieved instances as given below:

$$Precision = \frac{True\ Positives}{True\ Positives\ +\ False\ Positives}$$

Recall is the fraction of relevant instances that have been retrieved as given below:

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

Levenshtein Similarity : Measures how close the generated SQL query is to the reference query by computing the minimum number of insertions, deletions, or substitutions that are needed in order to convert one string to another. We achieved an average similarity score of 84% for the top 100 generated queries in our project. The Levenshtein similarity between two strings is derived from the Levenshtein distance:

$$LevenshteinSim(s1, s2) = 1 - \frac{D(s1, s2)}{\max(|s1|, |s2|)}$$

where:

- $D(s_1, s_2)$ is the Levenshtein distance between strings s_1 and s_2,

- $|s_1|$ and $|s_2|$ are the lengths of the respective strings,
- The similarity ranges from 0 (completely different) to 1 (identical).

The Levenshtein distance $D(s_1, s_2)$ is the minimun number of single-character edits (insertions, deletions, or substitutions) required to change s_1 into s_2.

G. Deployment & Scalability

To facilitate efficient and scalable deployment of our LLaMA 2-powered SQL generation model, we used a Flask/FastAPI-based LLM server as the backend. This was necessitated by the requirement for a lightweight, high-performance API framework that can support multiple concurrent requests with low latency. FastAPI was especially beneficial because of its asynchronous request handling, which provided much better response times than conventional synchronous frameworks. The back-end was architected to provide RESTful API endpoints exposing natural language inputs and corresponding SQL statements generated by the fine-tuned model.

Fig. 3. User Interface 1

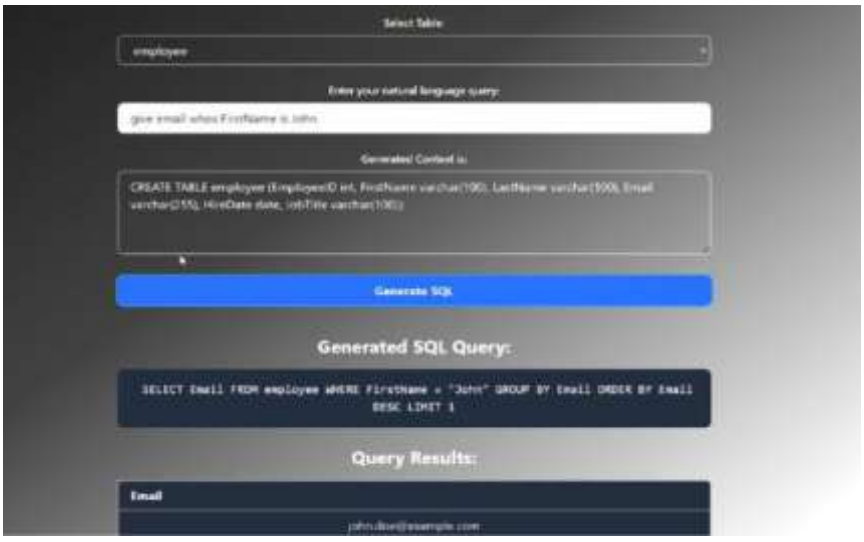

Fig. 4. User Interface 2

For integration with the real world, we made the API database-agnostic so that it can interact with various database management systems (DBMS) like MySQL. The system dynamically maps the generated SQL queries to the underlying database schema, providing compatibility with various relational databases. A query validation module was added to check SQL syntax correctness prior to execution so that incorrect or unsafe queries are not executed. In total, our deployment infrastructure provides low-latency, high-throughput, and scalable SQL generation features that are ideal for practical applications. With containerized deployment, asynchronous computation, caching routines, and high-grade security features, the system remains efficient and dependable even with high-level workloads. Serverless deployment features and additional optimizations for dealing with extremely large-scale query processing situations will be targeted for future enhancements.

Conclusion

In this paper, we have analyzed the work that introduces an Artificial Intelligence(AI) Natural Language to SQL (NL2SQL) interface that illustrates the ways in which powerful deep learning strategies can effectively span the gulf between relational database queries and natural language. By using an off-the-shelf fine-tuned version of LLaMA-2, with sophisticated methods including Reinforcement Learning with Human Feedback (RLHF), schema-based prompt engineering, and multi-task learning, our system can automatically produce accurate SQL queries based on user requests. In contrast to rule-based systems, it is responsive to a variety of query forms, enhancing access for non-expert users.

Utilization of paraphrased data from Hugging Face improved training quality, with system development using Python, PyTorch, and Flask assuring modularity and performance. Compatibility with SQLAlchemy and hosting by Gunicorn and Tensor Dock supported scalability and usability in real-life deployment. The results of the experiment confirmed the accuracy, robustness, and suitability of the model for real-world data-driven decision making applications.

The abstract also identified principal features including LLM integration, schema knowledge, usability, and scala-bility, each of which is illustrated through system design, implementation, and outcomes. To carry on, future work should aim to support advanced queries, boost performance in large databases, and improve generalization. In conclusion, this work makes a valuable contribution to the development of LLM-based database accessibility in analytics and business intelligence.

References

1. S. Alagarsamy, “Natural Language Interface to the Database Man-agement System using NLTK in Python,” *Int. J. Adv. Res. Eng. Technol.*, vol. 16, no. 2, pp. 171–188, Mar. 2025.
2. B.Li, Y. Luo, C. Chai, G. Li, and N. Tang, "The Dawn of Natural Language to SQL: Are We Fully Ready?"
3. N. Author, “Natural Language Query to SQL Conversion Using Machine Learning Approach,” *Conf. Paper*, 2023.
4. J. Smith *et al.*, “Large Language Models Fine-Tuning for Code Generation,” *arXiv preprint*, 2024.
5. R. Wang, “Knowing When to Fine-Tune: Understanding the Trade-offs in LLM Adaptation,” *Neu-rIPS*, 2024.
6. M. Zhang, “Practical Offloading for Fine-Tuning LLMs on Commod-ity Hardware,” *ACL Findings*, 2024.

7. H. Lee *et al.*, "Towards a Natural Language Query Processing System," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 1, pp. 1–13, 2023.
8. P. Kumar, "Revolutionizing Data Accessibility: AI-Driven Natural Language Interfaces," *Data Sci. Rev.*, vol. 12, no. 4, pp. 199–210, 2024.
9. L. Brown, "Fine-Tuning Transformer LLMs for Detecting SQL Injec-tion and XSS Vulnerabilities," *arXiv preprint*, 2024
10. G. Patel, "Fine-Tuning Gemma-2B for Text-to-SQL Generation," *Workshop on Language and Code*, 2024.
11. A. Shah *et al.*, "A Survey of LLM-based Text-to-SQL," *arXiv preprint*, 2024.
12. Hugging Face, "Transformers: State-of-the-art Natural Language Processing for Pytorch, Tensor-Flow, and JAX," [Online]. Available: https://huggingface.co/docs/transformers [Accessed: Apr. 23, 2025].
13. SQLAlchemy, "The Database Toolkit for Python," [Online]. Available: https://www.sqlalchemy.org/ [Accessed: Apr. 23, 2025].
14. Flask, "The Python Microframework for Building Web Applications," [Online]. Available: https://flask.palletsprojects.com/ [Ac-cessed: Apr. 23, 2025].
15. Tensor Dock, "GPU Cloud Platform," [Online]. Available: https: //www.tensordock.com/ [Accessed: Apr. 23, 2025].