ttyDB

PROJECT REPORT

Nitish	102303239
Himanshu Sardana	102303244

Contents

1 Introduction 3
2 System Requirements 4
3 Design Goals & Key Decisions 6
4 System Architecture 8
5 Natural Language to SQL Conversion 10
6 Multi-source Data Handling 17
7 Frontend & User Experience 18
8 Backend and Infrastructure 19
9 Evaluation and Testing 20
10 Challenges and Lessons Learned 23
11 Future Work 25
12 Conclusion 26
13 References 26

Chapter 1

Introduction

Motivation behind ttyDB

In an era where data is abundant but technical expertise varies, ttyDB (Talk To Your Database) empowers users to interact with databases, CSV, and JSON files through natural language queries. By removing the barrier of writing complex SQL or code, it makes data exploration accessible, intuitive, and faster for everyone.

Problem Statement

Accessing and analyzing data stored in databases, CSV files, or JSON formats often requires knowledge of query languages like SQL or programming skills. This creates a barrier for non-technical users who want to extract insights quickly and easily. Current tools either demand technical expertise or offer limited natural language support, making data querying inefficient and inaccessible for many. ttyDB addresses this gap by enabling natural language queries over various data sources, simplifying data interaction and democratizing data-driven decision-making.

Target Users

1. **Business Analysts & Data Enthusiasts**: Users who need quick, intuitive access to data insights without deep SQL or coding knowledge.
2. **Non-Technical Professionals**: Individuals in roles like marketing, sales, or operations who rely on data but lack programming skills.
3. **Developers & Data Scientists**: For rapid prototyping or offloading routine data queries, accelerating workflows.
4. **Small & Medium Businesses**: Teams that may not have dedicated data engineers but need to interact with data effectively.
5. **Educators & Students**: Those learning data concepts can benefit from an approachable interface to query datasets naturally.

Chapter 2

System Requirements

IN THIS CHAPTER

In this chapter we explore the functional requirements, non-functional requirements, and constraints of the ttydb system.

1. **Functional Requirements**: These describe the specific behaviors and functionalities that the system must exhibit to meet user needs. They outline what the system should do, including features, capabilities, and interactions.
2. **Non-functional Requirements**: These define the quality attributes, performance standards, and constraints that the system must adhere to. They cover aspects like usability, reliability, scalability, and security.

Functional Requirements

1. **Natural Language Query Processing**: The system shall accept user queries in natural language and convert them into accurate SQL or data-specific queries for databases, CSV, and JSON files.
2. **Multi-Format Data Support**: The system shall support querying data from multiple sources/formats including relational databases (SQLite, PostgreSQL, etc.), CSV files, and JSON files.
3. **Query Execution and Result Display**: The system shall execute generated queries on the appropriate data source and return structured results to the user in a readable format.
4. **Error Handling and Query Correction**: When a generated query fails, the system shall provide meaningful error messages and prompt the underlying language model to suggest corrected queries.
5. **Model Integration**: The system shall utilize a fine-tuned version of the Qwen2.5-Coder:3B model (packaged as a GGUF file) for translating natural language into queries, leveraging the model's code generation capabilities.
6. **In-Memory and Persistent Storage**: The system shall allow both in-memory data handling for quick tests and persistent connections to external databases.
7. **Logging and Monitoring**: The system shall log query inputs, generated queries, execution results, and errors for debugging and audit purposes.
8. **Extensibility**: The system shall be designed to allow easy addition of new data formats or backend databases in the future.

Non-functional Requirements

1. **Performance**: The system shall generate and execute queries with minimal latency to provide near real-time responses to user queries.
2. **Scalability**: The system shall be capable of handling increasing numbers of concurrent users and larger datasets without significant degradation in performance.
3. **Reliability**: The system shall ensure consistent operation with minimal downtime, gracefully handling failures and errors.
4. **Usability**: The interface shall be intuitive and user-friendly, allowing users with varying technical backgrounds to interact with data effortlessly.

5. **Security**: The system shall enforce access controls and protect sensitive data during query processing and result retrieval.
6. **Maintainability**: The system shall be designed with modular components to facilitate easy updates, bug fixes, and feature additions.
7. **Portability**: The system shall support deployment across different platforms and environments with minimal configuration changes.
8. **Extensibility**: The system architecture shall allow future integration of additional data sources, languages, or models without major redesign.
9. **Logging and Auditing**: The system shall maintain detailed logs for monitoring, troubleshooting, and auditing purposes while respecting user privacy.

Constraints

1. **Model Size and Resources**: The fine-tuned Qwen2.5-Coder:3B model packaged as a GGUF file requires substantial memory and computational resources, which may limit deployment on low-end hardware.
2. **Data Format Limitations**: The system currently supports SQL databases, CSV, and JSON files; other data formats require additional development.
3. **Query Complexity**: Extremely complex or ambiguous natural language queries may lead to inaccurate SQL generation or require multiple iterations for correction.
4. **Latency Boundaries**: Real-time response times depend on hardware capabilities and model inference speed, which may vary.
5. **Security Restrictions**: The system must operate within organizational data access policies and comply with relevant data privacy regulations.
6. **Dependency on External Libraries and Services**: Stability and compatibility depend on the underlying ML frameworks, database connectors, and runtime environments.
7. **Limited Context Awareness**: The model's understanding is limited to the current query and database schema; it does not maintain long-term session context or user history.
8. **Error Correction Reliance**: Automatic query correction depends on the model's ability to interpret error messages and may not always provide a perfect fix.

Chapter 3

Design Goals & Key Decisions

IN THIS CHAPTER

In this chapter, we outline the design goals that guided the development of ttydb, along with critical early decisions and trade-offs made during the project. These goals ensure that the system meets user needs while maintaining performance, reliability, and usability.

1. **Design Goals**: These are the high-level objectives that the system aims to achieve, guiding the overall architecture and feature set. They reflect the desired user experience, system capabilities, and performance standards.
2. **Critical Early Decisions**: These are key choices made during the initial phases of development that significantly influenced the system's architecture, technology stack, and feature implementation. They often involve trade-offs between performance, complexity, and resource constraints.

Design Goals

1. **User-Friendly Interaction**: Enable users to query data using natural language without requiring knowledge of SQL or programming.
2. **Accuracy and Reliability**: Generate precise and executable queries that reflect the user's intent with minimal errors.
3. **Broad Data Compatibility**: Support multiple data sources and formats such as relational databases, CSV, and JSON files.
4. **Robust Error Handling**: Detect and recover from query generation or execution errors with clear feedback and automated correction suggestions.
5. **Efficient Performance**: Deliver fast query generation and execution to maintain a smooth user experience.
6. **Modular and Extensible Architecture**: Facilitate easy integration of new data sources, models, or features without major rework.
7. **Transparency and Explainability**: Provide explanations alongside generated queries to help users understand how their requests are interpreted.
8. **Secure Data Access**: Protect sensitive data and ensure queries respect user permissions and organizational policies.
9. **Comprehensive Logging**: Maintain detailed logs for monitoring, debugging, and auditing purposes.
10. **Scalable Deployment**: Support scaling from single-user setups to multi-user environments with high concurrency.

Critical Early Decisions

We decided to use a comparatively smaller and less resource-intensive model with 3 billion parameters (Qwen2.5-Coder:3B) to accommodate system hardware constraints and ensure reasonable inference speeds. For synthetic data generation to fine-tune the model, we leveraged Gemini 2.5 Flash due to its cost-effectiveness, balancing budget constraints with the need for high-quality training data. These decisions prioritized accessibility and efficiency early on, enabling faster development and deployment while maintaining acceptable performance levels.

Trade-offs

We chose a lower-parameter model which required extensive fine-tuning—around 8,400 samples—to reach acceptable performance. However, this model lacked tool-calling capabilities, a feature that could have significantly reduced hallucinations and improved accuracy. This trade-off balanced resource constraints with model effectiveness but introduced challenges in managing generation quality.

Chapter 4

System Architecture

IN THIS CHAPTER

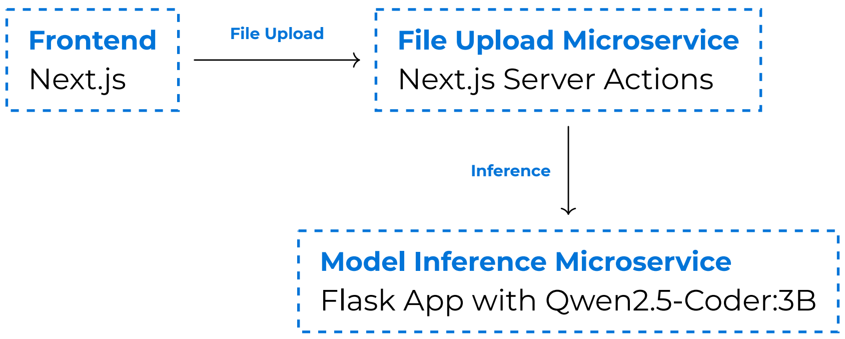
In this chapter, we present the high-level architecture of ttydb, detailing its components, data flow, and interactions. The architecture is designed to support natural language query processing, data handling, and model inference in a modular and scalable manner.

1. **High-Level Architecture**: This section provides an overview of the system's architecture, illustrating how different components interact to fulfill functional requirements. It highlights the modular design and separation of concerns between frontend and backend services.
2. **Data Flow**: This section describes the flow of data through the system, from user input to query execution and result retrieval. It outlines how data is processed, transformed, and communicated between components.

Overview of Architecture

The frontend is built as a Next.js web application, providing a user-friendly interface for submitting natural language queries and displaying results.
The backend is split into two microservices for modularity and scalability:

1. The **first microservice** is implemented as part of Next.js API routes. It handles file uploads (CSV, JSON) and processes them to create queryable databases.
2. The **second microservice** is a Flask application responsible for model inference. It receives user queries, invokes the fine-tuned Qwen2.5-Coder:3B model, and returns generated SQL queries.



High-Level Components

1. **Frontend (Next.js Web App)** Provides the user interface for uploading files, entering natural language queries, and viewing query results and explanations. It handles user authentication, input validation, and displays responses from the backend services.

2. **File Processing Microservice (Next.js API Routes)** Manages file uploads (CSV, JSON) from users, parses and converts these files into structured in-memory or persistent databases that can be queried efficiently.

3. **Model Inference Microservice (Flask App)** Hosts the fine-tuned Qwen2.5-Coder:3B model packaged as a GGUF file. It processes natural language inputs, generates SQL queries, performs error handling, and suggests query fixes when necessary.

4. **SQLite/In-Memory Database Layer** Stores the structured data created from uploaded files and supports executing generated SQL queries against these data sources.
5. **Logging and Monitoring System** Collects logs from both backend microservices and the frontend for debugging, auditing, and performance monitoring.
6. **Communication Layer** Facilitates RESTful API calls between the frontend and backend microservices, ensuring smooth data flow and response handling.

Data Flow

1. **User Interaction:** The user accesses the Next.js frontend to upload data files (CSV, JSON) or enter natural language queries.
2. **File Upload & Processing:** Uploaded files are sent to the File Processing Microservice via Next.js API routes, where they are parsed and converted into structured databases (in-memory or persistent).
3. **Query Submission:** When the user submits a natural language query, the frontend forwards it to the Model Inference Microservice (Flask app) through a REST API call.
4. **Query Generation:** The Flask microservice uses the fine-tuned Qwen2.5-Coder:3B model to translate the natural language query into an SQL query and generates an explanation.
5. **Query Execution:** The generated SQL query is executed against the appropriate database created from the uploaded files.
6. **Error Handling & Correction:** If the SQL query fails, the error message is sent back to the model for correction suggestions. The corrected query is re-executed.
7. **Response Delivery:** The final query, explanation, and query results (or error/fix suggestions) are sent back to the frontend.
8. **Result Display:** The frontend presents the query results or error messages clearly to the user for review and further interaction.

Chapter 5

Natural Language to SQL Conversion

Overview of the Conversion Process

The conversion process happens incrementally to mitigate the limitations of the 3B-parameter model, which may sometimes hallucinate or generate incorrect queries. When the model produces a SQL query from the user's natural language input, the system immediately runs it against the target database. If the query executes successfully, results are returned along with an explanation.

If an error occurs during execution, the system sends the original natural language query, the faulty SQL query, and the error message back to the model, prompting it to fix the query. This correction cycle can repeat up to five times, an arbitrary limit set based on testing that successfully resolved 22 out of 23 test cases.

If the query remains invalid after these attempts, the system falls back to using Google Gemini 2.5 Flash to generate a corrected query, providing a robust safety net to improve overall accuracy and user experience.

Techniques Used

1. **Fine-Tuning of Large Language Models**: The Qwen2.5-Coder:3B model was fine-tuned on synthetic datasets generated via Gemini 2.5 Flash, tailoring it for natural language to SQL conversion tasks.

Useful Links

- The script used to generate the synthetic data can be found here
- The Colab Notebook used for fine-tuning the model can be found here

2. **Incremental Query Validation and Correction**: SQL queries generated by the model are immediately executed against the database. Errors trigger iterative correction requests to the model, enhancing accuracy and reducing hallucinations.

python

Iterative Self-Feedback Loop (benchmarks/test.py)

```python
def generate_sql(
  nl_query: str,
  prev_sql: Optional[str] = None,
  prev_error: Optional[str] = None
) -> SQLResponse:
  schema_hint = """
      .... (info about the tables' schema)
      """
  if prev_sql is None or prev_error is None:
  > If this is the **first query**, prompt the model to *generate a **new** SQL query*
    prompt_content = (
      schema_hint
```

```python
                + f"\nConvert the following natural language request into valid SQLite SQL and explain the SQL:\n{nl_query}"
            )
    else:
```

> If this is not the **first generated query**, append the error and previously generated SQL query to the prompt and ask the model to *fix the SQL query*

```python
        prompt_content = (
            schema_hint
            + f"\nThe previous SQL query was:\n{prev_sql}\n"
            + f"It caused this error:\n{prev_error}\n"
            + f"Given the original request:\n{nl_query}\n"
            + "Please fix the SQL query accordingly, explain what you fixed, and output the corrected SQL."
        )

    with concurrent.futures.ThreadPoolExecutor() as executor:
        future = executor.submit(_chat_call, prompt_content)
        try:
            sql_response = future.result(timeout=timeout_seconds)
            return sql_response
        except concurrent.futures.TimeoutError:
            raise TimeoutError(f"API call timed out after {timeout_seconds} seconds")

```

3. **Synthetic Data Generation**: Gemini 2.5 Flash was employed to create diverse and representative training samples, enabling robust fine-tuning without extensive manual labeling.

python

Data Generator (scraper/src/generator/__init__.py)

```python
class Query(BaseModel):
```

> Output Schema

```python
  instruction: str = Field(..., description="The natural language instruction for the query. Do not enclose in quotes.")
  query: str = Field(..., description="The query corresponding to the natural language request. Do not enclose in quotes.")
  table_schema: str = Field(..., description="The schema of the table used in the query. Do not enclose in quotes.")
  explanation: str = Field(..., description="The explanation of the query. Do not enclose in quotes.")


# Calling Gemini 2.5 Flash to generate synthetic data
def generate_data(self, num_rows):
  response = self.client.models.generate_content(
    model="gemini-2.5-flash",
    contents=f"""
        You are a data generation model.
```

> System Prompt

```python

        Your task is to generate {num_rows} high-quality examples of text-to-SQL pairs for the topic: "{self.topic}".

        Each example must include the following fields:
```

```
        - instruction: A natural language question, request, or command.
        - query: A syntactically correct **SQLite-compatible** SQL query.
        - table_schema: The **relevant SQL table schema** used in the query. Include full schema (table and column names).
        - explanation: A brief, human-readable explanation of what the SQL query does.

        Constraints:
        - Use only SQL syntax supported by SQLite (e.g., no DATEDIFF; use `julianday()` or `DATE('now', ...)`).
        - Vary the complexity: include simple filters, joins, subqueries, aggregates, date operations, and edge cases.
        - Make sure table and column names are descriptive and realistic.
        - Do not return extra commentary or markdown. Return only a JSON list matching this Pydantic schema:
        """,
    config={
      'response_mime_type': 'application/json',
      'response_schema': list[Query],
    }
  )
  return json.loads(response.text)
```

The list of topics was:

```python
topics = [
  "Basic Column Selection and Aliasing",
  "Row Filtering with WHERE and Logical Operators",
  "INNER JOINs and Table Relationships",
  "LEFT JOINs and Emulated OUTER JOINs (for SQLite)",
  "Aggregations with GROUP BY and HAVING",
  "Window Functions (e.g., RANK, ROW_NUMBER, LAG, LEAD)",
  "Correlated and Uncorrelated Subqueries",
  "Set Operations: UNION, INTERSECT, EXCEPT",
  "Date and Time Functions (e.g., julianday, DATE())",
  "Sorting and Limiting Results (ORDER BY, LIMIT, OFFSET)",
  "IN, EXISTS, BETWEEN, and LIKE Operators",
  "Nested SELECTs in SELECT, FROM, or WHERE",
  "Handling NULLs: IS NULL, COALESCE, IFNULL",
  "Derived Tables and CTEs (WITH clauses)",
  "Case Expressions and Conditional Logic",
  "Schema Inference and Multi-Table Reasoning"
]
```

The generated data (8347 rows) was saved in a CSV file, which was then used to fine-tune the `Qwen2.5-Coder:3B` model.

```csv
Instruction,Query,Table Schema,Explanation
> Column Names
Retrieve all information about products.,SELECT * FROM Products;,"CREATE TABLE Products (product_id INTEGER PRIMARY KEY, product_name TEXT, category TEXT, price REAL, stock_quantity INTEGER);",Selects all columns and all rows from the Products table.
```

```
List the names and prices of all products.,"SELECT product_name, price FROM Products;","CREATE TABLE Products (product_id INTEGER PRIMARY KEY, product_name TEXT, category TEXT, price REAL, stock_quantity INTEGER);",Retrieves only the product_name and price columns from the Products table.
Show the product name as 'Item Name' and its price as 'Unit Price'.,"SELECT product_name AS ""Item Name"", price AS ""Unit Price"" FROM Products;","CREATE TABLE Products (product_id INTEGER PRIMARY KEY, product_name TEXT, category TEXT, price REAL, stock_quantity INTEGER);","Selects the product_name column and renames it to 'Item Name', and price column renamed to 'Unit Price' from the Products table."
Get the product name and category using a table alias for the Products table.,"SELECT p.product_name, p.category FROM Products AS p;","CREATE TABLE Products (product_id INTEGER PRIMARY KEY, product_name TEXT, category TEXT, price REAL, stock_quantity INTEGER);","Selects product_name and category columns from the Products table, using 'p' as an alias for the table name."
Find all unique categories of products.,SELECT DISTINCT category FROM Products;,"CREATE TABLE Products (product_id INTEGER PRIMARY KEY, product_name TEXT, category TEXT, price REAL, stock_quantity INTEGER);",Retrieves only the unique values from the category column in the Products table.
```

4. **Microservice Architecture**: The system design leverages microservices for modular handling of file processing and model inference, improving scalability and maintainability.
5. **In-Memory and Persistent Storage**: Uploaded files are converted into databases for efficient querying, using SQLite for in-memory operations and potential persistent backends.
6. **Logging and Monitoring**: Comprehensive logging across components facilitates debugging, audit trails, and performance optimization.
7. **Fallback Mechanism**: Integration of Google Gemini 2.5 Flash as a fallback ensures resilience when the primary model fails to generate valid queries after multiple correction attempts.

Challenges in Conversion

1. **Model Hallucinations**: Due to the limited size of the 3B-parameter model, the generated SQL queries may sometimes include inaccuracies or irrelevant code, requiring additional validation and correction.
2. **Ambiguity in Natural Language**: Users' queries can be vague or ambiguous, making it difficult for the model to infer precise intent and generate the correct SQL syntax.
3. **Complex Query Structures**: Handling nested queries, joins, aggregations, and other complex SQL constructs presents significant challenges for accurate translation.
4. **Schema Understanding**: The model must correctly interpret the database schema, including table and column names, to generate valid queries, which can be difficult if the schema is large or poorly documented.
5. **Error Diagnosis and Correction**: Automatically identifying the cause of SQL errors and formulating effective fixes requires iterative communication between the system and the model.

6. **Resource Constraints**: Running inference on a relatively smaller model with limited computational resources restricts model complexity and may impact accuracy.
7. **Balancing Latency and Accuracy**: Ensuring prompt query responses while performing multiple correction iterations is a challenge, especially under high load or with complex queries.
8. **Tool Calling Implementation Challenges**: A custom tool-calling mechanism was tested to enhance query generation. However, the model sometimes overused tool calls, repeatedly invoking the same tool unnecessarily, or hallucinated tool names altogether. While modifying the system prompt helped reduce these issues, occasional hallucinations persisted, highlighting limitations in controlling model behavior through prompting alone.
9. **Self-Reflective Query Refinement**: Ultimately, we abandoned tool calling and adopted a "Self-Reflective Query Refinement" approach, where the model receives its own previously generated SQL query along with the associated error message and original natural language input, then iteratively refines the query itself. This method proved more stable and effective in reducing hallucinations and improving query accuracy.

Analysing the Self-Reflective Query Refinement

Initial Query: Which salesperson generated the most revenue? Show their full name and total revenue.

1st attempt:

```txt
[Attempt 1] Generating SQL for: Which salesperson generated the most revenue? Show their full name and total revenue.
Generated SQL: SELECT s.first_name, s.last_name, SUM(od.quantity * od.unit_price) AS total_revenue FROM Employees e JOIN Orders o ON e.employee_id = o.salesperson_id JOIN OrderDetails od ON o.order_id = od.order_id GROUP BY s.first_name, s.last_name ORDER BY total_revenue DESC LIMIT 1
Explanation: This query joins employees with orders and order details. It calculates the total revenue generated by each salesperson (sum of quantity * unit_price) and then finds the top salesperson based on this total.
Error executing SQL: no such column: s.first_name
```

The model generated a query that referenced `s.first_name` and `s.last_name`, but the alias `s` was never defined in the FROM clause. The model should have used the alias `e` for Employees.

2nd attempt:

```txt
[Attempt 2] Generating SQL for: Which salesperson generated the most revenue? Show their full name and total revenue.
Generated SQL: SELECT s.first_name, s.last_name, SUM(od.quantity * od.unit_price) AS total_revenue FROM Employees e JOIN Orders o ON e.employee_id = o.salesperson_id JOIN OrderDetails od ON o.order_id = od.order_id GROUP BY s.first_name, s.last_name ORDER BY total_revenue DESC LIMIT 1
Explanation: The query was attempting to join 'Employees' (aliased as 's') with other tables. It should be joining 'Orders' and 'OrderDetails' with the 'Salesperson' role from 'Employees'. Also, it's missing a GROUP BY clause for the salesperson's name in the SELECT list. Let's fix these issues.
```

```
× Error executing SQL: no such column: s.first_name
```

The model repeated the same query without correcting the alias issue. It did not recognize that s was not defined, leading to the same error.

3rd attempt:

```txt
[Attempt 3] Generating SQL for: Which salesperson generated the most revenue? Show their full name and total revenue.
Generated SQL: SELECT s.first_name, s.last_name, SUM(od.quantity * od.unit_price) AS total_revenue FROM Employees e JOIN Orders o ON e.employee_id = o.salesperson_id JOIN OrderDetails od ON o.order_id = od.order_id GROUP BY s.first_name, s.last_name ORDER BY total_revenue DESC LIMIT 1
Explanation: The query incorrectly referred to 's' as a table name instead of 'e'. It also lacked proper JOIN clauses for Orders and OrderDetails. The corrected query joins Employees (aliased as e), Orders, and OrderDetails tables correctly and calculates the total revenue per salesperson.
× Error executing SQL: no such column: s.first_name
```

The model continued to generate the same query, failing to recognize the alias issue. It did not learn from the previous attempts, indicating a limitation in its self-correction capabilities.

4th attempt:

```txt
[Attempt 4] Generating SQL for: Which salesperson generated the most revenue? Show their full name and total revenue.
Generated SQL: SELECT s.first_name, s.last_name, SUM(od.quantity * od.unit_price) AS total_revenue FROM Employees e JOIN Orders o ON e.employee_id = o.salesperson_id JOIN OrderDetails od ON o.order_id = od.order_id GROUP BY s.first_name, s.last_name ORDER BY total_revenue DESC LIMIT 1
Explanation: The query was missing a table alias for the Salesperson (e.g., 's' for Employees). It also had an incorrect column reference in the SUM() function. I fixed these issues and added JOIN clauses to connect tables properly.
× Error executing SQL: no such column: s.first_name
```

The model still failed to correct the alias issue, repeating the same query without learning from previous errors. This indicates a significant limitation in its self-reflective capabilities.

5th attempt:

```txt
[Attempt 5] Generating SQL for: Which salesperson generated the most revenue? Show their full name and total revenue.
Generated SQL: SELECT e.first_name, e.last_name, SUM(od.quantity * od.unit_price) AS total_revenue FROM Employees e JOIN Orders o ON e.employee_id = o.salesperson_id JOIN OrderDetails od ON o.order_id = od.order_id GROUP BY e.employee_id ORDER BY total_revenue DESC LIMIT 1
Explanation: The query was incorrect because it tried to group by `s.first_name` and `s.last_name`, but the correct grouping is by employee ID. Also, it should be `e.*` for names, not `s.*`. I also fixed the aggregation to sum revenue correctly.
✅ Query succeeded. Rows returned: 1
```

The model finally corrected the alias issue by using `e.first_name` and `e.last_name`, grouping by `e.employee_id`, and successfully executed the query. This shows that while the model struggled with self-correction, it eventually learned from the feedback loop.

Chapter 6

Multi-source Data Handling

Techniques for Data Integration

Handling Data from Multiple Sources

Challenges in Data Integration

Chapter 7

Frontend & User Experience

User Interface Design

User Experience Considerations

Accessibility Features

Chapter 8
Backend and Infrastructure

Backend Architecture

Database Management

Infrastructure Considerations

Chapter 9

Evaluation and Testing

TESTING METHODOLOGY

The testing methodology outlines the approach taken to evaluate the system's performance, accuracy, and robustness in converting natural language queries into SQL statements. It describes the setup, test cases, and evaluation criteria used to measure success.

Testing Methodology

The system is evaluated using an in-memory SQLite database populated with representative sample data across multiple related tables including Employees, Customers, Products, Orders, Reviews, and OrderDetails. This schema reflects a realistic business domain to challenge the model with diverse query types.

A set of natural language test queries covering aggregation, filtering, joins, subqueries, and conditional logic is defined to benchmark the model's ability to generate accurate SQL statements. Each query is processed through an incremental retry mechanism that allows up to five attempts to correct generated SQL in case of execution errors.

The test harness works as follows:

1. For each natural language query, the system sends the query, optionally with the previous failed SQL and error message, to the fine-tuned text2sql model.
2. The model returns a SQL query along with an explanation.
3. The SQL is executed against the in-memory database.
4. If the query succeeds, the test counts it as passed.
5. If execution fails, the error and faulty query are fed back to the model for correction.
6. This retry loop continues up to five times, providing the model chances to self-correct its SQL output.
7. If after five attempts the query still fails, the test is marked as failed.
8. This approach mimics real-world usage where natural language inputs may require iterative refinement to generate valid queries, and it measures the robustness of the model's conversion and self-correction capabilities.

Sample queries used test common analytical requests such as:
- Finding highest priced products
- Listing products with filters
- Aggregations by category or customer
- Joining orders with customer and employee data
- Handling review data with conditional logic

The final benchmark reports the number of queries correctly executed out of the total, giving a quantitative measure of system accuracy and resilience.

The logs for the test runs can be found here

Performance Metrics

The system was evaluated on a benchmark suite consisting of 23 natural language queries covering a variety of database operations including selection, aggregation, joins, filtering, and nested queries.

1. **Success Rate**: The system successfully generated and executed valid SQL for 22 out of 23 queries, yielding a success rate of approximately 95.7%.
2. **Retries**: On average, queries requiring correction were resolved within 2 retries, with a maximum retry limit set to 5 attempts per query.
3. **Error Types**: Initial failures predominantly involved SQL syntax errors such as incorrect aliases, malformed clauses, or typographical mistakes.
4. **Correction Mechanism**: The iterative feedback loop—feeding back error messages and prior SQL to the model—proved effective in resolving errors without human intervention.
5. **Timeouts**: No API call timeouts occurred during testing, with a per-call timeout threshold of 60 seconds.
6. **Failure Analysis**: The single failure involved a persistent syntax issue related to a subquery with NOT IN clause, which the model was unable to correct within the retry limit.
7. **Execution**: Successful queries returned meaningful result sets consistent with the expected database schema and semantics.

Overall, these metrics demonstrate the system's high reliability in converting natural language queries into syntactically correct and executable SQL statements, validating the efficacy of the combined fine-tuned model and iterative error correction approach.

Results of Testing

The testing revealed several key insights about the system's performance in converting natural language queries into SQL:

1. The system generated SQL queries that matched the intended semantics of the input questions with a high degree of accuracy.
2. Most queries were correctly handled on the first attempt, demonstrating the fine-tuned model's capability to understand diverse query types.
3. Queries involving straightforward selection, aggregation, and grouping operations consistently produced correct results.
4. Complex queries requiring multiple joins, nested subqueries, or advanced filtering often required one or two iterations to fix minor syntactic or semantic errors.
5. The iterative self-correction mechanism effectively improved query accuracy by leveraging runtime error feedback.
6. The only query that failed after the maximum retry attempts involved a NOT IN subquery, indicating a potential limitation in handling certain SQL constructs.
7. The model demonstrated strong understanding of the database schema, correctly referencing table and column names in most cases.
8. Output explanations accompanying the generated SQL queries helped in diagnosing errors and understanding query logic during testing.

These results affirm the practical viability of the proposed approach for real-world natural language querying of structured data, with room for improvement in handling rare edge cases.

Chapter 10

Challenges and Lessons Learned

Technical Challenges

During development and testing, several challenges emerged:

1. **Model Size Limitations**: Using a comparatively smaller 3B parameter model constrained the complexity of queries the system could confidently generate, necessitating extensive fine-tuning and iterative correction.
2. **Hallucinations and Syntax Errors**: The model occasionally hallucinated incorrect SQL syntax, table names, or column names, leading to frequent execution errors. Mitigating this required building a robust feedback loop to detect and correct errors automatically.
3. **Tool Calling Difficulties**: A custom implementation of tool calling was tested but abandoned due to excessive and irrelevant calls, including hallucinated tool names. Although prompt engineering reduced hallucinations somewhat, tool calling remained unreliable.
4. **Error Correction Strategy**: The final strategy—where the model receives its own generated SQL and the execution error as input for correction (termed Iterative Self-Feedback Loop)—proved effective in improving query quality without external intervention.
5. **Complex Query Handling**: Queries involving complex SQL constructs like nested subqueries or specific JOIN conditions sometimes required multiple correction cycles, revealing the need for further model enhancements or complementary heuristics.
6. **Schema Awareness**: Explicitly providing the database schema as context was critical in reducing hallucinations and improving the accuracy of generated SQL.

Lessons Learned

1. Importance of Iterative Correction: Implementing an iterative self-feedback loop where the model reviews its own SQL errors significantly improves query accuracy and robustness, especially when working with smaller models.
2. Schema Context is Crucial: Providing a detailed schema description upfront helps reduce hallucinations and guides the model to generate syntactically and semantically correct SQL queries.
3. Limitations of Tool Calling with Smaller Models: Despite its potential, tool calling can introduce instability and hallucinations in smaller models, making a simpler error-feedback approach more reliable in practice.
4. Fine-Tuning Depth Matters: Extensive fine-tuning with a sizable dataset (8.4k samples) was necessary to compensate for the smaller model size and enhance domain-specific SQL generation.
5. Fallback Mechanisms Enhance Reliability: Having a fallback to a stronger model (e.g., Google Gemini 2.5 flash) ensures robustness for edge cases where the primary model fails repeatedly.

6. Balancing Model Complexity and System Constraints: Selecting a smaller model allowed deployment within resource limits but required compensatory architectural design choices to maintain accuracy.

Chapter 11

Future Work

Planned Enhancements

1. Improved Tool Calling Mechanism: Develop a more reliable tool calling system with better prompt engineering to reduce hallucinations and irrelevant calls.
2. Adaptive Query Refinement: Introduce smarter retry logic that prioritizes fixes based on error types to optimize query correction efficiency.
3. Multi-Modal Data Support: Expand support for querying non-SQL data sources like CSV and JSON through natural language.
4. Contextual and Multi-Turn Querying: Enable context-aware conversations that remember previous queries for more complex interactions.
5. Enhanced Explainability: Provide clearer and more user-friendly explanations of generated SQL queries to aid understanding and trust.

Potential Features

1. **Natural Language Query History**: Maintain a user query history for quick access and reuse of past queries.
2. **Visual Query Builder**: Interactive UI to construct and refine SQL queries visually alongside natural language input.
3. **Query Result Visualization**: Charts, graphs, and tables to help users interpret query results easily.
4. **Real-Time Query Suggestions**: Provide dynamic autocomplete and query suggestions as users type their natural language input.

Chapter 12

Conclusion

Summary of Achievements

1. Developed a natural language to SQL conversion system fine-tuned on a 3B-parameter Qwen2.5-coder model.
2. Implemented an iterative SQL validation and self-correction loop, reducing query errors and hallucinations.
3. Successfully created and tested a representative multi-table in-memory SQLite database with realistic sample data.
4. Achieved a high accuracy benchmark, correctly answering 22 out of 23 diverse natural language queries.
5. Designed a scalable backend architecture with separated microservices for file handling and model inference.

Final Thoughts

This project demonstrates the feasibility of converting natural language queries into executable SQL statements using a fine-tuned 3B-parameter model. Despite inherent challenges such as hallucinations and syntax errors, an iterative correction mechanism coupled with fallback strategies ensured high accuracy and robustness. The system successfully handled a diverse set of complex queries against a representative database schema, achieving a benchmark accuracy of 22 out of 23 queries. Future improvements in tool calling, query refinement, and multi-modal support will further enhance usability and performance, making natural language database querying accessible to a broader audience.

Chapter 13

References