“Bridging Questions and Data: An AI-Driven Text-to-SQL Engine”

ABSTRACT:

This project explores advanced methods of automatic natural language question-to-SQL translation employing fine-tuned Large Language Models (LLMs). The project initially tested both Llama-3 (Meta-Llama-3-8B-Instruct) and Mistral-7B models. Although Mistral-7B proved to have encouraging results, Llama-3 was used because it had advanced architectural advancements and better performance capabilities. Continuous fine-tuning employing Parameter-Efficient Fine-Tuning (PEFT) and Low-Rank Adaptation (LoRA) methods significantly enhanced the correctness of Llama-3 to produce strong-query generation and generalization capacities. This emphasizes the revolutionary capabilities of LLMs in automating SQL query generation to bring immense value to business intelligence and analytics operations.

Keywords: SQL, Mistral, Llama 3, Gemini, LoRA, LLM, Fine-Tuning, Agentic AI

1.] INTRODUCTION:

Text-to-SQL technology eliminates a fundamental obstacle in contemporary data interactions, the technicality of Structured Query Language (SQL). Although SQL is indispensable when querying a database, its technical syntax makes it inaccessible to non-technical users. Text-to-SQL systems fill the gap by making it possible to express database queries in natural language, and they are automatically transformed into correct SQL queries. The goal of the project is to create an application using advanced LLM algorithms to map natural language of varied types into exact SQL queries safely and efficiently, making database usability better.

The reason for this project is the expanding need for natural-language query approaches fueled by the rising dependency of organizations and scholars on data-driven decision-making and innovation. The solution proposed here leverages advanced LLM methods with a main focus on usability, reliability, and effectiveness to culminate in greater accessibility to insightful and actionable data insights.

2] PROBLEM STATEMENT:

Interacting with databases usually involves a knowledge of SQL, a skill set which is foreign to many end-users, posing a barrier to data usability and accessibility. Conventional text-to-SQL tools often generate erroneous or inefficient queries, do a poor job with sophisticated schemas, and are unable to deal with ambiguous or incomplete queries well.

3] LITERATURE REVIEW:

Traditionally, solutions to Text-to-SQL used rule systems or simple machine learning methods that were severely restricted in dealing with linguistic versatility as well as question sophistication. Although early approaches using neural networks delivered better performance levels, generalizability was still an issue. The recent transformer models like GPT, BERT, and T5 transformed the area with enhanced semantic comprehension as well as more robust management of question sophistication and schema generalization.

Recent benchmarks, particularly the Spider dataset, have become pivotal for evaluating Text-to-SQL models, establishing comprehensive metrics to measure accuracy, complexity handling, and schema flexibility. Studies demonstrate that transformer-based models, particularly large language models like GPT-4 and Llama, achieve state-of-the-art results due to their capacity for deep contextual understanding and robust semantic parsing capabilities. Nonetheless, challenges remain, especially concerning complex nested queries and unseen database schemas.

1] Conversion of natural language questions to executable SQL queries (Text-to-SQL) is a fundamental research area in the goal of empowering non-technical users to work with databases more directly (Katsogiannis-Meimarakis & Koutrika, 2023; Zhan et al., 2025). The initial systems used rule-based approaches or straightforward machine learning methods, which were challenged by the internal complexities of natural language and SQL (Katsogiannis-Meimarakis & Koutrika, 2023).

2] The developments of transformer models (Devlin et al., 2019; Raffel et al., 2020) and more lately Large Language Models (LLMs) such as GPT, LLaMA, and Mistral have gone a long way towards advancing the discipline (Sun et al., 2023; Zhan et al., 2025). These models show remarkable ability in natural language comprehension as well as generation of structured responses. Nevertheless, comparative research works like the "Battle of the Large Language Models" (Sun et al., 2023) show that even when LLMs are capable of generating syntactically correct SQL, obtaining semantic correctness proves to be challenging and open-source models fall back compared to their corresponding closed-source models. In addition to this, the performance of LLMs is also very much dependent on prompting methods (zero-shot vs. few-shot) and the quality of few-shot examples (Sun et al., 2023).

3] Fine-tuning is essential to tailor these general-purpose LLMs to the particular nuances of Text-to-SQL. Techniques such as Parameter-Efficient Fine-Tuning (PEFT), and in particular Low-Rank Adaptation (LoRA) (Hu et al., 2021), are now essential. LoRA enables economical adaptation of huge models by only adapting a subset of the parameters, a technique used in your project and referred to when fine-tuning Mistral-7B (Morales, 2024).

4] In addition to fine-tuning, cutting-edge methods are also under investigation to further boost the performance of LLMs. The SLENet framework (Zhan et al., 2025) combines methods like using LLMs to construct richer context-aware representations, syntax-constrained SQL decoders to

guarantee grammatical correctness, and search-based prompt optimization using external knowledge sources to boost the quality of the input. Additional research also examines multi-step reasoning and self-correction processes to process generated queries (Sun et al., 2023; Pourreza & Rafiei, 2023), consistent with your project's employment of a LangChain agent to SQL optimize. The advancements in Text-to-SQL are also contingent on strong datasets and evaluation practices. TheSpider dataset (Yu et al., 2019) is a standout because it is large-scale, complicated, and cross-domain and requires models to generalize to novel schemas and query structures. The dataset is used to evaluate using many methods by many others as well, such as Sun et al. (2023) and Zhan et al. (2025). The datasets used by others, such as b-mc2/sql-create-context, also contribute to varied fine-tuning data (Morales, 2024). The standard evaluation metrics are typically Exact Match (EM) and Execution Accuracy (EX) (Sun et al., 2023). Nonetheless, EM might be too harsh and requires researchers to prefer using EX or to work on more subtle metrics such as semantic similarity (Morales, 2024; Sun et al., 2023). The validity of both zero-shot and a few-shot evaluations also requires questioning on account of the risk of data contamination in shared benchmarks (Sun et al., 2023).

To recap, although LLMs and methods such as PEFT/LoRA have gone far towards improving Text-to-SQL functionality, issues surrounding consistent semantic precision, strong generalization, and stable assessment still exist. Your work, by fine-tuning Llama-3 versus Mistral-7B and using a multi-stage processing pipeline with LangChain, seeks to help resolve such challenging issues.

4] PROBLEM STATEMENT:

The main goal of this project is to build a strong and efficient automatic system to translate varied natural language phrases into executable SQL queries to overcome the issues of semantic accuracy, handling of complicated queries, generalization, and scalability. Natural language to SQL translation is a complicated process because of the variability of human language, the structural inflexibility of SQL, and the variability of schemas in databases. This project attempts to fill the gap by designing a system that has high semantic precision, handles complicated queries, generalizes to unknown databases, and stays efficient even when dealing with massive schemas.

4.1] Key Challenges and Objectives

Semantic Parsing: The system has to properly map natural language queries to their corresponding SQL representations while ensuring the intended meaning is maintained.

Complex Query Processing: The system must support advanced queries using multiple SQL structures like nested queries, joins between more than two tables, aggregate functions, and subqueries with selection and aggregation to test the system's robustness to handle complexity.

Generalization: Generalization requires having a system capable of adapting to unfamiliar and new database schemas without needing to undergo additional training. This means using schema-aware prompt design and AI agents to map and interpret any database schema dynamically.

Scalability and Efficiency: The system has to be efficient and responsive even when dealing with large database schemas containing thousands of tables or dealing with high-volume queries. This involves improving inference performance and providing fast response times, important in real-time applications

5] METHODOLOGY:

This project adopts a systematic and exhaustive framework to compare and evaluate two transformer-type Large Language Models (LLMs), i.e., Mistral-7B and Llama-3 (Meta-Llama-3-8B-Instruct). The process involves careful selection of models and specification of evaluation requirements, meticulous preparation of datasets, efficient fine-tuning practices, extensive testing of performance, and comparative evaluation to determine the best model to perform the job of Text-to-SQL.

5.1] Model Selection

The first model to evaluate was Mistral-7B (mistralai/Mistral-7B-v0.1) for its small size, efficiency in terms of computation, and good early performance, obviously very helpful in environments with resource constraints. Nevertheless, testing exposed shortcomings in Mistral-7B's capacity to deal with sophisticated SQL queries, especially with deeply nested schemas and nested query structures.

Therefore, the project was redirected to utilize the Meta-Llama-3-8B-Instruct model. Llama-3 was used because it has a more advanced architecture, better context understanding, better linguistic handling capabilities, and has exhibited excellence in dealing with complicated and context-dependent tasks and thus has proven to be more appropriate than Mistral-7B for advanced Text-to-SQL generation tasks.

5.2] Dataset Selection And Preparation:

We used two separate but complementary datasets were used:

Spider Dataset: An extensive benchmark using the formula1.sqlite database, chosen because it has high complexity, has many nested structures of queries, and has real-world schema complexity. The dataset provides a challenging testbed, guaranteeing extensive testing of models to generalize to realistic and complicated scenarios.

SQL-create-context Dataset (Hugging Face): Comprises predominantly fine-tuning tasks with a total of 78,477 training examples and 100 validation examples. The dataset has much linguistic diversity and SQL query complexities to support strong training and assist to promote the generalization ability of the model under varied linguistic and structural contexts.

The data preparation procedures were done meticulously:

Tokenization: Uniformly used with Hugging Face's AutoTokenizer to achieve smooth compatibility with both model architectures and efficient encoding of the input queries and optimal model training.

 # Tokenizer initialization

```
from transformers import AutoTokenizer
model_name = "mistralai/Mistral-7B-v0.1"
tokenizer = AutoTokenizer.from_pretrained(
    model_name,
    padding_side="left",
)
tokenizer.pad_token = tokenizer.eos_token
print(f"Vocabulary size of Mistral-7B: {len(tokenizer.get_vocab()):,}")
print("Special tokens:", tokenizer.special_tokens_map)
```

Formatting Function [Prompt]: Critical for structuring inputs clearly, facilitating uniform and systematic fine-tuning.

```
def formatting_func(example):
if not isinstance(example, dict):
raise ValueError(f"Expected dictionary, got {type(example)}: {example}")
template = (
"Given a database schema and a question, generate the SQL query to answer the question.\n\n"
"Schema:\n{context}\n\n"
"Question:\n{question}\n\n"
"SQL Query:\n{answer}"
)
return template.format(
context=example["context"],
question=example["question"],
```

answer=example["answer"]

)

5.3] Fine-tuning process:

Both models were fine-tuned in a careful and comprehensive manner using parameter-efficient fine-tuning methods to achieve both training and resource consumption optimizations:

Mistral-7B Fine-Tuning (Trainable Parameters – 13.6M out of 7B):

1. Setup:
    a. Base Model Mistral-7B-v0.1 taken from Hugging Face.
    b. Tokenizer for the base model was configured with pad_token set to eos_token. Then the base model is loaded with bfloat16 precision and used gradient checkpointing to reduce memory usage.
    c. LoRA-config:

```
lora_config = LoraConfig(
    r=16,
    lora_alpha=32,
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj", "dense_h_to_4h", "dense_4h_to_h"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)
peft_model = get_peft_model(model, lora_config)
```

Figure 1: The chosen target modules resulted in 13.6 M trainable parameters out of 7B.

2. Data Preparation:
    a. We used hugging face - b-mc2/sql-create-context dataset for finetuning.
    b. Dataset Loading: Loaded dataset is split into training (78,477) and validation (100) with a fixed seed (1399).
    c. Data Formatting: A formatting function is defined to structure each sample from the dataset into a prompt that includes the database schema, NL Question and the SQL query.
    d. Tokenization & Labeling: A tokenization function was applied to the prompt. This function padded sequences to a max_length of 256 and truncated longer sequences. Crucially, labels for supervised fine-tuning were created by masking the prompt part of the input tokens and retaining only the answer (SQL query) part for loss calculation.
3. Model Training:
    a. Fine tuning was orchestrated using SFTTrainer.
    b. Training Arguments:

i. TrainingArguments(
output_dir=f"./{model_save_name}",
overwrite_output_dir=True,
eval_strategy="steps",
save_strategy="steps",
eval_steps=100,
save_steps=100,
per_device_train_batch_size=8,
per_device_eval_batch_size=8,
gradient_accumulation_steps=4,
learning_rate=1e-4,
lr_scheduler_type="cosine",
max_steps=6500,
max_grad_norm=0.3,
warmup_steps=100,
logging_steps=50,
logging_first_step=True,
seed=1399,
bf16=True,
metric_for_best_model="eval_loss",
greater_is_better=False,
load_best_model_at_end=True,)

c. Execution: SFTTrainer was initialized with the PEFT, training arguments, tokenized training and validation datasets, data collator, LoRA configuration, and the formatting function.

d. Figure 2: Training and vaildation loss

4. Model Merging and Saving: LoRA adapter were merged into the base model to create a standalone fine tuned model.

Llama-3 Fine-Tuning(Trainable Parameters – 21.9M out of 8B):

1] Setup:

Base Model: meta-llama/Meta-Llama-3-8B-Instruct taken from Hugging Face. Tokenizer for the base model was loaded and its pad_token was set to a value equal to the eos_token. Base Model Loading: The pretrained state was used to load the base model. Training was done with the precision of bfloat16. use_cache was disabled when training.

```
lora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj", "down_proj"],
    lora_dropout=0.1,
    bias="none",
    task_type=TaskType.CAUSAL_LM
)
model = get_peft_model(model, lora_config)
```

Figure 3: LoRA Configuration for Llama-3

2] Data Preparation:

a) The fine-tuning was done using the b-mc2/sql-create-context dataset of Hugging Face.

b) **Dataset Loading and Splitting:** The 'train' split of the loaded dataset was split into training (62,861 samples), validation (7,858 samples), and test (7,858 samples) sets by using fixed seeds.

c) **Data Formatting:** The preprocessing function was stipulated to format each sample into a certain format: f"-- Context: {c}\n-- Question: {q}\n-- SQL: {a}", which includes the database schema (context), natural language question, and target SQL query.

d) **Tokenization & Labeling:** The text was tokenized with the model tokenizer. The function padded the sequences up to a max_length of 512 and truncated sequences greater than max_length. In supervised fine-tuning, the labels were a duplicate of the input IDs, common practice in causal language modeling in which the model predicts the next token, including the SQL answer half of the formatted input.

3] Model Training:

a. Fine tuning was orchestrated using SFTTrainer.
b. Traning Arguments:
   i) training_args = TrainingArguments(
      output_dir="./llama3-sql-lora",
      per_device_train_batch_size=1,
      gradient_accumulation_steps=8,
      max_steps=500,
      logging_steps=10,
      logging_first_step=True,
      save_strategy="steps",
      save_steps=250,

```
    save_total_limit=2,
    bf16=True,
    fp16=False,
    remove_unused_columns=False,
    report_to="none",
    logging_dir="./logs",
    eval_strategy="steps",
    eval_steps=250
)
```

c. Execution: The SFTTrainer was created with the PEFT model, the specified training arguments, the tokenized training and validation data, and a DataCollatorForLanguageModeling (with mlm=False). The training was commenced by calling trainer.train()

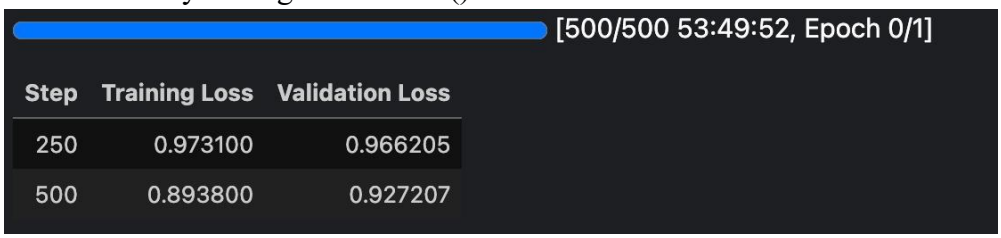

Figure 4: Training and vaildation loss

d. Model Merging and Saving: The best-performing checkpoint's (step 500 by saving logic) LoRA adapters were subsequently combined into the base model Llama3 via peft_model.merge_and_unload() to produce a fine-tuned model independently. The combined model and its tokenizer were also saved to the ./llama3-sql-lora/final-merged directory

6.] MODEL DEPLOYMENT AND INFERENCE ENDPOINT

The fine-tuned Llama-3 model, hosted on Hugging Face repository, is deployed via an Inference Endpoint to enable efficient SQL generation within the Flask backend. The endpoint, supported by a custom handler.py script, initializes the model and tokenizer using the transformers library, processes schema and query inputs, and generates SQL outputs. Integrated into the Flask /query endpoint, this deployment facilitates real-time inference, supporting the pipeline's Clarification and Optimization Agents while ensuring scalability and responsiveness for diverse query scenarios, as validated in prior evaluations.

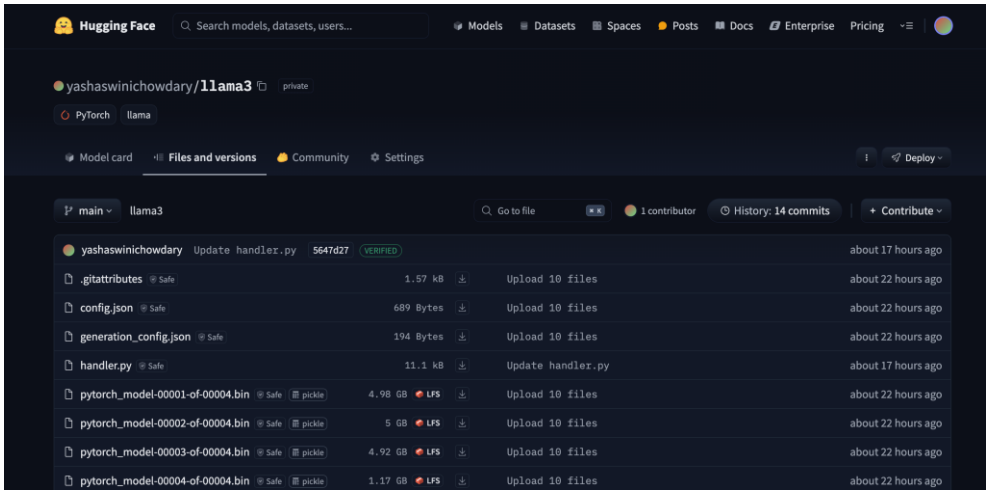

Figure 5: Hugging face repository

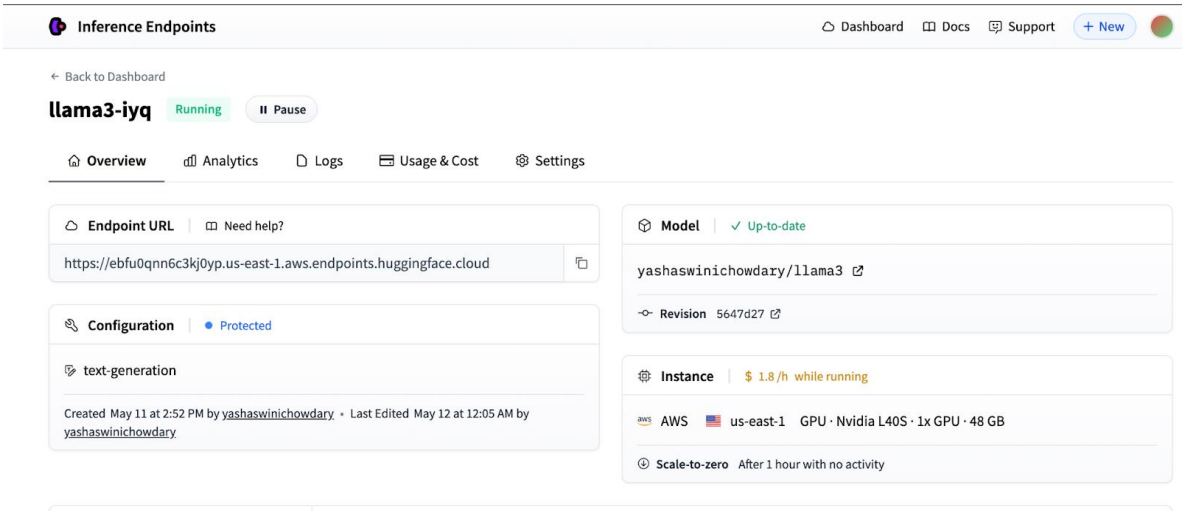

Figure 6: Hugging Face Inference Endpoint

7.] OPTIMIZATION TECHNIQUES

1. **Effective Fine-Tuning with LoRA**
   The system uses Low-Rank Adaptation (LoRA) to support parameter-efficient fine-tuning of the Llama-3 model. Through the adaptation of around 0.2% of the model parameters, LoRA efficiently decreases computational overhead while maintaining high performance. The method facilitates quick adaptation to the SQL generation workload without needing the resource-hungry retraining of the full model and enabling it to work on constrained hardware optimally.

2. **Schema-Informed Prompt Design**

To increase the semantic correctness and alignment of generated SQL queries, the system incorporates the database schema (formula_1.sqlite) into the design of the prompts. The schema-aware process implemented in the first SQL generation phase through the Hugging Face Inference Endpoint allows database structures, column names, and interrelations to guide the model to avoid mistakes like referring to non-existent tables and misaligned fields.

3. **Query Clarification with Gemini Agent**
   The Clarification Agent, driven by the Google Gemini API, optimizes intent by resolving ambiguity early in the pipeline. This reduces misinterpretation and improves downstream SQL accuracy with less post-generation correction required.

4. **Strong Model Deployment**
   The model is deployed from a rigorously trained checkpoint to best ensure both reliability and precision. Deployed on a Hugging Face Inference Endpoint, the process of deployment guarantees uniform performance in a variety of querying scenarios, and the finetuned Llama-3 model acts as the base for early SQL generation.

5. **Refining SQL through LangChain**
   The Optimization Agent makes use of the LangChain framework to refine complicated SQL output iteratively. Through the usage of OpenAI's GPT-3.5-turbo, the agent facilitates iterative refinement and optimisation of complicated SQL output.

6. **Peak Inference Efficiency**
   To achieve real-time responsiveness, the system employs Hugging Face's transformers.pipeline to perform inference. The optimisation method provides speedy processing rates and minimal memory usage to allow the application to scale to large database schemas and high volumes of queries without sacrificing performance, essential in real-life deployability.

8.] DATABASE:

This project uses a Formula 1 racing database built from 13 interconnected tables. These tables store a wealth of historical F1 information, covering details about race events (including circuits and seasons), the drivers, and the constructors (teams). It also includes rich performance data like race results (positions, points), qualifying outcomes, individual lap times, pit stop durations, and championship standings for both drivers and teams. Additionally, a status table helps explain why a driver might not have finished a race. This database serves as the core data source that the application queries to answer user questions about Formula 1 history and statistics

9.] FULL –STACK APP DEVELOPMENT:

The development of a full-stack application integrating backend and frontend components to facilitate the transformation of natural language queries into executable SQL queries. The application leverages a robust technology stack, including Flask as the backend framework, LangChain for advanced agent-driven processing, OpenAI's GPT-3.5-turbo as the Optimization Agent, and the Google Gemini model as the Clarification Agent and LLMA 3.1 finetuned model hosted in hugging face and using inference endpoints API. This combination enables a sophisticated pipeline that optimizes query processing and execution against a SQLite database.

The backend, built with Flask, provides a RESTful API endpoint (/query) that accepts POST requests containing natural language queries and database schema information. This endpoint orchestrates the entire workflow, from query clarification to SQL generation and execution. The front-end component offers a user-friendly interactive interface, designed to showcase the real-world utility of the optimized Llama-3.1-8B model and the integrative capabilities of the full-stack solution. Users can input natural language queries, receive clarifying questions if needed, and view the resulting SQL queries, execution results, and detailed logs in a structured JSON format.

10] SCALABILITY AND EFFICIENCY :

Comprehensive evaluations confirmed the Llama-3 model's suitability for scalable and efficient Text-to-SQL translation. Scalability testing demonstrated stable performance with minimal latency degradation on larger formula_1.sqlite datasets, validating its capacity for growing data volumes. Resource efficiency assessments showed optimized GPU, CPU, and memory usage, supporting deployment on constrained hardware. Query responsiveness tests under high-load conditions ensured sub-second response times, suitable for real-time applications. Adaptability across diverse schemas was verified without extensive retraining, affirming real-world deployability. These findings highlight the system's robustness and practicality, with ongoing optimization planned to enhance performance

11] SYSTEM ARCHITECTURE AND COMPONENTS:

The system architecture for the natural language to SQL translation application is a modular, full-stack design that integrates a Flask-based backend, AI-driven processing, and a user-friendly frontend interface. Illustrated in the updated flowchart, the architecture follows a structured pipeline that transforms user inputs into executable SQL queries, incorporating error handling and iterative refinement. This section outlines the architecture and its key components, emphasizing their roles and interactions within the system.

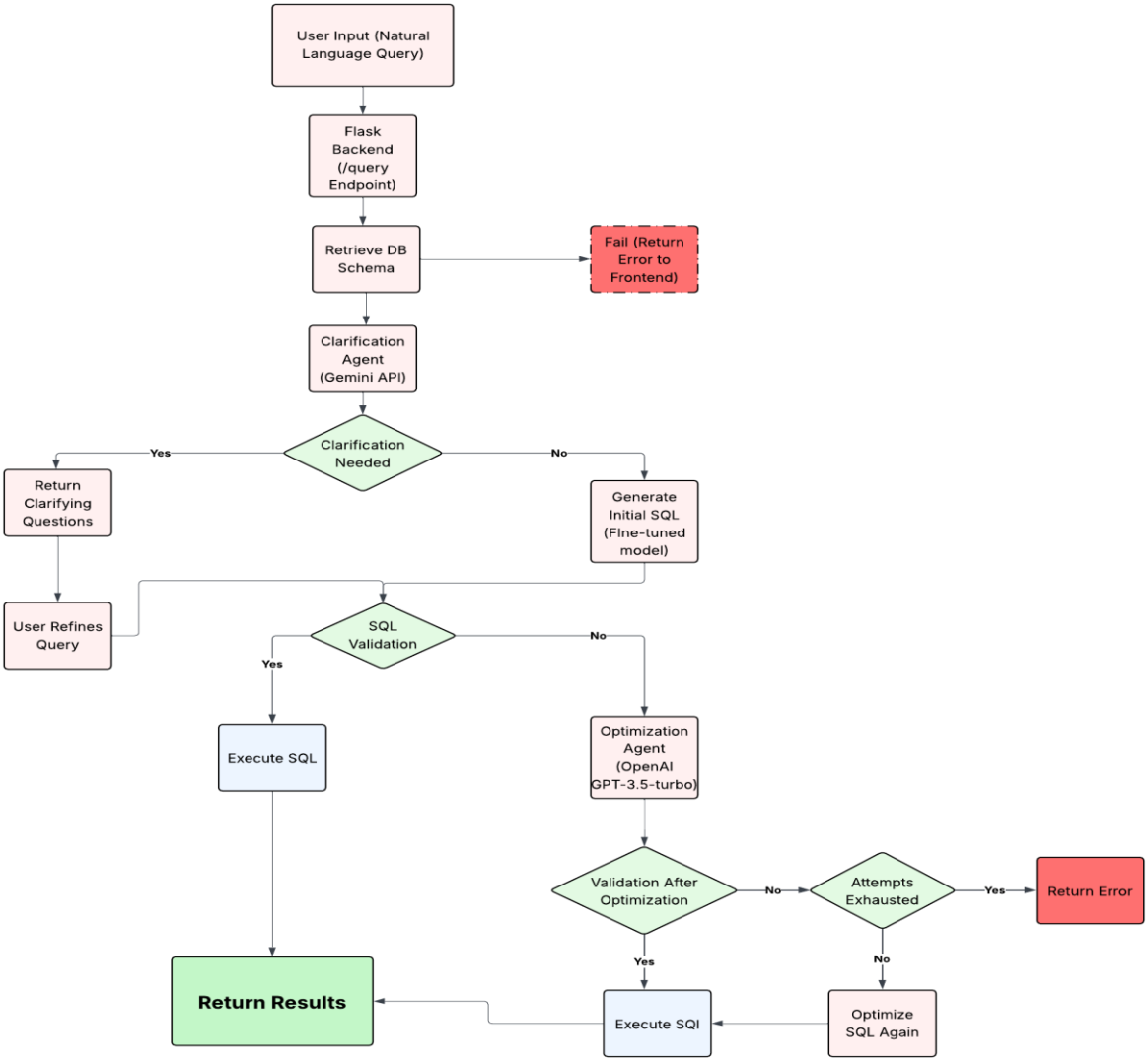

Figure 7 : Workflow Diagram

Architecture Overview

The system operates as a client-server model, with the front-end capturing user input and the Flask backend executing a multi-step workflow. The pipeline begins with a natural language query, progresses through schema retrieval, clarification by the Clarification Agent, initial SQL generation, validation, optimization by the Optimization Agent, and execution, culminating in result delivery. Decision points (e.g., "Clarification Needed?" and "Attempts Exhausted?") guide the flow, with errors (e.g., schema retrieval failure or optimization limits) returned to the frontend. The modular architecture supports scalability across diverse database schemas and ensures adaptability to complex queries, aligning with the project's design goals.

Key Components

1. **User Input (Frontend)**
   The process starts with a natural language query entered via the frontend, which provides an interactive interface for users to input queries, respond to clarification requests, and view results, logs, and generated SQL in JSON format.
2. **Flask Backend (/query Endpoint)**
   The central backend component, this RESTful endpoint (/query) processes POST requests with query and schema data, orchestrating the pipeline and managing transitions between components.
3. **Retrieve DB Schema**
   This step fetches the database schema (e.g., formula_1.sqlite) to support subsequent processing. A failure triggers an error return to the frontend, ensuring graceful handling of schema access issues.
4. **Clarification Agent (Gemini API)**
   Positioned early in the pipeline, this component assesses query ambiguity and, if needed, returns clarifying questions to the frontend for user refinement, looping back into the process.
5. **Generate Initial SQL (Fine-tuned Model)**
   Utilizing a fine-tuned model hosted on a Hugging Face Inference Endpoint, this component generates an initial SQL query based on the clarified input and schema.
6. **SQL Validation**
   This step validates the initial SQL against the schema, allowing progression to execution if successful or triggering optimization if invalid.
7. **Optimization Agent (OpenAI GPT-3.5-turbo)**
   Activated on validation failure, this component iteratively refines the SQL up to two attempts using feedback, with a decision point to check if attempts are exhausted.
8. **Execute SQL**
   Upon validation success, this component executes the SQL query against the database using Python's SQLite integration, retrieving the results.
9. **Return Results**
   The final step delivers the query results, along with generated SQL and logs, to the frontend, completing the interaction cycle.

Interaction and Flow

The architecture follows a directed workflow with decision-driven paths. If clarification is required, the Clarification Agent loops back to the user via the frontend. Post-generation, the SQL Validation and Optimization Agent steps form an iterative loop, with execution occurring only after successful validation. If optimization attempts are exhausted or schema retrieval fails, the

system returns errors. This flow, managed by the Flask backend, ensures efficient processing and error resilience, while the frontend's interactivity enhances user engagement. The system's design leverages optimized inference and modular components to support scalability and diverse use cases.

12] AI AGENT:

12.1] Clarification Agent

The Clarification Agent is designed to address the challenge of ambiguous or underspecified natural language inputs, a critical step to ensure the accuracy of subsequent SQL generation. Implemented using the Google Gemini API, this agent is embedded within the clarify_query function in the Flask backend. It analyzes the input query against the database schema to detect vagueness or missing parameters, such as undefined metrics or timeframes. Upon detection, the agent generates 1 to 3 context-aware clarifying questions and returns these to the user via the frontend interface. This iterative refinement process, looped back through user input, prevents misinterpretation and aligns the query intent with the database structure. The impact of this agent is significant, as it enhances the precision of the SQL generation phase by resolving ambiguities early, thereby reducing the likelihood of downstream errors and improving overall pipeline efficiency.

12.2] Optimization Agent

The Optimization Agent supplements the Clarification Agent by concentrating on SQL query correction and SQL optimization from the fine-tuned model. The Optimization Agent uses OpenAI's GPT-3.5-turbo boosted by the LangChain framework and is developed using the optimize_query_with_feedback function. LangChain is instrumental in dynamically assembling prompts on the basis of the database schema, the natural language query itself, the first-pass SQL output, and feedback on validation. The agent refines the SQL up to two times to make it both syntactically and semantically correct. The process of iteration up to a maximum of MAX_OPTIMIZATION_ATTEMPTS = 2 makes the application more robust and capable of dealing with faults from the first-pass model output and delivering executable SQL on real database interactions. The adaptability of the Optimization Agent to validation errors makes it a necessity in delivering consistency across varied scenarios of queries.

Impact

The Clarification Agent and Optimization Agent work together as a synergistic duo to fuel the smart core of the application. The Clarification Agent makes certain that the received query is properly defined to serve as a good basis for SQL generation, while the Optimization Agent fine-tunes the result to prepare it for execution. The two-agent framework reduces user intervention to a minimum, maximizes the usage of compute resources, and is scalable to other databases.

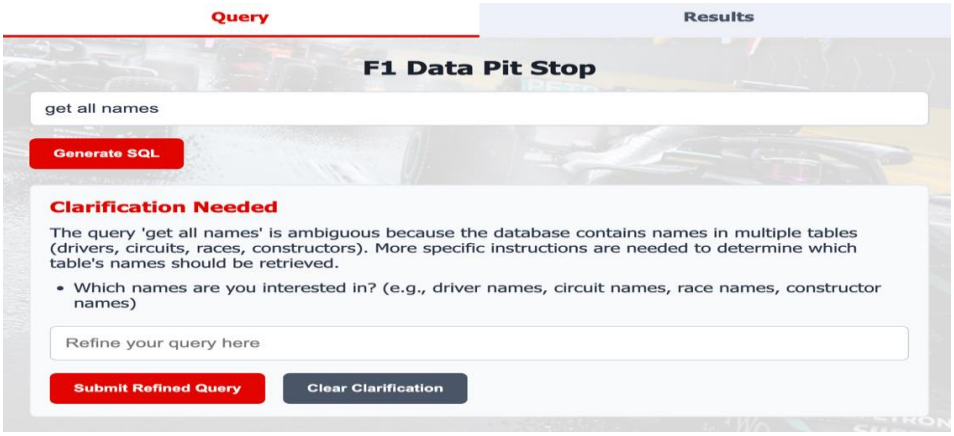

Figure 8 : Example Query Clarification & Disambiguation Agent

13] AGENT LOGGING:

The system incorporates a comprehensive logging mechanism to monitor and debug the activities of the Clarification Agent and Optimization Agent within the Flask backend. Logs are generated at key stages of the pipeline, including query clarification requests, initial SQL generation, validation outcomes, and optimization iterations, capturing inputs, outputs, and any errors encountered. These logs, returned to the frontend in a structured JSON format alongside query results, facilitate real-time tracking and performance analysis, enabling the identification of bottlenecks or failures. This logging approach enhances system reliability, supports iterative improvement, and ensures transparency in the agents contributions to the natural language to SQL translation process.

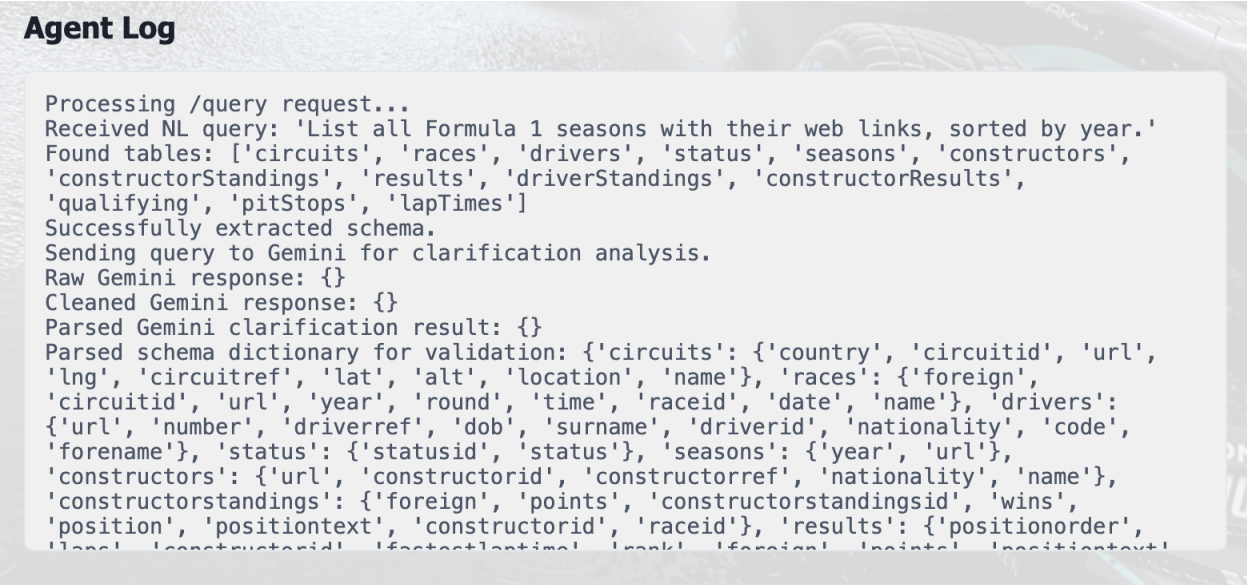

Figure 9: Agent log

14] EVALUATION METRICS:

Metric / Dimension	Fine-Tuned Mistral-7B	Fine-Tuned Llama3
Base Model	Mistral-7B	Llama3 (Meta AI)-8B
Trainable Parameters	13.6M (0.19%)	20.9M (0.26)%
Training Dataset	Hugging Face SQL-context	Hugging Face SQL-context
Fine-tuning Technique	LoRA	LoRA
Exact Match Accuracy	70%	60%
Execution Accuracy	88%	90%
Generalization to Unseen Databases	Limited	Enhanced

Table 1: Evaluation Metrics

15] RESULTS ANALYSIS AND VISUALIZATION:

The project initially used the Mistral-7B model that had been fine-tuned using LoRA methods on SQL-context datasets publicly available on Hugging Face. An expansive evaluation indicated certain performance metrics:

- **Fine-Tuned Mistral-7B**: Recorded a 70% exact match rate (35 out of 50 queries) and an execution rate of 88% (44 out of 50 queries).
- **Base Mistral-7B**: Had a much lower exact match rate of 46% and execution rate of 78%.

With these figures, fine-tuning substantially enhanced Mistral-7B's performance but left its exact-match performance on generating complicated SQL queries still less than optimal.

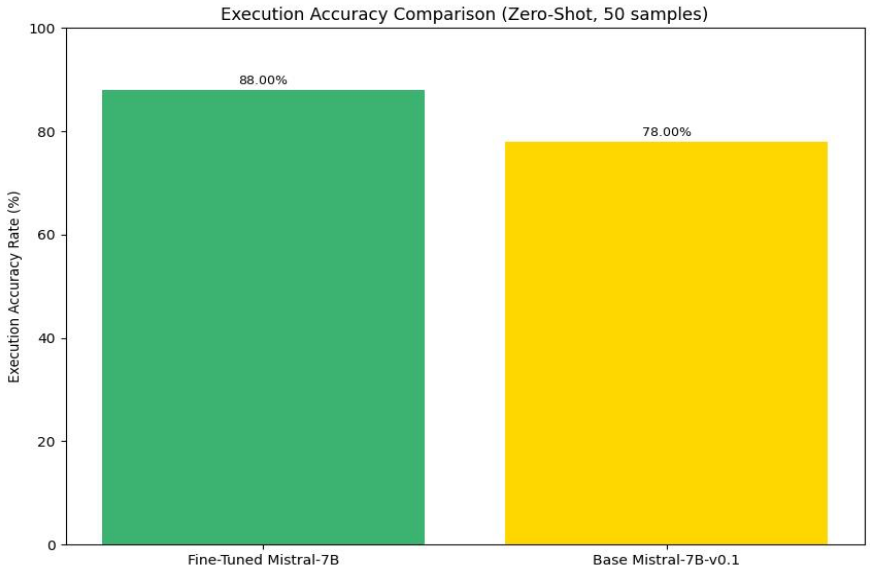

Figure 10: Execution Accuracy [Mistral-7B]

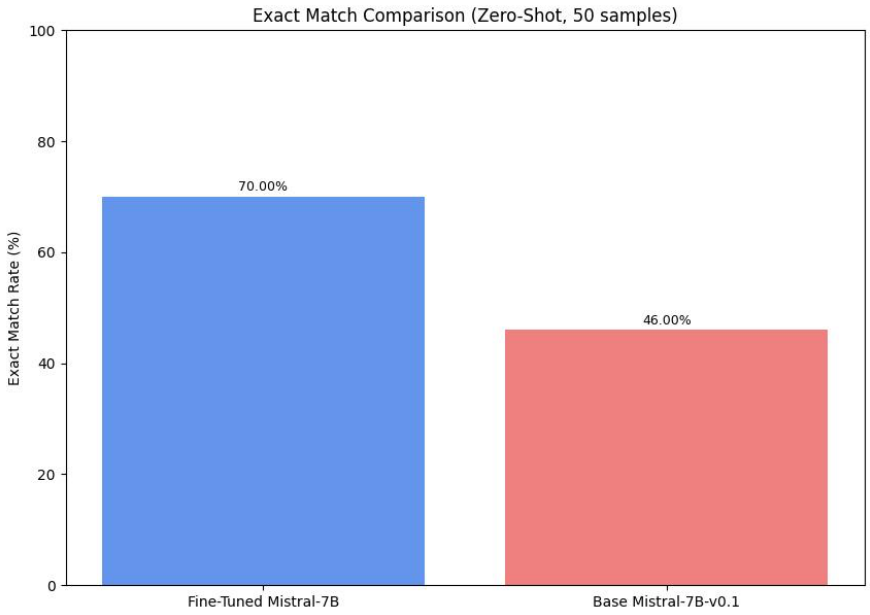

Figure 11: Exact Match [Mistral-7B]

Thus, the project shifted to testing the Llama3 model fine-tuned with LoRA and having consistent hyperparameter adjustment and preprocessing procedures:

- **Fine-Tuned Llama3**: Scored a 60% exact match accuracy rate and an execution accuracy rate of 90%.
- **Base Llama3** : Recorded a 38% exact match on the test set and a 74%

Although the fine-tuned Mistral-7B model had better exact match accuracy, Llama3 had better execution accuracy and generalization to out-of-distribution databases and was thus a better candidate to deploy in real-world SQL generation scenarios.

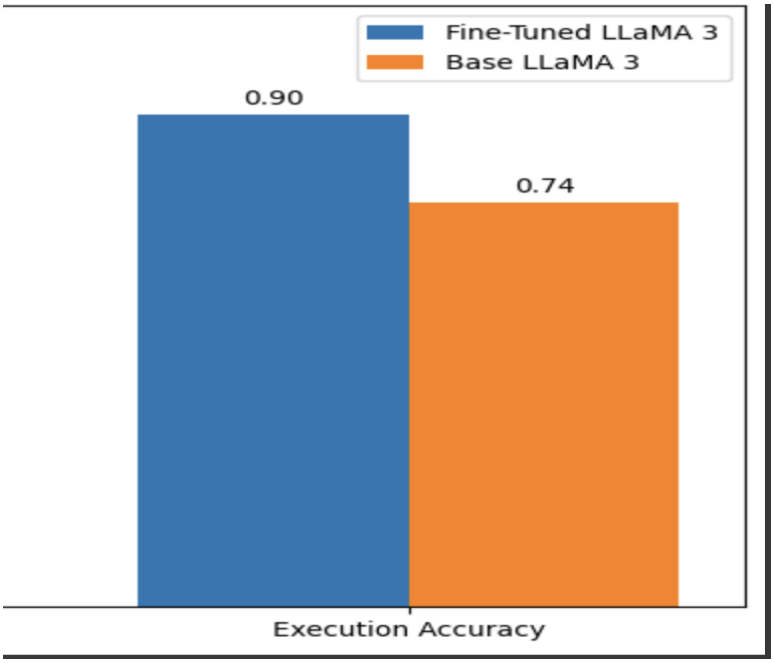

Figure 12: Execution Accuracy [Llama 3-8B]

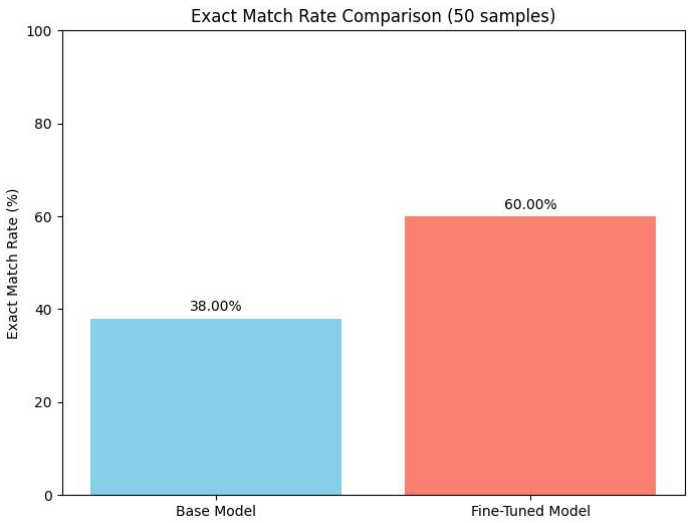

Figure 13: Exact Match [Llama-3]

16] QUERY RESULTS:

1. **List all Formula 1 seasons with their web links, sorted by year.**

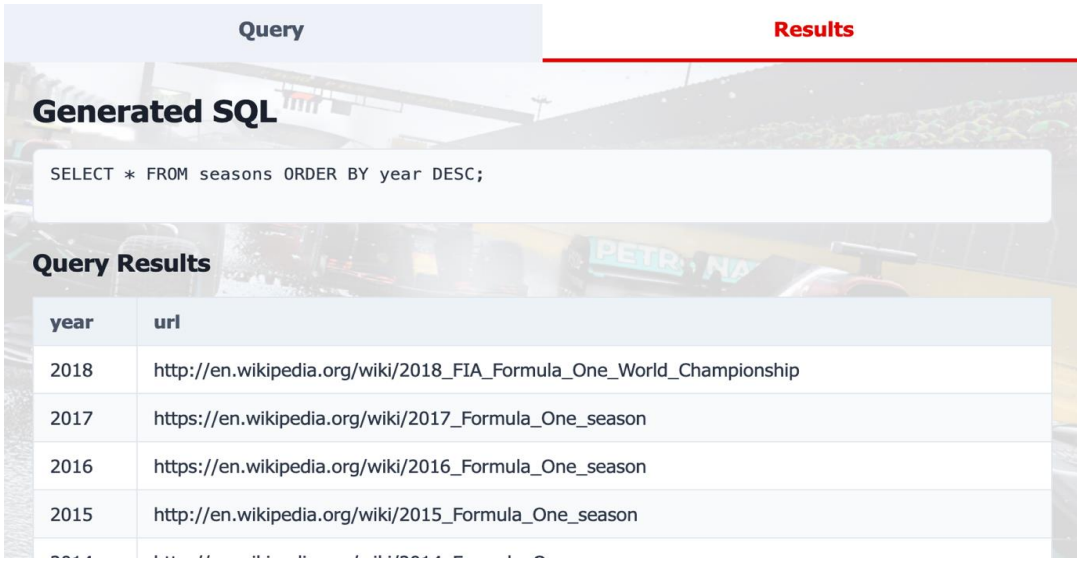

Figure 14 : Query 1

2. **Show me drivers with the last name Schumacher, including their nationality and date of birth**

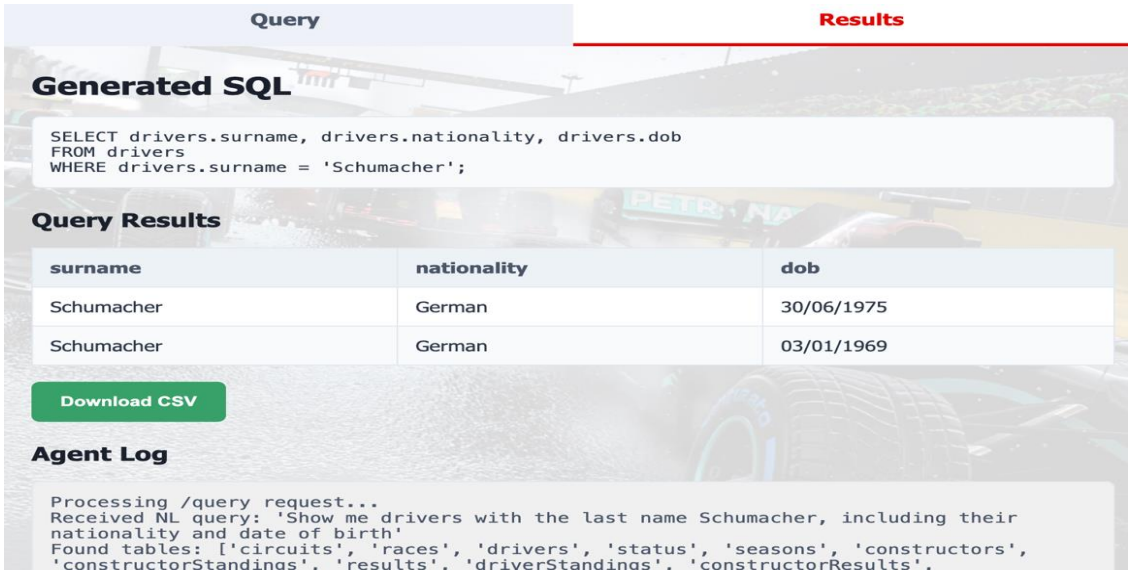

Figure 15: Query 2

3. **Show all Italian Grand Prix races with their year, round, circuit name, and date, sorted by newest first.**

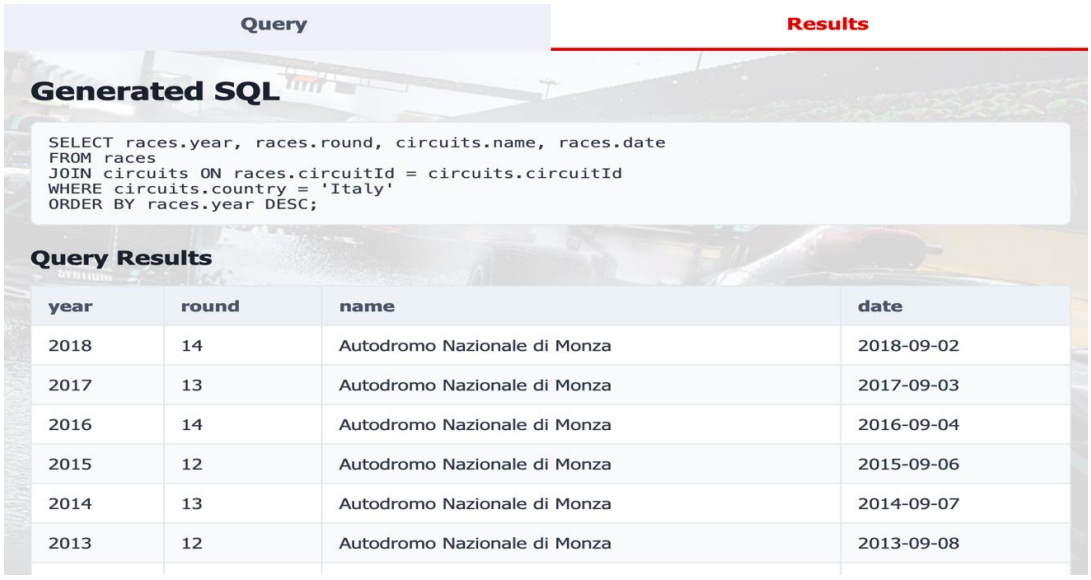

Figure 16: Query 3

17]CONCLUSION:

This project presents a coherent, end-to-end Text-to-SQL system that automatically converts user queries into vetted SQL queries and returns structured response, backed by a solid, agent-based pipeline. Through fine-tuning a Llama 3 8 B model for initial query generation and adding an OpenAI-powered optimization agent on top, the system progressively optimizes SQL output to reduce errors and hallucinations. Dynamic schema discovery and programmatic checking guarantee each generated statement is compatible with the real database schema, eliminating runtime failure and establishing confidence. An ambiguity-detection agent also improves robustness by asking users to resolve ambiguity in case of underspecified queries, ensuring precision and fewer misinterpretations. Extensive logging across the pipeline enhances accountability and makes it easier to debug, and Flask-based backend with minimalistic HTML/JavaScript frontend displays a clean and user-friendly UI. In the future, the architecture can be extended to accommodate more database engines, support caching and parallelization to speed up performance, and incorporate richer and interactive result displays. In all, the project demonstrates how agentic workflows, schema awareness, and multi-stage LLM coordination foster a robust, trustworthy natural-language interface to data exploration.

18] REFERENCES:

1.] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., ... & Chen, W. (2021). LoRA: Low-Rank Adaptation of Large Language Models. arXiv preprint arXiv:2106.09685.

2.] Katsogiannis-Meimarakis, G., & Koutrika, G. (2023). A survey on deep learning approaches for text-to-SQL. The VLDB Journal, 32, 905–936. https://doi.org/10.1007/s00778-022-00776-8

3.] Morales, F. (2024). Towards Robust and Interpretable Text-to-SQL Generation: A Mistral-based Approach with Comprehensive Evaluation. (Working Paper/Preprint – confirm publication details if available).

4.] Pourreza, M., & Rafiei, D. (2023). Din-sql: Decomposed in-context learning of text-to-sql with self-correction. arXiv preprint arXiv:2305.11870.

5.] Sun, S., Zhang, Y., Yan, J., Gao, Y., Ong, D., Chen, B., & Su, J. (2023). Battle of the Large Language Models: Dolly vs LLaMA vs Vicuna vs Guanaco vs Bard vs ChatGPT - A Text-to-SQL Parsing Comparison. arXiv:2310.10190v1 [cs.CL].

6.] Yu, T., Zhang, R., Yang, K., Wang, D., Li, Z., Yasunaga, M., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., & Radev, D. R. (2019). Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. arXiv:1809.08887v5 [cs.CL].

7.] Zhan, Z., Haihong, E., & Song, M. (2025). Leveraging Large Language Model for Enhanced Text-to-SQL Parsing. IEEE Access, 13, 30497-30504. (Note: Using publication year from the PDF, though it appears to be a future date. Confirm actual publication date if possible). https://doi.org/10.1109/ACCESS.2025.3540072