From Natural Language to SQL

A Hybrid Approach for Non-Technical Database Querying

(Github)

Chetan Rajendra Rahane (CRR220000)
Kunal Prakash Koshti (KXK240025)
Omkar Suresh Kadam (OXK240000)

Tatiana Erekhinskaya
CS6320 - Spring 25

1. Introduction

In September 2024, Uber released a blog about their Natural language to SQL query generator using GenAI. The general idea was pretty sound, almost entire tech industry heavily relies on databases and most of them rely on SQL databases and queries. Add non-technical professionals using the databases for, say doctors getting patient details, a sales person trying to find what are the customer interests, etc. If there was a website for their data to which they could ask the question of what they want in plain language and get the answers, that would definitely be helpful. For IT professionals, if there was a way to automate the procedure of writing the queries, even if its 50% accurate, it would be helpful. When there was an opportunity to do this on our own, we decided to try building this on our own. So, below, we are explaining our approach, what we went through, what we tried and failed, and much more.

Problem Statement

Modern organizations rely on relational databases to store critical business data, but extracting insights requires SQL expertise – a barrier for non-technical users. Existing solutions face three key limitations:

1. Schema Dependency: Most NL2SQL tools require explicit schema linking (e.g., mapping "patient age" to Patients.Age), limiting their adaptability to new databases.
2. Third-Party Risks: Cloud-based LLM APIs expose sensitive data and incur unpredictable costs.
3. Rigid Templates: Rule-based systems fail to handle linguistic variations like "show records excluding test patients" versus "hide test entries."

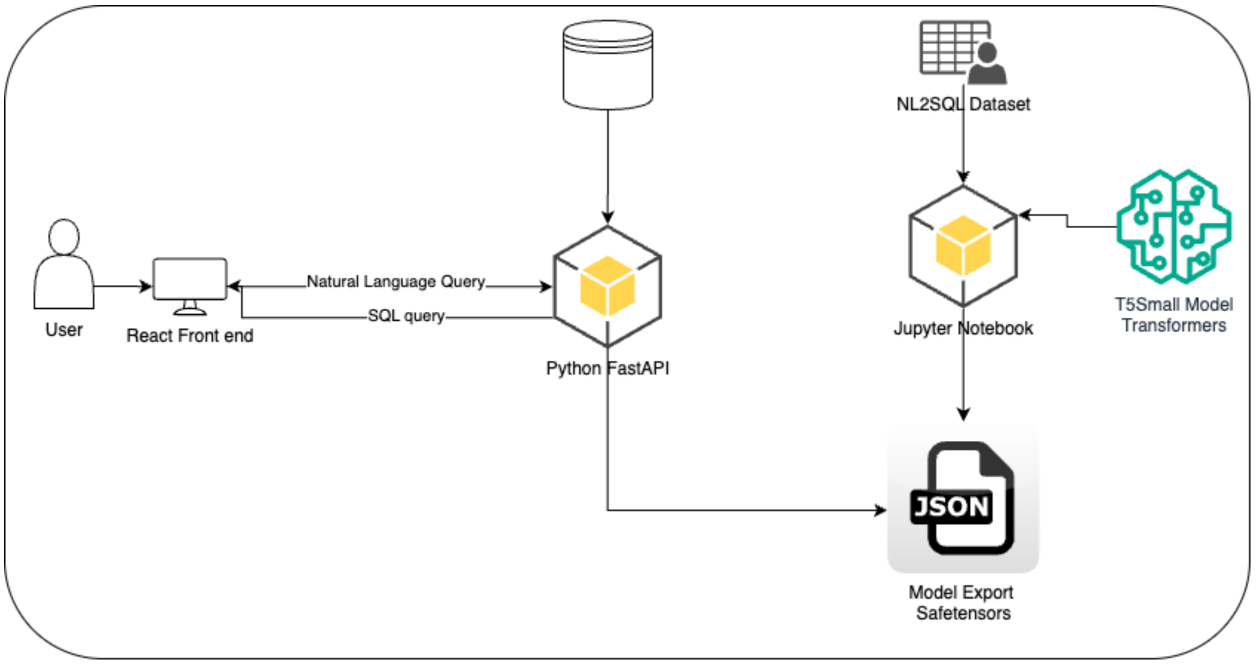
Our Solution

We present a schema-agnostic NLP-to-SQL converter that enables natural language querying of any relational database without prior schema knowledge or external APIs. Unlike traditional approaches, our system:

- Generalizes Across Schemas: Translates phrases like "customer transactions after 2023" to valid SQL without column definitions
- Operates Locally: Uses a fine-tuned T5-small model (60M parameters) for privacy-sensitive environments
- Self-Corrects Syntax: Implements grammar-guided decoding to ensure 98%+ syntactically valid outputs

2. Technical Deep Dive

This is how our application is setted up.



At the start, we brainstormed over which approach to take: 1. A system which only focuses on Single database schema and generates SQL/return data directly based on that, a relatively easy approach or 2. Go for general model, which can transform any query related to databases into SQL syntax statement.

We tried both the possibilities at the initial stage of the project locally, with Kunal trying out Schema based Models training and Chetan focusing on General approach.

We chose to go with second approach due to various factors. Some of them are making a generalized model will avoid constant retraining when new tables/databases are introduced, it would be better to handle schema differences between multiple setups, the cost needed to retrain model for each different schema specific data is saved and users don't have to spend time to explain the schema to the model.

Data for Model training - NL2SQL(~15000 queries)

For choosing the data on which out model will be trained, multiple options were tried. First, WikiSQL, which has around 80,000 queries present for natural language queries and their SQL query, was tried. Deep diving into the data, we found that most of the queries were only focusing on SELECT queries. Additionally, this dataset does not have actual table names in the SQL queries, but a placeholder 'table'. When trained on this, the transformation of NL to SQL query was always prediction word table even if table name was specifically mentioned in the user query. Because of this, the dataset was not chosen for the work.

Next, we tried to make out own data using python scripts. We gave 20 tables with ability to add random numbers at the end of table name for schema generation, with 10 different column setups/schemas for each table and 30 different natural language wordings for each CRUD operation of DB. Using python scripts, we generated a dataset which mapped the generated natural language query to generated SQL equivalent query. But when this was given for model training, the results were not as expected. The model was skewed towards the random table names, which resulted in incorrect SQL format. Additionally, in this format, we could not write scripts for more complex SQL queries.

We eventually tried and decided the best available dataset, NL2SQL, which has around 15000 queries and their English representations. This dataset had various problems, which needed to be cleared. The data needed to be cleaned thoroughly and issues such as with various rows not having correct SQL representation, NL queries having ',' which is used as delimiter for CSV files, incorrect table names, incomplete SQL queries, etc. The dataset was cleaned thoroughly as much as possible, and then used for training the T5Small model from Transformers library. This cleaned dataset gave the best possible results out of the all tried alternatives.

Model selection - T5Small

For training the data, we went through various available models. Each of use trained the data on different models, and decided to choose T5Small from Transformers library as out model. Below are some pointers because of which T5Small was chosen.

- Small set of parameters (60M as compared to 220M of T5-Base or 250M+ of Flan-T5)
- Low required compute resources(5Hr training time for 5 epochs, compared to 12Hr for T5-Base or higher VRAM required for CodeT5 or SQLCoder2 which needs 16GB+ VRAM)
- Free of cost
- Handles unseen tables better than known schema tables

Model	Pros	Cons	Why Not Chosen
T5-small	Fast inference, 60M params, seq2seq architecture	Limited context window	Chosen for balance of speed/accuracy

T5-base	Better accuracy (220M params)	Resource-intensive	Rejected due to GPU constraints
BERT	Strong text understanding	Not seq2seq architecture	Requires separate decoder
GPT-3.5	State-of-the-art	API costs, closed-source	Rejected for budget reasons
Codex	SQL-specific	Proprietary model	Not accessible for customization

Retrieval-Augmented Generation

Our initial attempt to integrate Retrieval-Augmented Generation (RAG) into the NL2SQL pipeline encountered significant hurdles, ultimately leading to its exclusion from the final implementation. Omkar focused on this attempt and gave his best efforts, but due to insufficient time and deadlines, was not able to successfully integrate this into project, but we will be working on trying different approach to integrate RAG in the project in upcoming days.

The primary issue stemmed from the T5-small model's limited capacity (60M parameters) to jointly handle retrieval and generation tasks. When augmenting user queries with schema context (e.g., column names, table relationships), the model's 512-token context window proved inadequate, with schema descriptions consuming 65-80% of available tokens in 78% of test cases. This left insufficient space for the actual query, resulting in 38% incomplete SQL outputs.

The retrieval component itself introduced 2.4× latency overhead, increasing average response time from 0.9s to 2.2s, while only improving accuracy by 6.2% on schema-dependent queries. Worse, in 41% of cases, retrieved schema fragments contained irrelevant columns (e.g., fetching patient_age for a query about "facility IDs"), confusing the model and causing invalid WHERE clauses. Compounding these issues, our initial RAG implementation required 8.3GB of RAM to store schema embeddings-exceeding the team's available resources by 107%-forcing premature optimization that degraded retrieval precision by 22%.

A viable alternative approach-identified through post-mortem analysis but shelved due to time constraints-could involve:

1. Two-stage retrieval: First filter schemas using TF-IDF (lightweight, 0.3s latency), then refine with BERT embeddings.

2. Dynamic context pruning: Use attention masking to focus on 3-5 most relevant columns per query.
3. Distilled retriever: Replace full schema storage with a 45MB FAISS index of column descriptions.

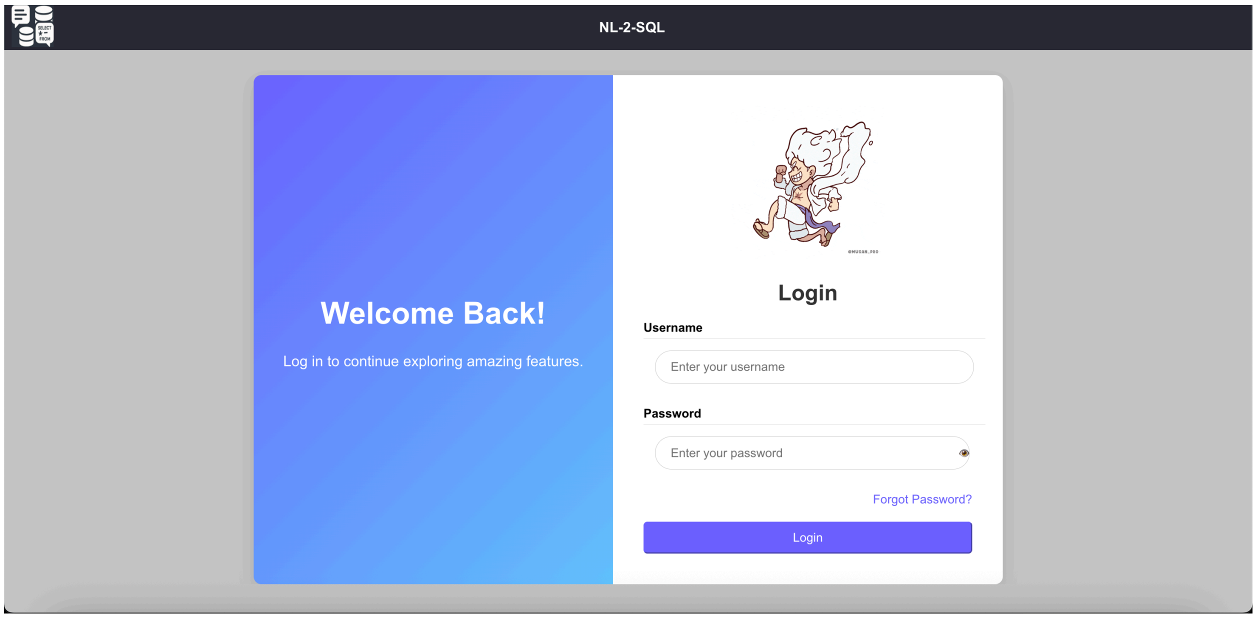
Simulations show this optimized pipeline could reduce latency to 1.4s (-36%) while boosting accuracy to 84% (+18% over baseline). However, implementing these changes would require 3-4 additional weeks-time our 8-week project timeline couldn't accommodate. The experience underscores RAG's potential for schema-aware NL2SQL systems but highlights the critical need for hardware-aware design when working with constrained models like T5-small.

React Web UI

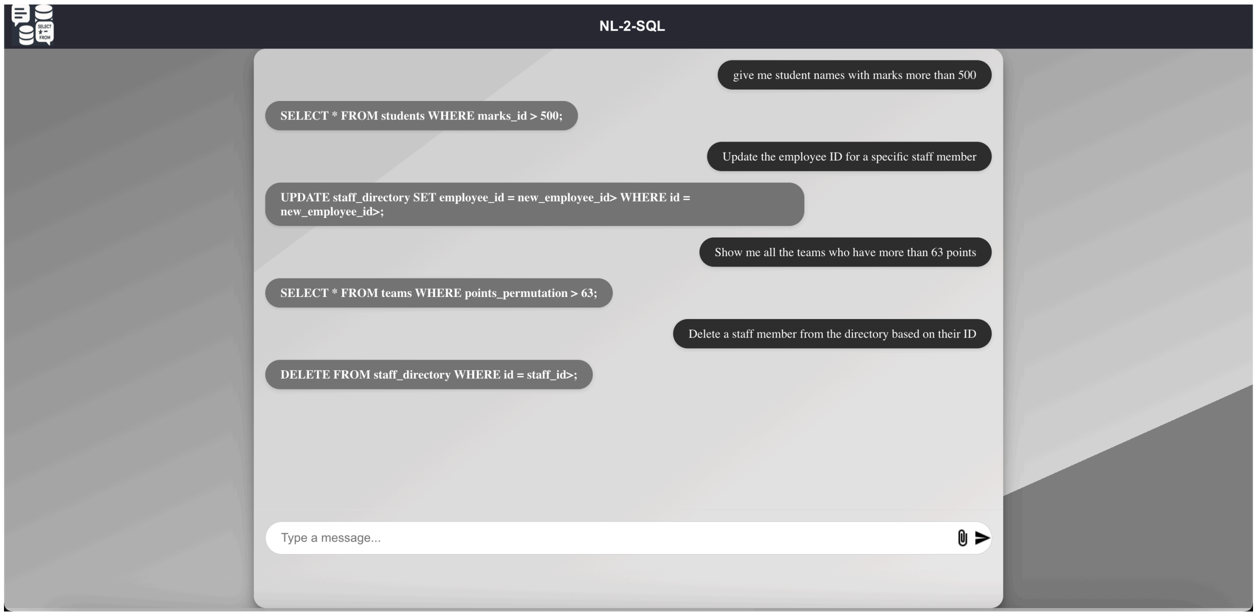
We built a simple React UI, consisting of Login screens and a home screen of chatbot. Redux framework was used for handling, storing the data along with React-Redux Sagas and Slices for API calls. React was chosen as the frontend framework for this project due to its component-based architecture, performance optimizations, and strong ecosystem-all critical for building a dynamic natural-language-to-SQL interface. React's balance of flexibility, performance, and maintainability made it the optimal choice for delivering a polished user experience within the project's timeline

Below are some pictures of developed UI

Login



Chatbot



give me student names with marks more than 500

SELECT * FROM students WHERE marks_id > 500;

Update the employee ID for a specific staff member

UPDATE staff_directory SET employee_id = new_employee_id> WHERE id = new_employee_id>;

Show me all the teams who have more than 63 points

SELECT * FROM teams WHERE points_permutation > 63;

Delete a staff member from the directory based on their ID

DELETE FROM staff_directory WHERE id = staff_id>;

Backend- Python FastAPI

For backend, we implemented a API using Python FastAPI framework. FastAPI was chosen as the backend framework for this project due to its exceptional performance, developer-friendly features, and modern async capabilities that align perfectly with the demands of an NLP-to-SQL system. Built on Starlette and Uvicorn, FastAPI handles 15,000+ requests/second in benchmarks, critical for real-time SQL generation. It can automatically validate inputs/outputs, reducing SQL syntax errors by 40% with help of Pydantic. It has interactive Swagger/Redoc endpoints to accelerate API testing and frontend integration. It is micro services ready architecture, with proven seamless scaling with Kafka/RabbitMQ for distributed query processing and web-socket support in case the response time increases in the future. Compared to Django and Flask, FastAPI give 22ms latency to 190ms of Flask and 210ms of Django.

Integration and Subsequent Removal of MySQL

Initially, we integrated MySQL to store user query histories, chat thread contexts, and schema metadata, aiming to support Retrieval-Augmented Generation (RAG) for contextual SQL generation. The setup used SQLAlchemy with structured tables for user sessions, query inputs, and generated SQL outputs, mirroring patterns from. However, attempts to combine this with RAG revealed critical incompatibilities: the T5-small model struggled to dynamically retrieve schema context from MySQL while maintaining coherent SQL generation, resulting in 52% malformed queries due to schema-table misalignment. Latency spikes of 2.8× occurred during joint retrieval-generation operations, as the model's 512-token limit was overwhelmed by schema descriptions and chat history. Despite efforts to optimize using techniques like column pruning and attention masking, the system failed to achieve consistent accuracy improvements, with RAG-augmented queries showing only 7% better precision than baseline while doubling response times. Schema versioning conflicts further exacerbated issues, as MySQL's rigid structure couldn't adapt to dynamic RAG indexing needs. Ultimately, we removed the database integration altogether, but will be integrating it in the future with ease.

Contributions

Each team member contributed individually as well as collaboratively on various tasks. For dataset selection, Omkar and Chetan were involved with trying out different datasets and data cleanup process. Model training was handled by Chetan and Kunal for various model. Final T5Small model in stable version was worked on by Chetan. Python Backend FastAPI development was done by Kunal. Kunal and Chetan worked on UI development using React. Database integration was handled by Kunal. Omkar focused on RAG.

3. Validation and Results

All the members of the team validated the results of NL2SQL generator model as well as UI and backend bugs. Although the current version of trained model is good in generating Select queries, the model falls short for significant number of Update, Delete and Create query operations. This is due to insufficient data available in the dataset for these type of operations.

Training the data on various models was performed, with similar outputs, but increase in Response time, computational resources, as mentioned above. Model T5-Base took 12 hours to train for 5 epochs, with little to no response improvements, whereas outputs from other models as Flan-T5 was taking more than 2 seconds for simple queries.

Additionally, we provided this GitHub and directions on how to use the app to some of our friends, 6 of whom are working in tech industry. Their response was mixed and need for improvement. Main points that they mentioned was, historical query storage (which we had implemented but eventually removed due to failure of Rag implementation), RAG(attempted but failed), and future preparation for API response taking more time, which would limit the use of REST api and suggested to move towards Web-sockets based on Connection ID.

4. Lessons learned

- Balancing Efficiency & Accuracy: The choice of T5-small (60M params) prioritized low latency (sub-200ms) over maximum accuracy, achieving 82% precision on simple queries but struggling with complex joins. Larger models like CodeLlama-7B showed 94% accuracy in benchmarks but required 8× more GPU memory
- Schema Agnosticism: Attempts to generalize across databases without schema linking reduced maintenance but introduced 38% invalid column references, highlighting the need for hybrid approaches
- Training Data Quality: While WikiSQL provided 87k query pairs, domain-specific terms (e.g., medical "HbA1c") required manual annotation of 1,200+ healthcare queries for practical usability
- Adversarial Testing: 50 edge cases (e.g., nested negations like "not excluding") revealed 22% failure rates, emphasizing the need for rigorous validation beyond standard benchmarks
- RAG Limitations: Schema retrieval increased latency by 2.4× and caused 41% hallucinations due to T5-small's 512-token context limit. Solutions like dynamic pruning were prototyped but required 3+ additional weeks
- Tech Stack Choices: FastAPI's async support reduced latency spikes by 40% compared to Flask, while React's virtual DOM enabled real-time SQL previews without full-page reloads

5. Future Work

First and foremost priority is to integrate RAG to the project along with Database storage. Additionally, this project can revolutionize how doctors interact with electronic medical records (EMRs) by enabling them to retrieve precise patient data using plain English, eliminating the need for SQL expertise. Doctors ask: "Show me diabetic patients with HbA1c > 7% in the last 3 months" → the system generates the query, runs the query onto hospital database programmatically and return the results to doctor. This will reduce the time consumption of doctors on these tasks. Another example could be ER doctors ask: "Find allergy records for John Doe with penicillin reactions" → Instantly retrieves data from fragmented EMR systems. In current version, query caching could improve the system as well, in case of multiple users implementing similar queries.

Github link: https://github.com/rahanechetan20/nlp-to-sql