DATA GOVERNANCE
MANAGING UNSTRUCTURED DATA

OST
Eastern Switzerland
University of Applied Sciences

Text-to-SQL for DataGovernance Technologies and Education

SA, BA

Fiona Pichler, Benjamin Kern

Text-to-SQL for DataGovernance Technologies and Education
– A Proof of Concept

BSc. Computer Science
Eastern Switzerland University of Applied Sciences
Switzerland
January 10, 2025

Imprint

Contact and Authors:

Author Name:	Fiona Pichler
Contact Information:	Fiona.Pichler@ost.ch
University	OST

Author Name:	Benjamin Kern
Contact Information	Benjamin.Kern@ost.ch
University	OST

Document Reference URL:

Documentation	Documentation
Code	Code
Kanban Board	Kanban Board
Dashboard	Dashboard

Abstract

This project explores the development of a Natural Language to SQL (NL-to-SQL) system using Large Language Models (LLMs). The primary objective is to research different approaches of generating SQL from NL and evaluate their feasibility. Subsequently a PoC was implemented to demonstrate real-world usefulness.

The research found four key approaches which were evaluated: pure LLM, in-context learning, fine-tuning the LLM, and Retrieval-Augmented Generation (RAG). Fine-tuning was ruled out due to an insufficient amount of training data and time. The remaining three approaches were implemented in Python and relevant LLM APIs and thoroughly evaluated.

Key findings indicate that while pure LLM approaches and in context learning provide a baseline, RAG significantly enhances the accuracy and reliability of the generated queries. The results of the test queries were evaluated in terms of 1. similarity (how similar is an output compared to the example solution?), 2. validity (is the output valid SQL?), 3. executability (can the output be executed and are the generated column names correct?), 4. reliability (how similar is the output to the same user prompt?).

Testsets were divided into two grades: basic and advanced, based on its complexity. For the advanced test cases, on average 45% can be executed on the databases, with only 70% of the requested columns extracted. Hallucinations could not be completely eliminated when extending the scope of the context, resulting in the low number of executable SQL queries. In this thesis, the llama3.2 model was found to have the most potential for further development.

The PoC application demonstrates the feasibility of using RAG with metadata about database schemas as well as JSONL input of users to generate SQL from NL, offering a user-friendly interface for both technical and non-technical users.

Management Summary and Web Publication

0.1. Background

In recent years Large Language Models have conquered the classrooms. Students and teachers alike use them for support. Those models rely on huge amounts of data. They can also be used to process huge amounts of data. This leads to the idea to use them on databases. As data is of value it should be a basic skill to use SQL to query databases. The usage of LLM's can help to overcome a lack of this skill. NL-to SQL systems can use LLM's while providing more context and guidelines than the LLM's themselves. Various benchmarks to evaluate the performance of NL-to-SQL systems already exist and with LLM's getting smaller and more powerful the foundation for NL-to-SQL systems exist.

0.2. Objectives

The team aims to test different approaches on building an NL-to-SQL chatbot. A foundation for future product development should be laid, comparing different LLM's and evaluating the approaches as well as existing frameworks.

0.3. Approach

At the beginning of the project the focus lay on research to understand the current state-of-the-art for using LLM's and approaches on NL-to-SQL systems. This research identified four core approaches: pure LLM as a baseline, in-context learning, fine-tuning the LLM, and Retrieval- Augmented Generation (RAG). Fine-tuning was ruled out due to an insufficient amount of training data and time. The remaining three approaches were implemented in Python and relevant LLM APIs and thoroughly evaluated. Four mid-sized and one small LLM were selected for performance evaluation, including Mistral, phi3:3.8b, falcon7b, llama3.1:8b, and GPT-4o-mini. The decision to use rather smaller LLM's was based on the capabilities of homecomputers as well as the available hardware for this project. After the first test runs falcon7b had to be substituted as the hallucinations were too strong. Falcon7b was substitued with phi3:14b. Two datasets were selected: One was provided by DataGovernance Technologies (MS SQL Server) and for the OST use case the well-known Pagila dataset(PostgreSQL) was used. The test SQL queries and their corresponding NL texts were extracted from database lecture slides and others were generated using ChatGPT. They were divided into basic and advanced test cases. As a benchmark an existing NL-to-SQL system was evaluated as well, by name: vanna.ai. It turned out to need more training than considered at first.

0.4. Results

When comparing the LLM's most of the times gpto4-mini and llama were leading, while the smallest LLM phi3:3.8b could not be fully hindered from hallucinating.

The results of the test queries were evaluated in terms of 1. similarity (how similar is an answer compared to the example solution?), 2. validity (is the output valid SQL?), 3. executability (can the output be executed and are the generated column names correct?) 4. reliability (how similar are answers to the same question?). It proved difficult to evaluate similarity using the usual cosine function, as the same query output can be achieved with different queries. Since the pure LLM approach guesses table names, the similarity metric was still considered useful for comparison. While executability includes validity, the validity metric can be achieved without the correct table names, which is needed for comparison to the pure LLM approach. For the advanced test cases, on average 45% can be executed on the databases, with only 70% of the requested columns extracted. Getting the LLMs to output valid SQL without additional explanations was a challenge. The hallucinations could not be completely eliminated, resulting in the low number of executable SQL queries. The reliability metric in Fig. 3 shows the high number of hallucinations by phi3:3.8b. As probably the smallest LLM in these tests, it proved particularly difficult to produce valid SQL. This thesis found potential to be further exploited.

0.5. Outlook

This project has successfully laid the foundation for future advances in NL-to-SQL systems. Further work should focus on eliminating the hallucinations, as well as providing more fine-grained test metrics to analyse the LLM's shortcomings thoroughly. Emerging approaches left out by this thesis should be looked into by name multi-step reasoning and structured outputs.

0.6. Conclusion

This project successfully evaluated and compared different approaches by way of implementation of a testing scripts with benchmarks, which allows for easy comparison between different LLMs and approaches.

Furthermore, the project then successfully implemented these findings in a chatbot, which allows users to query databases with minimal technical understanding. For more tech-savvy users, it allows them to provide further context by way of uploading jsonl files with SQL to NL mapping. This allows the user to add information, that may not be contained in the database itself, such as domain specific terms or abbreviations.

For tech-illiterate users, the chatbot provides a user-friendly interface, where they can ask questions in natural language and receive SQL queries as well as the resultset in return - the only requirement being a direct connection to the database. Users do not have to worry about the underlying training logic, as the chatbot will automatically train on the database data.

It also allows the users to easily choose between LLMs, ensuring data privacy where only self-hosted LLMs are used.

Contents

0.1. Background . . . iii
0.2. Objectives . . . iii
0.3. Approach . . . iii
0.4. Results . . . iii
0.5. Outlook . . . iv
0.6. Conclusion . . . iv

I. Report 1

1. Introduction 2
1.1. Objectives . . . 2
1.2. Problem Statement . . . 2
1.3. Overview . . . 2

2. Assignment 3
2.1. Background . . . 3
2.2. Task . . . 3
2.3. Proposed Methodology . . . 3
2.4. Specifications/Constraints . . . 4
2.5. Deliverables . . . 4

3. Conditions and Constraints 5
3.1. Project Conditions . . . 5
3.2. Technical Constraints . . . 5
3.3. Operational Constraints . . . 5
3.4. Delivery Constraints . . . 5

4. Procedure and Structure of the Thesis 6
4.1. Introduction . . . 6
4.2. Problem and Task Introduction . . . 6
4.3. Procedure . . . 6
4.3.1. Step 1: Research and Requirements . . . 6
4.3.2. Step 2: Metrics . . . 7
4.3.3. Step 3: Testdata . . . 7
4.3.4. Step 4: large language models (LLMs) . . . 7
4.3.5. Step 5: Testscript Development . . . 7
4.3.6. Step 6: System Design and Implementation of the proof of concept (PoC) . 8
4.4. Participants Involved . . . 8
4.5. Summary . . . 8

5. State of the Art and Related Work 9
5.1. Introduction . . . 9
5.1.1. Purellm . . . 9
5.1.2. In Context Learning . . . 9

5.1.3. Finetuning . . . 9
5.1.4. RAG . . . 9
5.1.5. Structured Output . . . 10
5.1.6. Multi-step reasoning . . . 10
5.1.6.1. Decomposition of the Query . . . 11
5.1.6.2. Iterative Execution . . . 11
5.1.6.3. Logical Reasoning . . . 11
5.1.6.4. Intermediate Validation . . . 11
5.1.6.5. Use of Feedback . . . 11
5.1.7. Agent-like function calling . . . 11
5.1.7.1. Interpretation of User Query (Agent: Language Understanding Agent) 12
5.1.7.2. Identify Relevant Tables and Columns (Agent: Database Schema Agent) . . . 12
5.1.7.3. Determine Joins and Relationships (Agent: Relational Query Agent) 12
5.1.7.4. Handle Time Constraints or Filters (Agent: Condition Handling Agent) . . . 12
5.1.7.5. Construct Aggregations and Groupings (Agent: Aggregation Agent) 12
5.1.7.6. Generate SQL Code (Agent: Code Generation Agent) . . . 12
5.1.8. NLQ translation system . . . 13

6. Evaluation **14**
6.1. Introduction . . . 14
6.2. Evaluation Methodology . . . 14
6.2.1. Qualitative Criteria . . . 14
6.2.2. Quantitative Criteria . . . 14
6.3. Assessment of Results . . . 14
6.3.1. Data Collection and Analysis . . . 14
6.3.2. Evaluation Metrics . . . 14
6.4. Achievement of Objectives . . . 15
6.4.1. Objective Fulfillment . . . 15
6.4.2. Discrepancies and Unexpected Outcomes . . . 15
6.5. Future Research Directions . . . 15
6.6. Conclusion . . . 15

7. Project Vision and Implementation Strategy **16**
7.1. Project Vision . . . 16
7.2. Background . . . 16
7.3. Implementation Strategy . . . 16

8. Outcomes: Assessment and Objectives Fulfillment **17**
8.1. Assessment of Results . . . 17
8.2. Achievement of Objectives . . . 17
8.3. Conclusion . . . 17

9. Conclusions and Future Directions **18**
9.1. Conclusions . . . 18
9.2. Future Directions . . . 18
9.3. Final Thoughts . . . 18

II. SW Project Documentation 19

1. Overview 20
1.1. Purpose . . . 20
1.2. Content . . . 20
1.2.1. Vision . . . 20
1.2.2. Requirements Specification . . . 20
1.2.3. Design . . . 20
1.2.4. Implementation . . . 20
1.2.5. Test . . . 20
1.2.6. Result . . . 20
1.2.7. Further Development . . . 20
1.2.8. User Documentation . . . 20

2. Vision 21

3. Requirement Specification 22
3.1. Prioritization . . . 22
3.2. Use Case Diagram . . . 22
3.2.1. Actors . . . 23
3.2.1.1. Student . . . 23
3.2.1.2. Auditor . . . 23
3.2.1.3. Data Engineer . . . 24
3.2.1.4. Administrator . . . 24
3.3. Use Cases . . . 24
3.3.1. UC00: Setup and Configuration . . . 24
3.3.1.1. Main Success Scenario . . . 24
3.3.1.2. Alternate Scenarios . . . 24
3.3.2. UC01: Simplified Query Construction for Learning . . . 24
3.3.2.1. Main Success Scenario . . . 24
3.3.2.2. Alternate Scenarios . . . 24
3.3.3. UC02: Query Explanation for Educational Purposes . . . 25
3.3.3.1. Main Success Scenario . . . 25
3.3.3.2. Alternate Scenarios . . . 25
3.3.4. UC03: Compliance Data Extraction via NL . . . 25
3.3.4.1. Main Success Scenario . . . 25
3.3.4.2. Alternate Scenarios . . . 25
3.3.5. UC04: Assistance with Complex Query Generation . . . 25
3.3.5.1. Main Success Scenario . . . 25
3.3.5.2. Alternate Scenarios . . . 25
3.3.6. UC05: Query Optimization Suggestions . . . 26
3.3.6.1. Main Success Scenario . . . 26
3.3.6.2. Alternate Scenarios . . . 26
3.4. Functional Requirements . . . 26
3.5. Non-Functional Requirements . . . 28
3.5.1. Usability . . . 28
3.5.2. Reliablity . . . 28
3.5.3. Performance . . . 28
3.5.4. Supportablity . . . 28
3.6. Outcome . . . 28

4. Design and Architecture: Approaches and Testing Script **29**
4.1. Approaches . . . 29
4.1.1. Pure LLM . . . 29
4.1.2. In context learning . . . 29
4.1.3. RAG . . . 29
4.1.3.1. Database RAG . . . 30
4.1.3.2. JSONL RAG . . . 31
4.1.4. Fine Tuning . . . 31

5. Design and Architecture: Chatbot Application **32**
5.1. Technology Stack . . . 32
5.1.1. OpenWebUI . . . 32
5.1.2. Django with Frontend Framework . . . 33
5.1.3. Flask with Frontend Library . . . 33
5.1.4. Frontend Frameworks . . . 34

6. Implementation **35**
6.1. Frontend . . . 35
6.2. Backend . . . 35
6.2.1. chat . . . 35
6.2.2. preferences . . . 35
6.2.3. core . . . 36
6.3. Models . . . 36
6.4. RAG . . . 36
6.5. Limitations . . . 37

7. Test **38**
7.1. Test Metrics . . . 38
7.1.1. Correct PostgreSQL Syntax . . . 38
7.1.2. Similarity to target SQL . . . 38
7.1.3. Executability . . . 38
7.1.4. Reliability . . . 38
7.1.5. Performance . . . 38
7.2. Testsets . . . 38
7.2.1. PostgreSQL . . . 39
7.2.2. T-SQL . . . 39
7.3. Tested Approaches . . . 39
7.3.1. Pure LLM . . . 39
7.3.2. In context learning . . . 39
7.3.3. Vanna.ai . . . 40
7.3.4. RAG / our application . . . 40
7.4. Testing application . . . 40

8. Result **41**
8.1. Similarity . . . 41
8.1.1. Pure LLM . . . 41
8.1.2. In Context Learning . . . 42
8.1.3. RAG . . . 42
8.2. Reliability . . . 43
8.3. Executability . . . 43
8.3.1. Pure LLM . . . 43

8.3.2. In context learning . . . 44
8.3.3. Vanna.ai . . . 44
8.3.4. RAG . . . 45

9. Further Development **46**
9.1. Challenges of Further Development . . . 46
9.2. Potential Functional Enhancements . . . 46
9.3. Methodological Approach to Further Development . . . 46
9.4. Importance of Further Development to Stakeholders . . . 47
9.5. Guidelines for Documenting Further Development . . . 47
9.6. Evaluation and Timing . . . 47

10. User Documentation **48**
10.1. Testscript . . . 48
10.2. Chatbot Application . . . 48
10.2.1. Getting started . . . 48
10.2.2. DB Preferences . . . 48
10.2.3. LLM Preferences . . . 49
10.2.4. Chat page . . . 49

III. Projectmanagement and Projectmonitoring **53**

1. General Considerations **54**
1.1. Standards . . . 54
1.2. Configuration Management . . . 54
1.2.1. Development Tools . . . 54
1.2.2. Deployed Software . . . 55
1.2.3. Software Versioning and Dependencies . . . 55
1.2.4. Software Quality . . . 55
1.2.5. Git Workflow and Branching Strategy . . . 55
1.2.5.1. Pull Requests and Branch Protection . . . 55
1.2.5.2. Branching Strategy . . . 55
1.2.5.3. Naming Strategy . . . 56

2. Project Management **57**
2.1. Approach . . . 57
2.2. Resources . . . 57
2.3. Scheduling . . . 57
2.4. Milestones . . . 58
2.4.1. Timeline Diagram . . . 58
2.5. Process and Meetings . . . 58
2.6. Organization and Roles . . . 58
2.7. Risk Management . . . 59

3. Project Monitoring **61**
3.1. Timetracking . . . 61
3.1.1. Benjamin . . . 61
3.1.2. Fiona . . . 61
3.1.3. Total . . . 62
3.2. Project Management . . . 63

IV. Appendix **64**

Acronyms **65**

Glossary **66**

Bibliography **67**

Part I.

Report

1. Introduction

The recent Artificial Intelligence (AI) hype has led to more investments in AI research and development. This has led to the development of more sophisticated AI models that can perform a wide range of tasks. One of the most popular AI models is the (LLM. LLMs are AI models that can understand and generate human language. They have been used in various applications such as chatbots, language translation, and text summarization. One of the most recent applications of LLMs is in the field of (Natural Language to Structured Query Language (NL-to-SQL)) systems. NL-to-SQL systems are AI systems that can convert natural language queries into Structured Query Language (SQL) queries. This has the potential to make databases more accessible to non-technical users.

1.1. Objectives

The primary objective of this project, is to evaluate approaches to provide LLMs with context in order to generate more consistent output. The two main stakeholders are DataGovernance Technologies (DGT), which aims to make its solution available to a broader audience, and Eastern Switzerland University of Applied Sciences (OST), which aims to provide students with a tool that makes learning SQL more interactive, fun, engaging and efficient.

1.2. Problem Statement

Currently students and non-technical users struggle to query databases using SQL. While they can use LLMs to generate SQL queries, the output often needs further corrections, due to the lack of context. This project aims to evaluate the possible solutions to this problem, reducing the need for SQL knowledge.

1.3. Overview

This document outlines the approach taken to conduct research on the topic, our testing approaches and the implementation of the PoC. It provides a comprehensive overview of the current state of the art, the problem domain, the architectural strategies employed, and the quality measures instituted throughout the development process. Additionally, it details the project's timeline, resource allocation, risk management strategies, and the methodologies used to ensure a structured and effective execution. In subsequent chapters, we will delve into the project's requirements—both functional and non-functional—, and architecture. We will also discuss the quality measures that have been applied, the initial project proposal, project plan, time tracking, personal reports, meeting minutes, and the bibliography. The document concludes with an evaluation of how the project's objectives were met and an outlook on potential future developments.

2. Assignment

2.1. Background

One of the objectives of the Swiss software company DataGovernance Technologies Ltd (DGT) is to provide non-technical professionals with actionable insights derived from data. The emergence of Large Language Models (LLMs) has made this vision increasingly attainable, particularly by enabling the generation of SQL queries directly from natural language (NL) input.

Similarly, OST seeks to address a related challenge by making the process of learning SQL more accessible and engaging for their students. Through interactive and efficient methods, OST aims to simplify the complexities of SQL for learners. This project explores the various approaches available for translating NL into SQL, evaluating their effectiveness and potential.

As part of this investigation, a proof of concept (PoC) was implemented to demonstrate and assess these methodologies in practice. Not everybody knows SQL although most people should, since data is everywhere and data is most likely stored in a database.

2.2. Task

This project concerns itself with two main goals:

- Evaluate the effectiveness and state-of-the-art of Text-to-SQL generation.
- Develop an PoC application that can query SQL databases (PostgreSQL and MS SQL) using natural language.

The following subtasks are to be completed:

- Research and evaluate the state-of-the-art approaches in Text-to-SQL generation.
- Develop a test script comparing and evaluating the different approaches.
- Develop a PoC application that can query SQL databases using natural language.

The project will build on the work done in the bachelor thesis N̈atural Language to GraphQL(̈2024).

2.3. Proposed Methodology

The Unified Process Model serves as a template:

- Inception: Define the project scope, requirements and conduct research.
- Elaboration: Evaluate the state-of-the-art approaches and develop a test script.
- Construction: Implement the PoC application using the best approach.
- Transition: Document the results and hand over the project to the stakeholders.

2.4. Specifications/Constraints

- The project should be developed using open-source tools and libraries.
- The PoC application should be able to query both PostgreSQL and MS SQL databases.
- The PoC should be able to execute the queries and show the results in a user-friendly frontend.

2.5. Deliverables

- Documentation, including text abstract (English), management summary (English), brochure abstract
- A test script comparing and evaluating the different approaches.
- A PoC application that can query SQL databases using natural language.

Additionally, the requirements of the Computer Science program and the OST (deadlines, evaluation, documents, etc.) apply.

3. Conditions and Constraints

This chapter outlines the conditions and constraints under which this project operates. These provide the framework within which project decisions are made and guide the technical and organizational approach of the project.

3.1. Project Conditions

- **Stakeholders:** The System under Development (SUD) must cover use cases in the interest of two stakeholders: DGT as well as OST. The SUD must therefore be usable for education as well as data engineering, and for laymen.
- **SQL Assistance:** The project seeks to build a fullstack webapplication that assists in tasks pertaining to SQL.
- **Client Specifications:**

3.2. Technical Constraints

- **Technology Stack:** The backend and frontend must be decoupled, so after this project completes the SUD can easily be integrated into the DGT client, i.e. the frontend can be switched out.
- **Open Source Software:** Because of data privacy concerns preference should be given to open source LLMs over which DGT and OST have complete control.

3.3. Operational Constraints

- **Compliance:** All development and documentation adhere to the rules and standards set forth by the Bachelor's/Master's thesis guidelines of the OST Computer Science program.

3.4. Delivery Constraints

- **Software Quality:** The delivered PoC must adhere to the highest standards of software quality, including clean, decoupled code.
- **Documentation:** Comprehensive documentation is required, including a text abstract in English, a management summary, and a technical report, among potentially other agreed deliverables.

This framework of conditions and constraints will shape the project's execution, ensuring that the development process aligns with the needs of DGT and OST, technical best practices, and academic requirements.

4. Procedure and Structure of the Thesis

4.1. Introduction

This section describes the methods and approaches used in conducting the research and development of the project. It serves to give an overview of the project and the steps taken to complete it.

4.2. Problem and Task Introduction

The primary objective of this project is to create a fullstack webapplication which helps the user understand and access data which lies in a relational database. LLMs already assist in generating SQL queries; they are however often faulty because they lack the information of the database schema. The challenge lies hereby in finding ways of injecting the database schema knowledge of a specific database into the LLM such that the LLM can give more accurate, as well as more complex assistance than without knowledge of the database schema. This project aims to resolve this problem by exploring various possibilities of injecting said domain knowledge such as fine-tuning an LLM, few-shot learning and other paradigms. Finally, a proof of concept must be implemented as a fullstack webapplication.

4.3. Procedure

The project was executed using the Scrum methodology, with 2-week sprints to ensure continuous delivery and adaptation. The following steps were taken to complete the project, though not as sequentially as implied by the numbering. For instance, in order to stay agile it is important to introduce additional metrics should the initially chosen metrics not suffice.

4.3.1. Step 1: Research and Requirements

- **Objective**: Conduct thorough research
- **Activities**:
  - Establish requirements for the project with stakeholders.
  - Establish an overview of existing approaches which are able to fulfill said requirements.
  - Document the approaches as well as their potential.
  - Choose the most promising approaches for further evaluation.
- **Risks**: Miscommunication between stakeholders and project team

4.3.2. Step 2: Metrics

- **Objective**: Establish metrics for comparing approaches
- **Activities**:
  - Research existing metrics for evaluating and evaluating seq-2-seq models and their output.
  - Choose the most suitable metrics, based on their ability to distinguish between the approaches and their potential to provide insights to users, future developers and stakeholders.
- **Risks**: Choosing insufficient metrics, not being able to evaluate the approaches.

4.3.3. Step 3: Testdata

- **Objective**: Establish a set of testdata for evaluating the approaches
- **Activities**:
  - Generate testdata based on the DataGovernance Technologies Database (DGDB).
  - Generate testdata based on the Pagila database.
- **Risks**: Human bias in generating testdata, not enough testdata

4.3.4. Step 4: LLMs

- **Objective**: Choose LLMs for evaluation
- **Activities**:
  - Research existing LLMs and their capabilities.
  - Establish criteria for selecting LLMs.
  - Select four LLMs for further evaluation.
- **Risks**: Human bias in generating testdata, not enough testdata

4.3.5. Step 5: Testscript Development

- **Objective**: Rank the approaches
- **Activities**:
  - Implement logic to run the testdata through LLMs.
  - Implement logic to evaluate the output of the LLMs.
- **Risks**:

4.3.6. Step 6: System Design and Implementation of the PoC

- **Objective**: Design and implement the PoC
- **Activities**:
  - Design the architecture of the PoC.
  - Evaluate different frameworks and technologies
  - Develop the PoC using best practices and the chosen technologies.
- **Risks**: Technical implementation challenges, integration issues.

4.4. Participants Involved

The following individuals were involved in the project:

- **Supervisor**: Prof. Stefan Keller
- **Project Team**: Fiona Pichler, Benjamin Kern
- **CEO Data Governence Tech**: Georg Bommer
- **OST Advisor**: Prof. Dr. Mitra Purandare (provided valuable feedback during the mid-term presentation)

4.5. Summary

In summary, this chapter has provided an overview of the procedure and structure of the work involved in this project. It has outlined the steps taken, as well as its associated risks, and the key participants involved in the project. Additionally, it has introduced the problem and task, and provided an overview of the remaining parts of the submission.

5. State of the Art and Related Work

5.1. Introduction

Initially we focused on refreshing our knowledge concerning seq-2-seq models (and LLMs) which was acquired from the modules AI Foundations and AI Applications. While the refreshers were interesting, it soon proved to be much too low level to concern ourselves with transformer architecture.

Advancement in the AI field have been so vast and rapid, that there are now purpose built abstractions on much higher levels.

When exploring approaches we found the following:

5.1.1. Purellm

This is the most obvious approach and only serves as a baseline. No further modification is needed, only setup of self-hosted LLMs.

5.1.2. In Context Learning

This approach was already mentioned in the bachelor thesis proceeding this project about generating natural language to graph ql [4]. Due to the short timeframe between the two projects, no noteworthy additions can be made here and the reader is referred to [4] for further information.

5.1.3. Finetuning

Finetuning works by getting a so called foundation model, which is then further trained on additional data to achieve better performance on a specific task. A simple example which does not concern itself with LLMs would be a foundational model which has learned to identify edges, now being finetuned in order to recognize numbers. The same could be attempted for generating SQL queries; while the model has certainly been trained on SQL queries, it has not been trained on the specific database schema. Finetuning would then be used to train the model on the specific schema, using example queries. The research conducted here however led to the revelation that we would need "to have thousands or tens of thousands of example" [2] queries, which is not feasible for our project. Furthermore, this would indicate that for each new database schema, the model would need to be retrained using again thousands of examples, which again would need to be generated by hand.

5.1.4. RAG

RAG works by inserting relevant context into a vector database. Unlike relational databases the database is not queried by SQL queries, but by strings which are embedded to find the most relevant entries stored in the database. This retrieved context can then be passed to the LLM.

Research revealed that one of the most widely adopted vector databases is ChromaDB, highly regarded for its ease of use and scalability as well as flexibility.

The following quote from a book provides interesting insights for comparing finetuning and RAG:

"The decision of whether to implement RAG or fine-tune a model relies on the proportion of parametric versus non-parametric information. The fundamental difference between a model trained from scratch or fine-tuned and RAG can be summed up in terms of parametric and non-parametric knowledge:

- Parametric: In a RAG-driven generative AI ecosystem, the parametric part refers to the generative AI model's parameters (weights) learned through training data. This means the model's knowledge is stored in these learned weights and biases. The original training data is transformed into a mathematical form, which we call a parametric representation. Essentially, the model "remembers" what it learned from the data, but the data itself is not stored explicitly.

- Non-Parametric: In contrast, the non-parametric part of a RAG ecosystem involves storing explicit data that can be accessed directly. This means that the data remains available and can be queried whenever needed. Unlike parametric models, where knowledge is embedded indirectly in the weights, non-parametric data in RAG allows us to see and use the actual data

" [5, p. 4] The author further explains that RAG and finetuning are not mutually exclusive, but can instead be combined.

However the ever changing nature of datbases (and its all too well-known need for migration management) makes RAG the much more alluring option, as everytime the database changes, the model would need to be retrained. Updating RAG information on the other hand is much quicker and simpler.

5.1.5. Structured Output

Structured output [3] has been introduced by OpenAI in the midst of this bachelor thesis. It provides a way of standardizing the output of LLMs in a structured way, reducing hallucinations and improving reliability in further processing of the output. The general approach to this problems has previously been to instruct the LLM in the prompt to use delimiters and special formatting. This was also discussed in the natural language to graphQL thesis [4] . The output can then be further cleaned using regular expressions. Unfortunately we did not have time in this thesis to test the real-world applicability of this approach. Since it promises more reliable output, it will most likely find wider adaptation in the future.

5.1.6. Multi-step reasoning

Multi-step reasoning is an approach in which an AI model solves a complex problem by breaking it down into smaller, manageable steps. In the context of NL-to-SQL (natural language to SQL) generation, multi-step reasoning allows a LLM to methodically convert a natural language query into an SQL query by reasoning about intermediate steps.

While this approach has not been implemented in our PoC, the PoC could easily be extended to include this approach, thanks to the modular pipeline design. Extending the current approach with this could improve query generation for queries which contain multiple complex joins and many foreign keys.

The steps taken by multi-step reasoning are as follows:

5.1.6.1. Decomposition of the Query

The natural language query is broken into smaller semantic components (e.g., SELECT clause, FROM clause, WHERE clause). The LLM identifies relevant database entities (tables and columns) separately for different components.

5.1.6.2. Iterative Execution

The SQL query is built incrementally. After each step, the generated SQL is checked for correctness or modified based on user input or further logical steps.

5.1.6.3. Logical Reasoning

The LLM applies rules to handle: Joins between tables based on foreign key relationships. Grouping and aggregation logic for functions like SUM, COUNT, etc. Conditions in the WHERE clause or logical operators (e.g., AND, OR).

5.1.6.4. Intermediate Validation

At each step, intermediate outputs are verified: Are the columns or tables valid? Is the query logically coherent? This can be implemented via schema validation, execution on a database, or simulated reasoning.

5.1.6.5. Use of Feedback

When dealing with ambiguous queries, the LLM can iteratively clarify intent: E.g., “Did you mean to include all rows or only those where column X is not null?”

5.1.7. Agent-like function calling

Agent-like function calling is a concept where the language model operates in a step-by-step, "goal-directed" manner, using discrete functions to reason through problems and generate solutions in a structured way. In the context of NL-to-SQL generation, this approach breaks down the natural language query and iteratively invokes various "agent-like" functions, each handling a part of the problem, to build the SQL query step by step. This allows the system to manage complexity and ambiguities more effectively while handling user input in a more structured way.

This approach sounds very similar to multi-step reasoning, but is more modular. It splits the reasoning task into separate functions or agents, with each "agent" independently performing a specific part of the process (such as parsing the query, extracting entities, handling joins, etc.). These agents work independently but coordinate toward solving the problem. The key difference here is that there is not necessarily one central model processing through all the steps, but several "agents" each handling a task.

While this approach was not implemented, adding it to the PoC would be relatively simple due to the modular pipeline design.

Agent-like function calling can be applied in the context of generating SQL queries from natural language prompts in the following way:

5.1.7.1. Interpretation of User Query (Agent: Language Understanding Agent)

- Function Goal: First, the system needs to understand the natural language query and interpret the intention behind it (e.g., extract entities, operations).
- Function: The model analyzes the user's query and generates a high-level understanding of the goal.

5.1.7.2. Identify Relevant Tables and Columns (Agent: Database Schema Agent)

- Function Goal: The system needs to understand which database tables, columns, and relationships will be relevant for the query.
- Function: This agent queries the database schema (e.g., table names, columns, foreign key relationships) to narrow down the relevant tables and fields that will participate in the query.

5.1.7.3. Determine Joins and Relationships (Agent: Relational Query Agent)

- Function Goal: For more complex queries, especially those involving multiple tables, the system must figure out how to join these tables.
- Function: Using the database schema information, this agent determines which tables need to be joined (based on primary/foreign key relationships) and builds the necessary JOIN statements.

5.1.7.4. Handle Time Constraints or Filters (Agent: Condition Handling Agent)

- Function Goal: If the query includes filters like dates, ranges, or other conditions, the system needs to add these constraints properly.
- Function: This agent analyzes any conditions present in the query and integrates these into the SQL WHERE clause, such as a range or specific value for columns.

5.1.7.5. Construct Aggregations and Groupings (Agent: Aggregation Agent)

- Function Goal: If the query requests an aggregation (e.g., SUM(), COUNT(), AVG(), etc.), the system needs to create the aggregation functions correctly and group the data accordingly.
- Function: This agent reads the high-level intent (e.g., show total sales) and generates the proper SQL aggregation, ensuring proper grouping.

5.1.7.6. Generate SQL Code (Agent: Code Generation Agent)

- Function Goal: Finally, the various agents, having decomposed the problem into manageable tasks, need to generate the final SQL query.
- Function: This agent assembles all previous components into a valid SQL query and provides it to the user.

5.1.8. NLQ translation system

This book proposes interesting algorithms, which do not concern themselves with LLMs, but focus on building SQL by using linguistic and syntactic patterns. As the authors describe:
"Giordani and Moschitti [76] designed an NLQ translation system that generates SQLs based on grammatical relations and matching metadata using NL linguistic and syntactic dependencies to build potential SELECT and WHERE clauses, by producing basic expressions and combining them with the conjunction or negation expressions, and metadata to build FROM clauses that contain all DB tables" [1, p. 23]

Further research (and even the authors of this book) go on to explain the limitations of this apprach, as it cannot handle complex queries. Implementations of such approaches are therefore deemed futile.

6. Evaluation

6.1. Introduction

This chapter goes into the findings of this project, evaluating its outcomes against predefined objectives and criteria.

6.2. Evaluation Methodology

6.2.1. Qualitative Criteria

Qualitative criteria involve descriptive and interpretive measures. In this project, qualitative criteria were assessed based on the requirements discussed with DGT. The need for simplicity in the PoC which implies high usability, and the need to have loose coupling, such that the backend can be integrated in the DGT client, was particularly emphasized.

6.2.2. Quantitative Criteria

Quantitative criteria involve numerical and statistical measures. These include similarity, executability, and reliability metrics. The project utilized various quantitative measures to assess the effectiveness of the different approaches to NL-to-SQL translation.

6.3. Assessment of Results

6.3.1. Data Collection and Analysis

In this project testdata was generated based on the DGDB and the Pagila database. This data was then used to evaluate the different approaches. The analysis focused on evaluating the similarity to the solution and the reliability across different testruns of the generated SQL queries. Techniques like cosine similarity and executability checks were employed to enhance the evaluation process.

6.3.2. Evaluation Metrics

Key metrics used in the evaluation included:

- **Similarity:** The similarity metric compares the output of the LLM with the sample solution provided by the test set. Cosine similarity was used to measure how close the generated SQL queries were to the expected ones.
- **Executability:** This metric checks whether the generated SQL queries can be executed on the target database without errors. It also evaluates the correctness of the result set.
- **Reliability:** This metric assesses the consistency of the LLM's responses to the same query over multiple runs.

6.4. Achievement of Objectives

6.4.1. Objective Fulfillment

This project consisted of two main objectives: Evaluating and comparing promosising approaches and implementing a proof of concept. The first objective was successfully achieved through the evaluation of different approaches to NL-to-SQL translation using a variety of metrics. The executability metric proved crucial in proving that Retrieval Augmented Generation (RAG) is by far the most promising approach.

Furthermore, the second objective was also successfully achieved through the development of a fullstack web application that integrates LLMs and the RAG approach to provide users with a natural language interface to a relational database.

All should and must requirements were met, with the only feature left out being user management. This was however to be expected from the beginning, as it was out of scope for this project. The PoC allows users to easily connect databases and LLMs (open source as well as OpenAI). The explainer mode allows for non-technical users to ask information about queries with context, while the generator mode allows for the generation of SQL queries and immediate display of the resultset in the table. Advanced users can add context of their own by uploading files.

6.4.2. Discrepancies and Unexpected Outcomes

While most objectives were met, some challenges were encountered. For instance, the issue of hallucinations in LLMs was not completely resolved, leading to occasional inaccuracies in the generated SQL queries. Additionally, the performance of the system varied depending on the complexity of the queries and the quality of the input data.

6.5. Future Research Directions

Future work could focus on further refining the RAG approach to reduce hallucinations and improve the accuracy of the generated SQL queries. Additionally, exploring other emerging techniques such as multi-step reasoning and structured outputs could provide further enhancements.

6.6. Conclusion

Overall, this project successfully demonstrated the feasibility of using LLMs and RAG for NL-to-SQL translation. The developed PoC provides a strong foundation for future advancements in AI-powered database accessibility and education.

7. Project Vision and Implementation Strategy

7.1. Project Vision

The goal of this project is a full stack application, which can be connected to a database and an LLM to query the database using natural language. The application uses the chosen LLM to convert natural language to SQL. To find the best results, different LLMs can be connected to the application.

We strive to implement an application and evaluate results which satisfy both the needs of OST as well as DGT.

7.2. Background

While databases are omnipresent in todays digital world, not everybody "speaks" SQL. Telling an application in natural language what to do in a database, makes data more accessible for everybody. Existing final products are limited and often do not satisfy all the requirements posed by OST and DGT, such as data privacy, modularity and support of open source technologies.

7.3. Implementation Strategy

We aim to conduct this project using an iterative and agile approach. We will start by researching the state of the art in the field of LLMs and their application in the context of database querying. We will then proceed to implement a prototype of the application, which will be used to test the different LLMs. The results of these tests will be used to refine the application and the LLMs used. We will then conduct further tests to evaluate the performance of the application and the LLMs. The final application will be compared to existing solutions to determine its effectiveness. At the end of the project we will compare the result with the conditions and constraints established, the assignment as well as the fulfillment of gathered requirements.

8. Outcomes: Assessment and Objectives Fulfillment

8.1. Assessment of Results

This project identified the most promising approaches, measured and compared them and successfully implemented the PoC. Results were compared with the existing product Vanna.ai. The resulted work was evaluated in terms of requirements and the quality of metrics chosen.

8.2. Achievement of Objectives

All objectives were successfully met during this project. Thorough research was conducted, evaluation metrics chosen and different approaches implemented. Requirements of both the academic and commercial world were met and the PoC was successfully implemented.

8.3. Conclusion

This project has successfully managed to satisfy the needs of two stakeholders with different requirements, priorities and interests. The PoC proves valuable for further development not only to DGT, but also to OST.

9. Conclusions and Future Directions

9.1. Conclusions

The project found that the PoC was able to successfully generate SQL queries from natural language input. The PoC was able to generate queries with a high degree of accuracy, and the results were consistent across different LLMs and testdata sets.

The metrics used for evaluation where wisely chosen and the executability metric in particular was a good indicator of the quality of the generated queries. Overall, the PoC has laid a solid foundation for future development and research in the field SQL query generation from natural language. This project is also very relevant in regards to data privacy; while LLMs like ChatGPT are very powerful, they lack the security of selfhosted models. Since our PoC supports all Ollama models, this has opened up great potential for retaining DGT customers data privacy.

9.2. Future Directions

For DGT, future directions will involve the integration of the PoC backend into the existing client, as well as integrating it into the security framework of said client.

For OST, future directions will presumably involve further research into the effectiveness of the different LLMs and the impact of the testdata on the results. Besides exploring alternative approaches one could also investigate how RAG performance can be further improved - for example by including more sample queries for training.

9.3. Final Thoughts

Lessons learned include the importance of iterative development. No matter how great the planning is done in the beginning, requirements and tasks will evolve over time, especially when dealing with problems in the AI space.

Despite that, we managed to deliver a solid PoC and a testing script with results that clearly distinguishes the different approaches, LLMs and testdata sets.

Part II.

SW Project Documentation

1. Overview

1.1. Purpose

The purpose of the software project documentation is to provide a comprehensive and detailed account of the development and implementation of the developed NL-to-SQL-system.

1.2. Content

1.2.1. Vision

Outlines the project objectives, problem statement, and the strategic approach taken to address the identified issues.

1.2.2. Requirements Specification

Lists the functional and non-functional requirements for the PoC.

1.2.3. Design

Discusses design decisions and considerations leading to the final PoC.

1.2.4. Implementation

Describes the implementation of the main application.

1.2.5. Test

Defines the test metrics, tested approaches and overall the testing process.

1.2.6. Result

Provides an assessment of the project's outcomes, the results of the tests and highlighting the main achievements and areas of improvement.

1.2.7. Further Development

Suggests approaches and subjects for further development considering the findings of this project.

1.2.8. User Documentation

Offers documentation to set up the main application implemented in this project.

2. Vision

For more details see chapter 7. Project Vision and Implementation Strategy.

3. Requirement Specification

This chapter defines the requirements for the SUD. Use cases are defined, which are then used to outline the functional as well as the non-functional requirements.

3.1. Prioritization

Requirements are prioritized according to the following three levels. These levels refer to their respective importance of a proof of concept, not necessarily respective to a finished product.

Must These requirements are essential for the proof of concept and must be fulfilled.
Should These requirements should be fulfilled for an extensive evaluation of the proof of concept.
Could These requirements would be "nice to have" - they are deemed out of scope of this project and may only be fulfilled if there is a surplus of time.

3.2. Use Case Diagram

In this section we first present the use case diagram. Subsequently we elaborate on the introduced actors.

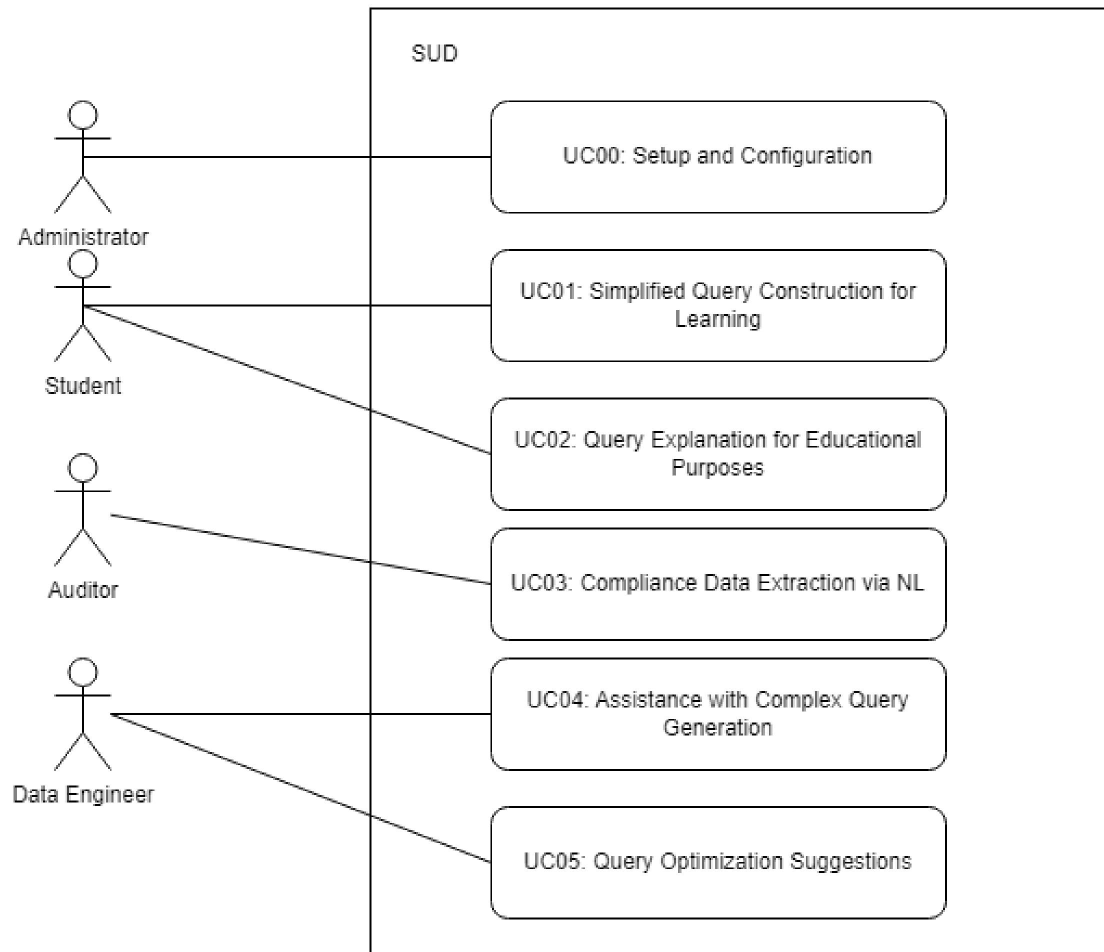


Figure 3.1.: Use case diagram

3.2.1. Actors

We outline the three different actors that will be using our system.

3.2.1.1. Student

The software should be used in education - help students creater a more in-depth understanding quicker than before. As such, one actor is a typical student. He is defined by incompetence regarding SQL.

3.2.1.2. Auditor

Since this thesis also has an industry partner, some roles follow from this. One such role is the auditor: Data Governance Tech provides a client for mining semi-structured data. This data can then be used to verify compliance. If companies have certifications, then they typically have an auditor come in and verify compliance. As such, this auditor needs to verify things such as least-privilege, need-to-know and similar concepts. An auditor typically has more of a legal background, rather than a technical one. This means that his SQL knowledge is limited or non-existent. He should only have access to the files pertaining to the audit.

3.2.1.3. Data Engineer

The last actor is a typical data engineer at Data Governance Tech. His job is to maintain the data pipeline, which includes the DGDB, the DGDWH as well as the OLAP cube. His knowledge in regards to SQL is extensive, but may be limited in regards to the vast schema the data pipeline uses. engineer may need help in creating or adapting the complex queries used in the data pipeline.

3.2.1.4. Administrator

The administrator is the admin of the SUD. He is responsible for connecting new models as well as setting up new users. He may also train models on new database schema given the necessary instructions.

3.3. Use Cases

In the following section we outline the use cases that the SUD should satisfy.

3.3.1. UC00: Setup and Configuration

3.3.1.1. Main Success Scenario

The administrator sets up the new system. He adapts the system to a new database schema. He then sets up the users, as well as their security levels.

3.3.1.2. Alternate Scenarios

If there is an error during setup / configuration the admin is presented with as much information as possible allowing him to remedy the error.

3.3.2. UC01: Simplified Query Construction for Learning

3.3.2.1. Main Success Scenario

A student who is learning SQL uses the system to convert natural language questions into SQL queries to help understand how data is queried from databases and thus help students learn SQL by showing how natural language questions map to SQL queries. In order to achieve that, the student inputs a query like 'Show me the students who scored above 90 in math', and the system generates the SQL query `SELECT * FROM students WHERE math_score > 90;`. With this generated query, the system then queries the database. The query is then shown to the user, as well as its output.

3.3.2.2. Alternate Scenarios

If the query cannot be executed against the database, or a SQL error is thrown, then the query is shown to the user. If no query could be generated, then the user receives as much information about the error as possible, without compromising security.

3.3.3. UC02: Query Explanation for Educational Purposes

3.3.3.1. Main Success Scenario

The system explains complex SQL queries in simple, understandable terms. In doing so it provides detailed explanations to help the student understand query components (joins, subqueries, CTEs, etc.). The student submits a query and asks for an explanation, receiving a step-by-step breakdown.

3.3.3.2. Alternate Scenarios

If the query cannot be explained, then the user receives again as much information as is possible without compromising security.

3.3.4. UC03: Compliance Data Extraction via NL

3.3.4.1. Main Success Scenario

An auditor with limited SQL knowledge inputs a natural language request to retrieve data related to compliance (e.g., access logs or permission levels). The SUD helps auditors quickly extract compliance-relevant information without needing to learn SQL. Thus the auditor inputs a simple query like, "List all users with admin privileges," and the system generates the SQL query ‘SELECT * FROM users WHERE role = 'admin';‘. Futhermore, the SUD can also handle complex queries. Data Governance Tech integrates with various sources such as Exchange, Active Directory and many more. Thus, there are many different types of users. If the auditor requests information about users then the system generates a query which queries the correct subset of users.

3.3.4.2. Alternate Scenarios

If the query cannot be explained, then the user receives again as much information as is possible without compromising security. If the query returns information that the auditor is not allowed to access, then said information is filtered out. The reason for filtering and the information about the filtered content is then presented to the auditor alongside the result.

3.3.5. UC04: Assistance with Complex Query Generation

3.3.5.1. Main Success Scenario

The system helps the data engineer create or optimize complex SQL queries using natural language descriptions and thus allows engineers to focus on business logic rather than the details of SQL query syntax. The engineer requests "I need to adapt the datapipeline such that exchange users are included in the overall entities and can be mapped as owner of emails in the OLAP cube". The system then tells the engineer where and how the SQL queries in the SSIS have to be changed, adapted or added.

3.3.5.2. Alternate Scenarios

Security is of no concern here, as there is only testdata in the system or local installation that the data engineer is working on. If there is an error in generating the instructions and queries, then as much information as possible is presented to the data engineer, facilitating the debugging process.

3.3.6. UC05: Query Optimization Suggestions

3.3.6.1. Main Success Scenario

The system analyzes a SQL query which has been provided by the data engineer and offers optimization suggestions based on the structure and data pipeline. Suggestions include but are not limited to improving query performance by suggesting indexes, optimizations, or rewrites.

3.3.6.2. Alternate Scenarios

If there is an error, then once again as much information as possible is presented to the data engineer, as security is of no concern.

3.4. Functional Requirements

The following table describes the requirements derived from the use cases listed above, as well as referencing the specific use case it has been derived from.

A complex query is defined as a query that includes atleast one of these: CTEs, multiple joins, window functions, aggregation functions.

Number	Title	Derived from UC	Level	Description	Fulfilled
FR01	LLM-Connection	UC00	Must	The SUD provides an interface for connecting LLMs that is generalized, so OpenAI GPT as well as open source models can be connected.	Yes
FR02	User-Administration	UC00	Could	The SUD provides the possibility to administrate users as well as configure different security levels based on tables in a database for said users.	No
FR03	Fullstack	UC03	Must	The SUD is a fullstack webapplication. Since we also target non-technical users, it must be user-friendly and thus not only a console application.	Yes
FR04	NL-to-SQL	UC01, UC03	Must	The SUD generates simple SQL queries from NL input, validates them and shows them to the user.	Yes
FR05	Result-Or-Query	UC01, UC03	Must	When a query has been validated, it is executed against the database and the results can be shown to the user.	Yes
FR06	Execute-Query	UC01, UC03	Must	If the intent is to generate a query then the SUD shows both the generated query and the results.	Yes
FR07	Explain-Query	UC02	Must	When presented with a complex query (as it occurs in a datapipeline for example) the system explains in plain language the concepts behind the queries (e.g. AzureSecEntities table contains users from Azure) and what the query does.	Yes
FR08	Debug-Info	UC04	Should	The SUD helps the user (in this case data engineer) generate and extend complex queries for the datapipeline.	Yes
FR09	Optimize-Query	UC05	Should	The SUD helps the user optimize queries for performance and simplicity.	Yes
FR10	Security	UC03	Could	The SUD implementes security based on security information in the database. Error messages only contain information that the user is allowed to access.	No

3.5. Non-Functional Requirements

We keep track of the non-functional requirements in accordance to the FURPS Standard, where FURPS stands for Functionality, Usability, Reliability, Performance, and Supportability. Since functional requirements are outlined above, they are omitted here.

3.5.1. Usability

NFR	Title	Level	Description	Fulfilled
NFR01	Usability	Must	The SUD is a fullstack webapplication. Since we also target non-technical users, it must be user-friendly and thus not only a console application.	Yes
NFR02	Loading-Feedback	Must	For requests that take long, an indicator is shown to the user.	Yes

3.5.2. Reliablity

NFR	Title	Level	Description	Fulfilled
NFR03	Failure in Query building	Must	The SUD handles failed execution of generated queries gracefully.	Yes

3.5.3. Performance

NFR	Title	Level	Description	Fulfilled
NFR04	Answer-in-10s	Must	Any request to the SUD is answered within 10 seconds.	Yes

3.5.4. Supportablity

NFR	Title	Level	Description	Fulfilled
NFR05	Decoupled-Design	Must	The code and architecture is written highly cohesive and decoupled wherever appropriate, allowing for easy changes in datastore, SUD and frontend/backend technology.	Yes

3.6. Outcome

As one can clearly see, all must and should requirements are fulfilled. The could requirements are not fulfilled, which was to be expected as they are out of scope for this project and only meant to be implemented if there is a surplus of time.

4. Design and Architecture: Approaches and Testing Script

This chapter is split into two main sections: The first section discusses the approaches and the testing script, while the second section dives into the chatbot application.

This chapter outlines the architectural framework and design decisions of the project, focusing on how the object-oriented problem domain is structured and potentially interfaced with a database for data persistence.

4.1. Approaches

Table 4.1.: Approaches

Method	Description
Pure LLM	Asking the LLM with no further context or training for a solution, solely based on the users prompt.
In context learning	The Prompts include 3 example prompts specific to the database with answers.
RAG (Retrieval Augmented Generation)	The user prompt is embedded and subsequently compared with the content in a vector database, such that relevant context can be retrieved. This relevant context is then passed alongside the user prompt to the LLM.
Finetuning	Continue the model's training and adjusting the parameters for a certain task.

4.1.1. Pure LLM

The LLM is asked to give a PostgreSQL solution for a Natural Language Question without further context, though an instruction was added that the output should be a valid SQL query.

4.1.2. In context learning

The LLM is asked to give a PostgreSQL solution for a Natural Language Question, three examples are given with the prompt.

4.1.3. RAG

The principle of storing relevant information about the database in a vector database. The content is stored as a document alongside metadata. In our case there were three different types of RAG:

- RAG with MS SQL database
- RAG with PostgreSQL database
- RAG with JSONL database

4.1.3.1. Database RAG

The following information is extracted from the database using the respective Information Schema Tables: Tablenames, Columnnames, Column Types, Foreign Keys as well as Table Descriptions which can be added by the programmers in PgAdmin or SQL Server Management Studio respectively.

When implementing RAG, the question arises how this information should be stored in the vector database. The following options were considered:

Option 1: Storing Each Table as a Document This approach organizes the schema at the table level, making it easy to retrieve metadata for a single table, including all its columns and relationships. It works well for simple schemas where queries are likely to focus on individual tables. However, for complex queries involving multiple tables, it can be limiting, as the retrieved context may lack detailed cross-table information.

Option 2: Storing Each Column as a Document This method provides the most granular level of information by treating each column as a separate document. It allows for precise retrieval of metadata, including data types, constraints, and foreign key relationships. While it generates more documents, this approach excels in scenarios where queries are complex and span multiple tables, as it maximizes the amount of relevant context that can be retrieved.

Option 3: Storing Foreign Key Relationships as Documents By isolating foreign key relationships, this approach focuses specifically on table connections. It is particularly useful for understanding the relational structure of the database. However, it may fall short in providing the complete context required for complex queries, as it omits details about individual columns outside the relationships.

Option 4: Storing the Entire Database Schema as a Single Document This approach captures the schema as a whole, making it ideal for small databases or global schema overviews. While it provides comprehensive information, it is less efficient for retrieval in large schemas, as queries often only require parts of the schema. Retrieving and processing the entire schema can introduce unnecessary overhead.

Evaluating the best option Given that your queries are complex, often span multiple tables, and require as much detailed context as possible for accurate SQL generation, Option 2: Storing Each Column as a Document is the most suitable approach. The reasons are as follows:

- Fine-Grained Context Retrieval: With each column as a document, the RAG system can retrieve specific details about the columns mentioned in the user's query. This level of granularity ensures no critical context is missed.
- Multi-Table Queries: Complex queries often require information from multiple tables. Column-level documents allow the system to piece together information from all relevant tables and their columns, including relationships, data types, and constraints.
- Comprehensive Context: More context typically leads to better LLM performance. By including metadata for every column, the LLM has access to detailed schema knowledge, improving its ability to generate accurate SQL queries.

- Scalability for Complex Schemas: In large databases with numerous tables and relationships, column-level granularity ensures that the schema is well-represented without requiring retrieval of unrelated data.

While other options have their strengths, **storing each column as a document (Option 2)** provides the most flexibility and depth of context, especially for handling the complex, multi-table queries typical in your use case. By retrieving detailed metadata for each column, the system can generate precise and context-aware SQL queries, ensuring high accuracy and relevance.

4.1.3.2. JSONL RAG

The JSONL RAG serves as an additional input method for users, allowing them to directly influence the context available. Some databases like Pagila contain tablenames which are self-explanatory. However, this is not always the case: DGDB for instance uses a lot of abbreviations as well as domain-specific terms. In such cases, it is helpful to allow the users to add context. This context must be mapping between what should be applied in SQL (e.g. SP in tablename) and NL (e.g. Sharepoint is a Microsoft Product used to share information).

4.1.4. Fine Tuning

During the mid-term presentation we asked Mitra Purandare on her views of the potential of fine-tuning. The discussion revealed that training data for the database would most likely be insufficient as compared to the vast amount of data that the LLM was trained on. Given the time-constraints of this project it was therefore decided not to pursue this approach.

5. Design and Architecture: Chatbot Application

This section explores the various design decisions and architectural considerations made during the development of the chatbot application. It discusses the choice of technologies, frameworks, and the overall structure of the system.

5.1. Technology Stack

The following section explores the choice of technologies and frameworks used in the project, detailing the rationale behind each selection and how they contribute to the system's overall design and functionality.

5.1.1. OpenWebUI

OpenWebUI is a self-hosted WebUI which supports interacting with various LLM runners. OpenWebUI provides a comprehensive set of features right out of the box as the following non-conclusive list shows:

- Popular LLMs can be connected directly via the User Interface (RAG)
- RAG for PDF files is supported straight out of the box via an inbuilt ChromaDB
- Basic Usermanagement (admin vs non-admin roles with option for read-only chats)
- Chathistory
- Extensibility through pipelines

Ease of Development: Frontend must not be implemented, chat functionality is already existing which greatly facilitates initial development as the developer must not bother with websockets for chat interactivity. The functionality of the chat can be expanded via so called "pipelines" or "actions". Instead of directly connecting an LLM , one can choose to connect a pipeline. A pipeline is just a method that receives the users input as a parameter and then returns the generated response. This method is fully customizable and as such can be used to implement filters for security or RAG with a db of one's own choice. However, in our testing of this functionality we could not get OpenWebUI to recognize our pipeline file as a pipeline and it therefore never showed up in the frontend. In such cases, one must then start debugging OpenWebUI's own code, which is rather cumbersome. While OpenWebUI would facillitate and speed up intial development, this initial progress could be lost further down the line for the aforementioned reasons.

Backend Flexibility The pipeline provides great flexibility in processing the user requests. Further extensibility however is not intended and therefore difficult; for instance if one would like to extend the functionality of the user management then this would need to be done by working through the implementation of OpenWebUIs usermodel - something which the creators did not intend. This could also lead to conflicts in the future, when the creators release updates to the usermanagement.
Future Proof If OpenWebUIs code is changed beyond additional files, then in pullrequests from

the forked repository one must resolve merge conflicts of code that was not written by our team. This could prove difficult in the future. Futhermore, if OpenWebUIs development is halted, then the software could be a security risk in the future.

Frontend Features Beyond its very comprehensive set of standard features, OpenWebUI also provides so called "valves" for its pipelines, which are values that can be changed by the user through the frontend. Their intent can be freely chosen by the developer. This allows the user to enter api credentials for LLMs, connection strings, privacy settings and more, as they can be accessed in the pipeline through environment variables.

Community and Support Documentation is lackluster and seems to serve more the purpose of marketing rather than comprehensive technical documentation. It is also rather limited. The community is small.

5.1.2. Django with Frontend Framework

Ease of Development Initial development is cumbersome, as one needs to write a lot of boiler-plate code for the chat interactivity by using websockets.
Backend Flexibility Backend flexiblity is also guaranteed, as one is not restricted by concepts such as piplines provided by OpenWebUI. Futhermore, Django provides built in features such as user management, an admin panel and more. These are easily customizable and extensible. If one does not want to use these options, then one can also implement their own usermanagement.
Frontend Features Frontend can be fully customized - Settings can easily be added, returned data can be represented in a html table or any other type of table, etc. Maximum flexibility is guaranteed. However, all features must be built from the ground up, which may involve some boilerplate code.
Future Proof Django is the most used python backend framework to date and is therefore future proof. As for the frontend framework - one can be chosen with the same quality attributes.
Community and Support Because of its popularity, Django has a large community and support.

5.1.3. Flask with Frontend Library

Ease of Development Same as with Django, with the difference being that Flask is more lightweight and does not include as many built-in features as Django does. This lack of in-built funtionality makes Flask the slightly worse option when it comes to ease of development.
Backend Flexibility Same as with Django, but additional plugins must be searched for by the developer themselves.
Frontend Features Frontend can be fully customized - Settings can easily be added, returned data can be represented in a html table or any other type of table, etc. Maximum flexibility is guaranteed. However, all features must be built from the ground up, which may involve some boilerplate code.
Future Proof Flask is the after Django the most used python backend framework. As for the frontend framework - one can be chosen with the same quality attributes.
Community and Support Because of its popularity, Flask has a large community and support.

Criteria	Weight	Django w/ Frontend		Flask w/ Frontend		OpenWebUI	
		Value	Weighted	Value	Weighted	Value	Weighted
Ease of Development	0.4	8	3.2	6	2.4	5	2.0
Backend Flexibility	0.2	10	2.0	10	0.2	3	0.6
Frontend Features	0.1	9	0.9	9	0.9	7	0.7
Future Proof	0.2	10	2.0	10	2.0	3	0.6
Community and Support	0.1	8	0.8	8	0.8	3	0.3
Total	1.0		8.9		6.3		4.2

Table 5.1.: Evaluation Table with Subcolumns for Each Option

As the focus of this thesis lies on building a PoC and on exploring the capabilities of LLMs to generate and understand SQL, the weight of ease of development is rather high.

In the context of choosing a backend framework
facing the need for simple setup, quick development and extensibility
we decided for Django
neglecting other on first sight simpler frameworks
to achieve sophistication and extensibility in the backend
accepting the initial complexity of getting familiar with Django templates and the like.

5.1.4. Frontend Frameworks

As the goal of this project was to build a PoC, the frontend framework was not a priority and should be kept as simple as possible. Furthermore, the backend should be incorporated into the DGT Client in the near future. Since DGT primarily uses Blazor with WebAssembly, the frontend will be switched out. Loose coupling of the frontend and backend is therefore a priority.

In the context of choosing frontend frameworks
facing the need to have a decoupled frontend and backend
we decided for a pure JS and CSS frontend
neglecting sophisticated design in the frontend
to achieve simplicity
accepting the need to write CSS manually instead of using bootstrap or similar.

6. Implementation

Since the testing scripts were primarily used to evaluate the approaches we did not invest a lot of time into decoupling the code.

This chapter therefore dives into the implementation of the main chatbot application. The application was implemented using the Django framework.

6.1. Frontend

The frontend structure follows the structure proposed by Django. There are three main pages:

- Chat page
- LLM Preferences page
- Database Preferences page

To reduce duplicate code, styling which could be shared, was placed in the static directory served by Django. The frontend communicates with the backend using a REST API, which leads to desirable loose coupling. To allow for chat functionality, messages and responses are sent through Django Channels (websockets). All other requests are made via the REST API.

6.2. Backend

We focused on clean, decoupled code in the backend. There are three main Django apps:

- chat
- preferences
- core

6.2.1. chat

The chat app's only purpose is to handle the chat functionality and the chat page. The user can choose to use the explainer mode, which changes the instructions used in the backend to prompt the LLM. If the user uses the generator mode, the query is generated and if possible executed.

6.2.2. preferences

The preferences app handles the LLM and database preferences. This includes their requests, models needed as well as the views.

6.2.3. core

The core app contains the main logic of the application, such as RAG and the pipeline. The code is written modularly so if the pipeline (which is responsible for processing the user input) needs an additional stage, it can easily be added there.

For RAG and LLMs we used abstract classes to essentially define an interface. This allows programmers to easily add new LLM types or RAG models supporting new SQL dialects.

6.3. Models

The following diagram depicts the models used in the PoC chatbot application. Chatsettings are used to store the users preferences of the chat window, such as selected database. DatabaseFiles are the files which the user uploads to the application to give further context on the connected database. A note to future developers: When handling files one must be careful of the type when retrieving or deleting; one might only delete the file record, not the actual file itself.

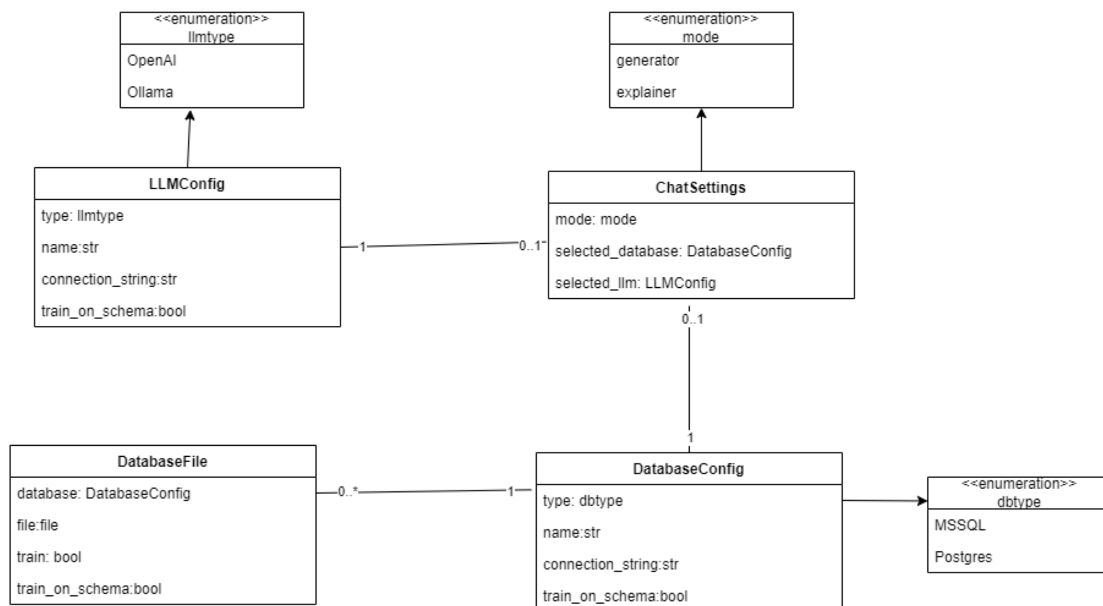


Figure 6.1.: Diagram depicting the objects used in the application

6.4. RAG

Due to the high importance of RAG we attempt to explain it here a bit further.

The following graphic depicts how the application uses RAG to generate SQL queries.

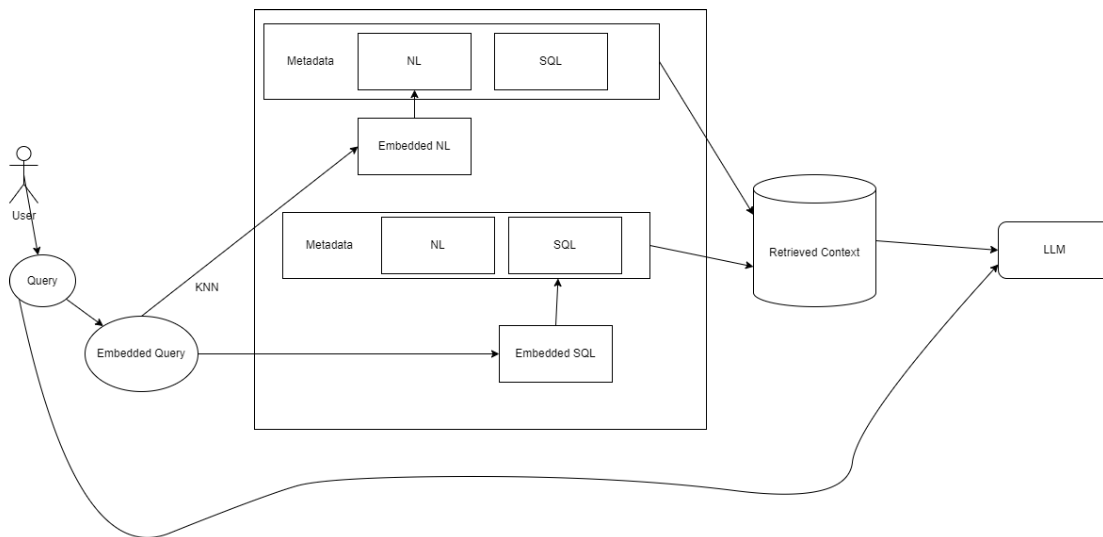


Figure 6.2.: Diagram depicting the flow of the request to the RAG

When the user prompts the chatbot with a question, the pipeline processes the input. The users prompt is embedded and the RAG model retrieves the 10 closest embeddings (KNN with K = 10). The content of the retrieved embeddings is then passed as context to the LLM, and the LLM output returned to the user. The attentive reader might notice that embeddings are inserted twice; once with the NL part embedded, and once with the SQL part embedded. This allows to retrieve the most relevant context for both usecases; generating queries and explaining queries.

6.5. Limitations

As of right now, the application only supports a single user, as user settings are only kept for one entity and database connections, as well as LLM settings, are not directly linked to a user. Due to the lack of time, usermanagement has not been implemented.

7. Test

7.1. Test Metrics

7.1.1. Correct PostgreSQL Syntax

The goal of our application is valid SQL output. We used a parser to check the syntax. Every output was checked whether it had errors. The result of this test is the average for each testcase for a response to have an error or not. For PostgreSQL pglast was used. For T-SQL no parser was found.

7.1.2. Similarity to target SQL

We compared the response of the LLM to the expected SQL statement by calculating the similarity, by embedding the answers as vectors and calculating the cosine distance. This helped to compare the different LLMs and approaches. While raising the problem that some responses though more similar to the solution are not valid SQL. The metric was still considered usefull for comparison but can not show the quality of a response by itself.

7.1.3. Executability

Executability includes correct table and column names as well as validity and the intended use. The Responses were executed on the database and the retrieved data was compared to the expected data from the example solution. As pure LLM and In-context learning have no to almost no knowledge of the correct table und column names this metric was mostly to measure the RAG's performance and to compare it to vanna.ai.

7.1.4. Reliability

How similar is the Response to the same question. We asked the same question 5 times and calculated the similarity, of the first output to the output when asked again. output 1 was compared to output 2 output 2 was compared to output 3 and so on. We calculated the average similarity of these four results.

7.1.5. Performance

The approaches were tested in parallel on different hardware to increase efficiency. Therefore performance was not measured. The tests focused on the quality of the output instead of the velocity of the LLM's. There are already a lot of measurements online to compare different LLM's and it was not considered a crucial part for this thesis.

7.2. Testsets

The Testsets were divided into two levels, basic and advanced to get a more fine grained view, what each approach can cope with. Each Testcase was a json object containing an "SQL" string

and an "Natural Language (NL)" string.

7.2.1. PostgreSQL

Around 40 NL-SQL pairs were taken from the Database course. They had to be edited for the natural language to be more specific. 70 more were generated using ChatGPT. Those testcases were devided into 53 basic and 53 advanced testcases. The number was a result of choosing the best testcases and eliminating very similar ones.

7.2.2. T-SQL

Since DGDB contains fewer tables with actual data and more tables which are required to run and manage the DGT Client (such as metatables about the state of Scanners), the set of queries with human-understandable output is rather limited. Therefore the advanced testset only contains 30 queries, which do include information a user might inquire about (e.g. retrieving the total amount of users, combining Sharepoint and Fileshare users). The basic testset of DGDB therefore contains 50 queries, while the advanced set contains 30 queries. The queries were generated by hand by one of the authors of this paper.

7.3. Tested Approaches

7.3.1. Pure LLM

For comparison the LLM's response to pure specific directions, with no examples given was tested. A lot of prompt engineering was needed, to get useful responses. The following prompt was used along with the database type passed as PostgreSQL or MS SQL:

```
    "You are a Translator. You Translate Natural Language into valid {dbtype} code." +
f"You only answer in valid {dbtype} code. Your answer is directly fed to the database.
Only emit code, emit valid {dbtype}, the answer should be 100\% {dbtype}"+
f"remove all notes, remove all assumptions , remove all explanations from your answer.
The whole answer has to be valid {dbtype} code.
Only use the examples and context provided here as help,
 ONLY add the information that the user asked for.
 The context is as follows: No context provided.
```

7.3.2. In context learning

The next step from Pure LLM was in context learning where examples are provided as part of the prompt. Three examples that where not part of the test sets were chosen. The examples were the same for the basic and the advanced test sets. The Prompt was the same but with the following context added for PostgreSQL:

```
 "'Here are 3 examples: ' +
'userprompt "Count the number of rentals made by each customer,
    sorted by the highest number of rentals",
response "SELECT customer\_id, COUNT(*) AS rental\_count FROM
```

```
    rental GROUP BY customer\_id ORDER BY rental\_count DESC" '

'userprompt "List all categories with the count of films in each,
    sorted by the highest number of films",
response "SELECT category.name, COUNT(film\_category.film\_id)
    AS film\_count FROM category JOIN film\_category
    ON category.category\_id = film\_category.category\_id
    GROUP BY category.name ORDER BY film\_count DESC" '

'userprompt "Calculate the duration of each rental
    in days for returned rentals",
response "SELECT rental\_date, return\_date,
    EXTRACT(DAY FROM return\_date - rental\_date) AS rental\_duration
    FROM rental WHERE return\_date IS NOT NULL"
```

For T-SQL the same prompt was used with the following context:

```
'Here are 3 examples: '
'userprompt "Give me all azure users which
    do not have a Security Identifier",
response "select * from AzureSecEntity where SID is null"'

'userprompt "I want to know the total number of olap cubes
    which do not include sharepoint versions",
response "select count(*) from cube where IncludeSPVersions = 0;"'

'userprompt "Give me all the confluence pages
    which have more than 5 files attached",
response "select * from ConfluencePage where FileCount > 5;"'
```

7.3.3. Vanna.ai

For comparison vanna.ai was tested using all four LLMs except for the gpt 4o-mini model. It was connected to the PostgreSQL Test-Database and trained with the database schema as well as the same three examples, we used for in context learning. Since DGDB contains confidential information, Vanna.ai was only tested with the PostgreSQL database.

7.3.4. RAG / our application

The Heart of this SA/ BA was also tested, using RAG to translate NL to SQL. The RAG application is more thoroughly described in section RAG.

7.4. Testing application

Using python scripts for each Testing criteria, the results are summarized in a table to compare them. For each of the two test sets and each of the four approaches the tests were run 10 times, to then use the average of the results. The sample size is rather small due to the time it took to get the samples. 10 samples per approach per testcase was the best trade off to get more objective results in a short time.

8. Result

This chapter dives into the results found by our testing script. Results were averaged over 10 testruns. Finally, the results were grouped by LLM, Database and testset (basic vs advanced) The following tables depict these results. To give readers immediate context, the highest value achieved for the DGDB is highlighted in orange, the highest value for the OST is highlighted in yellow, and the highest value overall is highlighted in green.

8.1. Similarity

The similarity metric compares the output of the LLM with the sample solution provided by the testset. As a similarity metric the cosine function was chosen, as it takes the direction of the embedding vector into account. It is therefore the best possible choice for comparing embeddings. The value of the similarity is therefore in the range of [-1;1] where 1 corresponds to equality.

8.1.1. Pure LLM

LLM	Database	Similarity (%) basic		advanced	
		avg	max	avg	max
Mistral-7b	DGDB	0.52	0.54	0.59	0.60
Mistral-7b	OST	0.92	0.93	0.75	0.77
phi3:3.8	DGDB	0.39	0.56	0.41	0.58
phi3:3.8	OST	0.86	0.89	0.53	0.74
llama3.2	DGDB	0.49	0.52	0.61	0.63
llama3.2	OST	0.94	0.94	0.78	0.79
phi3:14	DGDB	0.59	0.63	0.60	0.62
phi3:14	OST	0.88	0.89	0.76	0.77
gpt-4o-mini:	DGDB	0.57	0.54	0.63	0.66
gpt-4o-mini:	OST	0.94	0.94	0.80	0.81

As one can see based on the highlighted numbers, the highest similarities were achieved on the OST (Pagila) database. This is rather obvious, as Pagila uses tablenames which are not domain specific, such as "Users" for users, whereas the DGDB uses the table "SecEntity" (i.e. Security Entities) among others to store users. Since the Pure LLM approach has no context, it has no chance of ever guessing the tablename SecEntity correctly.

8.1.2. In Context Learning

LLM	Data-base	Similarity (%) basic		advanced	
		avg	max	avg	max
Mistral-7b	DGDB	0.60	0.62	0.65	0.67
Mistral-7b	OST	0.89	0.90	0.77	0.78
phi3:3.8	DGDB	0.43	0.61	0.38	0.64
phi3:3.8	OST	0.54	0.89	0.46	0.76
llama3.2	DGDB	0.64	0.67	0.65	0.67
llama3.2	OST	0.95	0.97	0.80	0.81
phi3:14	DGDB	0.66	0.67	0.65	0.67
phi3:14	OST	0.90	0.91	0.78	0.79
gpt-4o-mini:	DGDB	0.62	0.64	0.67	0.68
gpt-4o-mini:	OST	0.95	0.96	0.78	0.80

Unsurprisingly the in context learning approach performs slightly better, since some tablenames are provided as context. Since only 3 example solutions are provided to the LLM, the results are only marginally better.

8.1.3. RAG

LLM	Data-base	Similarity (%) basic		advanced	
		avg	max	avg	max
Mistral-7b	DGDB	0.47	0.48	0.56	0.58
Mistral-7b	OST	0.68	0.70	0.65	0.66
phi3:3.8	DGDB	0.28	0.54	0.27	0.57
phi3:3.8	OST	0.39	0.76	0.33	0.68
llama3.2	DGDB	0.53	0.55	0.61	0.63
llama3.2	OST	0.79	0.81	0.70	0.72
phi3:14	DGDB	0.55	0.61	0.60	0.62
phi3:14	OST	0.80	0.81	0.71	0.72
gpt-4o-mini:	DGDB	0.64	0.71	0.63	0.63
gpt-4o-mini:	OST	0.88	0.89	0.76	0.77

Surprisingly RAG seems to perform comparibly to the other approaches, when only taking the similarity into account. This is because of model hallucinations; even if the model is instructed to only add necessary information it often injects more information than necessary, because it is provided in the context. One might ask why the similarity metrics of purellm is similar for RAG. While pure LLM must completely guess tablenames, for some tablenames the guess is quite similar. E.g. instead of accessing the table DoMailFolder, it tries to access MailFolder, which is incorrect but similar enough as to not affect similarity significantly.

8.2. Reliability

Reliability only depends on the LLM as all inputs are the same, the results were grouped per LLM and the average taken. The low result of phi3:3.8 emphasizes the model's hallucinations. Since it is probably the smallest model in this test, this is no surprise.

LLM	Reliability (%)	Rank (1 = best)
Mistral-7b-v.01	0.86	4
phi3:3.8	0.26	5
llama3.2	0.97	1
phi3:14	0.91	3
gpt-4o-mini:	0.93	2

8.3. Executability

Since similarity metrics alone have been proven to be insufficient in distinguishing the approaches, we also took executability into consideration. The output of the LLM has been run on the appropriate database i.e. Pagila or DGDB. Furthermore, for each output it was calculated how many columns appear in the sample solution. E.g. if the resultset contained the columns "firstname" and "lastname", and the sample solution was "select firstname, lastname from users;" a perfect column identification took place. This metric however must be taken with a grain of salt, as some sample queries also included "select * from users;" - since no columns were explicitly asked for, the columns identified would be 0.

8.3.1. Pure LLM

LLM	Database	columns correctly identified (%)				queries executable on database (%)			
		basic		advanced		basic		advanced	
		avg	max	avg	max	avg	max	avg	max
Mistral-7b	DGDB	0.00	0.05	0.00	0.00	0.10	0.17	0.05	0.05
Mistral-7b	OST	0.50	0.63	0.00	0.00	0.06	0.08	0.00	0.00
phi3:3.8	DGDB	0.00	0.00	0.00	0.00	0.04	0.04	0.05	0.05
phi3:3.8	OST	0.72	1.0	0.00	0.00	0.07	0.11	0.02	0.02
llama3.2	DGDB	0.34	1.0	0.73	1.0	0.08	0.20	0.08	0.15
llama3.2	OST	0.63	0.71	0.37	1.0	0.21	0.23	0.03	0.06
phi3:14	DGDB	0.04	0.34	0.26	0.50	0.06	0.10	0.06	0.10
phi3:14	OST	0.67	1.0	0.16	1.0	0.02	0.02	0.03	0.06
gpt-4o-mini:	DGDB	0.17	0.40	0.28	0.50	0.14	0.20	0.08	0.15
gpt-4o-mini:	OST	0.53	0.60	0.10	0.25	0.11	0.13	0.05	0.08

8.3.2. In context learning

LLM	Data-base	columns correctly identified (%) basic		advanced		queries executable on database (%) basic		advanced	
		avg	max	avg	max	avg	max	avg	max
Mistral-7b	DGDB	0.40	1.0	0.25	0.50	0.04	0.07	0.05	0.05
Mistral-7b	OST	0.59	0.68	0.41	0.50	0.50	0.53	0.16	0.19
phi3:3.8	DGDB	0.70	1.0	0.00	0.00	0.05	0.07	0.00	0.00
phi3:3.8	OST	0.50	0.58	0.50	1.0	0.29	0.32	0.05	0.08
llama3.2	DGDB	0.83	1.0	0.48	1.0	0.08	0.13	0.05	0.05
llama3.2	OST	0.61	0.66	0.53	0.63	0.71	0.77	0.36	0.43
phi3:14	DGDB	0.86	1.0	0.00	0.00	0.07	0.10	0.00	0.00
phi3:14	OST	0.60	0.67	0.42	0.58	0.39	0.49	0.12	0.21
gpt-4o-mini:	DGDB	0.40	0.60	0.44	0.50	0.08	0.13	0.08	0.13
gpt-4o-mini:	OST	0.69	0.71	0.46	0.55	0.71	0.77	0.59	0.68

As one can see in the above two tables, execution metrics suffer for pure LLM and in context learning, since a similar enough tablename is no longer sufficient. The tablename must be entirely correct, or the query cannot be executed.

8.3.3. Vanna.ai

LLM	Data-base	columns correctly identified (%) basic		advanced		queries executable on database (%) basic		advanced	
		avg	max	avg	max	avg	max	avg	max
Mistral-7b	OST	0.59	0.79	0.42	0.80	0.27	0.32	0.1	0.18
phi3:3.8	OST	0.43	0.55	0.28	0.67	0.17	0.23	0.03	0.04
llama3.2	OST	0.55	0.61	0.50	0.67	0.49	0.52	0.21	0.28
phi3:14	OST	0.46	0.55	0.25	0.5	0.32	0.38	0.09	0.13

Since Vanna.ai is trained on the database, it is able to generate much more accurate queries in comparison to in context learning and pure LLM.

8.3.4. RAG

LLM	Data-base	columns correctly identified (%) basic		advanced		queries executable on database (%) basic		advanced	
		avg	max	avg	max	avg	max	avg	max
Mistral-7b	DGDB	0.25	0.63	0.05	0.33	0.05	0.07	0.09	0.15
Mistral-7b	OST	0.66	0.73	0.51	0.88	0.25	0.30	0.06	0.09
phi3:3.8	DGDB	0.29	0.5	0.02	0.08	0.13	0.20	0.11	0.2
phi3:3.8	OST	0.56	0.78	0.43	0.7	0.17	0.21	0.08	0.09
llama3.2	DGDB	0.41	0.53	0.41	0.57	0.38	0.47	0.34	0.45
llama3.2	OST	0.64	0.68	0.45	0.61	0.62	0.64	0.30	0.38
phi3:14	DGDB	0.30	0.57	0.09	0.33	0.23	0.40	0.25	0.35
phi3:14	OST	0.62	0.69	0.42	0.50	0.40	0.45	0.12	0.19
gpt-4o-mini:	DGDB	0.49	0.58	0.11	0.16	0.71	0.80	0.80	0.90
gpt-4o-mini:	OST	0.70	0.73	0.50	0.55	0.62	0.66	0.45	0.55

As one can clearly see, our RAG performs best in terms of executability, even outperforming vanna.ai. Being able to retrieve the most similar information from the database, gives it an agility that in context learning and pure LLM lack.

9. Further Development

This chapter explores potential avenues for the further development of the project, highlighting both functional enhancements and methodological approaches to continue evolving the system. The importance of further development to stakeholders is discussed, along with guidelines for documenting future work and evaluating its impact on the project.

9.1. Challenges of Further Development

Challenges of further development include the following:

- Security: While blacklisting tables and columns may provide some security, more granular security will be challenging to be implemented. E.g. some endusers may be allowed to access information about some movies, but not all. This use case will become relevant for DGT in the future.
- Context Scope: One of the challenges identified in this project is the balance between providing enough context, while not providing too much. Too much context leads to hallucinations, while too little leads to poor performance of executability metrics.

9.2. Potential Functional Enhancements

The following functional and non-functional requirements could be addressed in future iterations of the project:

- **New Features or Functionalities:**
  - Implementing user management to support multiple users with different roles and permissions.
  - Adding support for more SQL dialects beyond PostgreSQL and MS SQL.
- **Improvements or Refinements to Existing Features:**
  - Improving the accuracy of SQL query generation by refining the RAG approaches and implementing alternative approaches revealed in the research, such as structured output.
  - Enhancing the error handling and feedback mechanisms to provide more informative responses to users.

9.3. Methodological Approach to Further Development

Future development should be approached with a focus on clean, change-friendly design. This includes:

- Conducting regular code reviews to maintain code quality.
- Implementing continuous integration and continuous deployment (CI/CD) pipelines to streamline the development process.

9.4. Importance of Further Development to Stakeholders

Further development is crucial from a stakeholder perspective for several reasons:

- **Making databases accessible:** By improving the existing application, databases are made more accessible to a broader audience, including non-technical users.
- **Functional Expansion:** Adding new features and functionalities can enhance the system's value to users and stakeholders.
- **Introduce more metrics:** While the metrics introduced in this project were sufficient to evaluate the approaches, more fine-grained metrics could provide deeper insights into the system's performance and how it could be further improved.

9.5. Guidelines for Documenting Further Development

Documentation should be kept short and concise, as to avoid a large amount of documentation which is quickly outdated. Further development should take advantage of the decoupled nature of the codebase and continue to ensure clean, modular code which in turn reduces the need for extensive documentation.

9.6. Evaluation and Timing

Further development may take different paths for DGT and OST due to their different goals. While security may be of relevance to DGT, increasing the quantifiability of the performance of system may be more important to OST. This is due to the fact that OST represents academic interests, while DGT represents commercial interests.

The timing of further development should also take releases of new LLMs and general advances in the AI space into account.

10. User Documentation

10.1. Testscript

Since the testing script is not designed for endusers in mind, the instructions will be ommitted here and can instead be found in the README of the testing script.

10.2. Chatbot Application

This section provides a user guide for the Chatbot application developed in this project. The Chatbot is designed to help users query SQL databases using natural language inputs. The application is accessible via a web interface and provides a user-friendly environment for interacting with the database.

10.2.1. Getting started

Navigate to

```
\chatdjango\ChatApp
```

and follow the steps described in the README in the same directory. This starts the application.

10.2.2. DB Preferences

From the chat page, navigate to the by db preferences by pressing the DB Connections button. Here you can add new database connections. Before training, make sure the connection works by pressing the Test Connection button.
You can upload multiple jsonl files with SQL to NL mappings, providing further context for the database. To enable rag training, make sure the toggles are enabled. Train on Schema leverages the direct connection to the database to extract metadata about its tables.
They must be in the following format (json objects as lines):

```
{"SQL": "SELECT * FROM table", "NL": "Get all entries from table"}
```

Make sure to hit the save button before clicking the Train button. While training is in progress, a spinner will be displayed.

Database Preferences

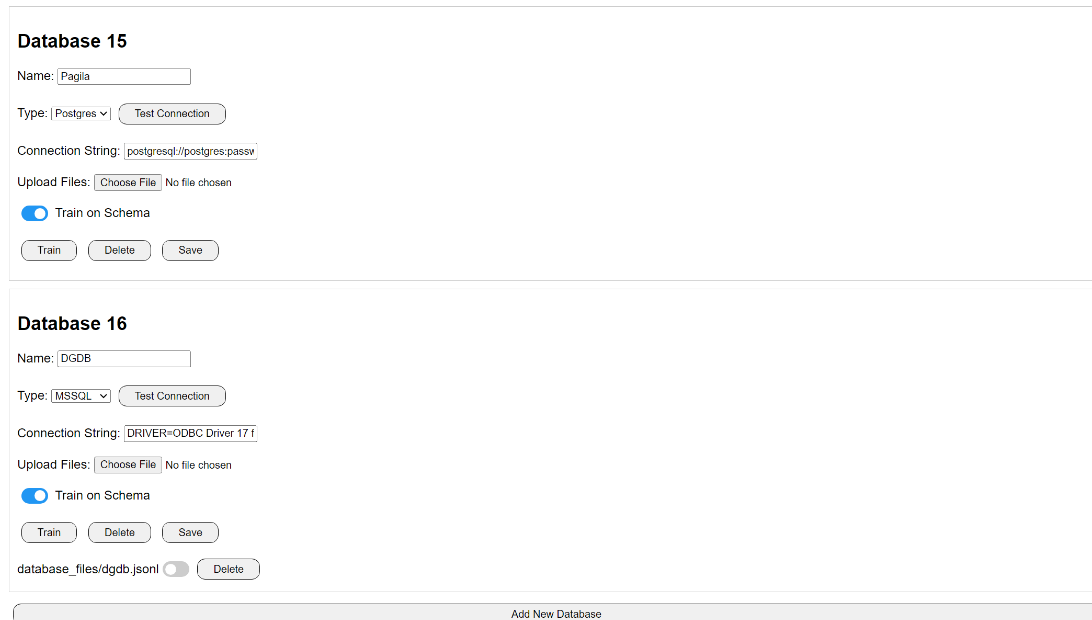


Figure 10.1.: DB Preferences page

10.2.3. LLM Preferences

From the chat page, navigate to the by LLM preferences by pressing the LLM Connections button. Two types are currently supported:

- Locally hosted Ollama models
- OpenAI models

LLM Preferences

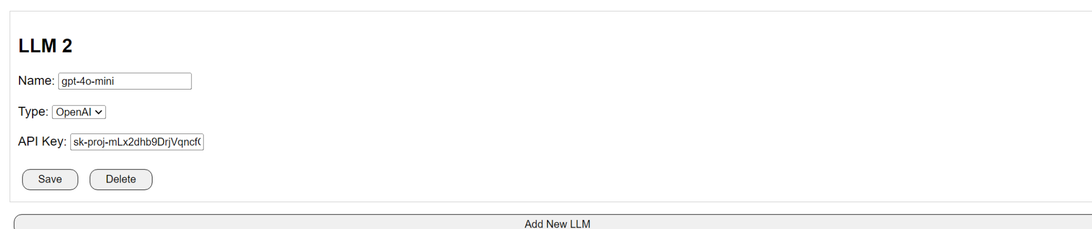


Figure 10.2.: LLM Preferences page

Make sure that the modelname corresponds to the actual name of the model, like gpt-4o-mini. For OpenAI, an api key is required.

10.2.4. Chat page

Choose one of the databases and LLMs you have set up; use the dropdowns to select them.

On the main chatpage you can also choose the mode; either explainer or generator. Paste a query in the input field and press enter. While the chatbot is processing your request, a spinner will be displayed.

DB Connections LLM Connections

Mode: Explainer Database: Pagila LLM: gpt-4o-mini

SELECT category_id, COUNT(film_id) AS film_count FROM film_category GROUP BY category_id ORDER BY film_count DESC LIMIT 1;

This SQL query is designed to find out which film category contains the most films in the `film_category` table. Let's break it down step-by-step: 1. **SELECT category_id, COUNT(film_id) AS film_count**: - This part of the query means we want to select two pieces of information: - `category_id`: the ID of the category. - `COUNT(film_id)`: the number of films that belong to each category (the total count of film IDs for each category). We are naming this count as `film_count`. 2. **FROM film_category**: - This indicates that we are retrieving data from the `film_category` table. 3. **GROUP BY category_id**: - This groups the results by `category_id`. It means that for each unique category ID, all the film IDs under that category will be counted together. 4. **ORDER BY film_count DESC**: - After counting the films in each category, we want to sort the categories by the number of films they contain. `DESC` means "descending," so we will start with the category that has the most films. 5. **LIMIT 1**: - This limits the results to just one record. In this case, we only want the category with the highest count of films. Putting it all together, the query is asking for the category ID that has the highest number of films, along with the count of those films. The result will give you just one category ID and how many films are in that category.

Type your message here... Ask

Figure 10.3.: Chat page in explainer mode

To generate queries, use the mode generator.

DB Connections LLM Connections

Mode: Generator Database: Pagila LLM: gpt-4o-mini

10 entries per page Search:

category_id	film_count
15	74

Showing 1 to 1 of 1 entry « ‹ 1 › »

give me the category with the most films

SELECT category_id, COUNT(film_id) AS film_count FROM film_category GROUP BY category_id ORDER BY film_count DESC LIMIT 1;

Type your message here... Ask

Figure 10.4.: Chat page in generator mode with successful query

If the query cannot be executed, the result will be displayed in red. In the case below the query could not be executed, as the wrong database was selected (while pagila contains an actor table, DGDB does not).

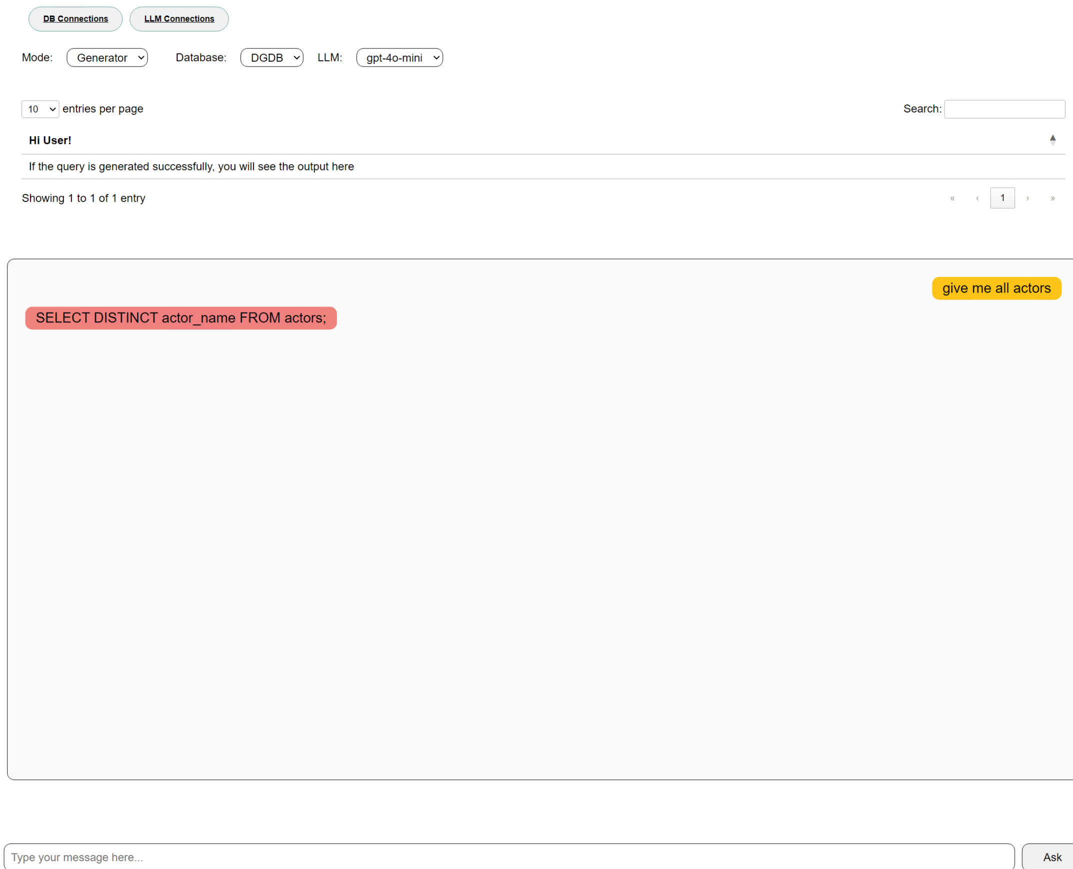


Figure 10.5.: Chat page in generator mode with failed query

Part III.

Projectmanagement and Projectmonitoring

1. General Considerations

This chapter addresses the foundational elements that underpin the project's execution, focusing on standards compliance and configuration management practices.

1.1. Standards

Throughout the project, various standards were adhered to in order to ensure quality and interoperability. These standards included coding standards and data exchange formats.

- **Coding Standards**: The project followed PEP8 for Python coding to maintain readability and consistency across the codebase.
- **Data Exchange Formats**: JSONL was used as the primary data format for storing training data. Furthermore, JSON was used for API communication in the PoC, ensuring compatibility and ease of integration.

1.2. Configuration Management

Configuration management strategies employed in the project included the use of various development tools and software to support the development process.

1.2.1. Development Tools

Several development tools were used throughout the project, each playing a specific role in the development lifecycle.

- **Operating Systems**: Since MS SQL server only runs on Windows, it was decided to only employ Windows hardware.
- **Azure DevOps**: Utilized for issue management, sprint planning, version control, release management, pull requests and branching strategies and deployment (latex pipeline). For the PoC no such pipeline could be established, as this would have required tight integration of DGT infrastructure with OST, which was not possible due to security and data privacy concerns.
- **Pylint**: Used to ensure adherence to coding standards and identify potential issues in the codebase.
- **LaTeX**: Employed for creating the project documentation, ensuring a professional and structured format.

1.2.2. Deployed Software

The project's infrastructure included various software components:

- **Databases**: MS SQL Server for hosting DGDB. PostgreSQL was used for Pagila. ChromaDB to store the embeddings of RAG. SQLite as the database to store settings and configuration of the PoC.
- **Server Software**: Ollama was used to allow easy hosting of open source models. The PoC also required Visual Studio C++ Build Tools to be installed on the system, in order for ChromaDB to work.
- **Third-Party Services**: Integrated various libraries and services to enhance functionality and performance, including ChromaDBs embedding functions.

1.2.3. Software Versioning and Dependencies

A structured approach was taken to manage software versioning and dependencies:

- **Version Control**: Git was used for version control, with a clear branching strategy to manage development cycles.
- **Dependency Management**: Managed using virtual environments Venv as well as Miniconda requirements files in Python, ensuring consistent environments across different stages of development.

1.2.4. Software Quality

To ensure software quality, Pylint was used to monitor for code quality issues. This setup provided immediate feedback, highlighting potential issues early in the development process and maintaining high standards throughout the project.

1.2.5. Git Workflow and Branching Strategy

A Pull Request workflow was employed to maintain code quality and ensure the stability of the main branch. Key aspects included:

1.2.5.1. Pull Requests and Branch Protection

- **Pull Request Review**: Every PR required approval from a designated reviewer before merging to the main branch.
- **Main Branch Protection**: The main branch was protected to ensure that only approved changes were merged, maintaining the integrity of the codebase.

1.2.5.2. Branching Strategy

The branching strategy facilitated collaboration and organized development:

- **Documentation Repository**: Branches were created directly from the main branch.
- **Code Repository**: Feature branches were created from the main branch and further divided into branches associated with specific backlog items.

1.2.5.3. Naming Strategy

Consistent naming conventions were maintained to ensure clarity and organization:

- **Feature Branches**: Prefixed with "/feat".
- **Bug Branches**: Prefixed with "/bug".

This naming convention helped organize branches in Azure DevOps Overview, creating a hierarchical folder structure.

Example Naming Convention Branch names typically adhered to the following format:

```
"/feat/FeatureNumber/BacklogItemNumber-Desc"
```

This structured approach streamlined development processes and enhanced visibility across the project.

2. Project Management

This chapter outlines the project management methodologies and planning strategies employed throughout the project lifecycle, ensuring the structured progression from inception to transition.

2.1. Approach

Our project management approach is rooted in the Scrum framework, adapted to meet the unique challenges and requirements of our project. This agile methodology facilitated rapid iterations, enabling us to respond to changes efficiently. Bi-weekly sprints were the cornerstone of our process, with each sprint leading to a potentially shippable product increment.

2.2. Resources

The project resources were the following:

- Benjamin Kern, 360 hours
- Fiona Pichler, 240 hours
- Cloud: Azure DevOps
- Test data: Pagila and DGDB databases

2.3. Scheduling

The project was scheduled regarding six milestones, with the start date on 19.09.2024 and the end date on 10.01.2025. The Milestones were always scheduled at the end of a sprint. The scheduling strategy was structured as follows using the Rational Unified Process model phases:

- Inception: Approximate vision, defining the scope, and rough estimates for efforts.
- Elaboration: Identification of most requirements, iterative implementation of core architecture, resolution of high risks, and more realistic estimates for efforts.
- Construction: Iterative implementation of functionality, resolution of lower risks, and preparation for deployment.
- Transition: Beta tests, deployment, and tying up any loose ends. In regard of Christmas and New Years holidays this phase is schedules for longer than usual.

Milestone	Date	Description	Deliverables
M0	06.10.2024	Project Kickoff	Dev Infrastructure, Project Goal Defined, Long term plan
M1	20.10.2024	Project Governance	Initial requirements (NFR and FR), Riskmanagement
M2	03.11.2024	Data Sources	Testdata bases established
M3	17.11.2024	Architecture	Tools, Libraries, Frameworks, APIs evaluated, Core Architecture established.
M4	15.12.2024	MVP	MVP implemented
M5	10.01.2025	Project End	Documentation finalized, Deliverables packaged and delivered

2.4. Milestones

2.4.1. Timeline Diagram

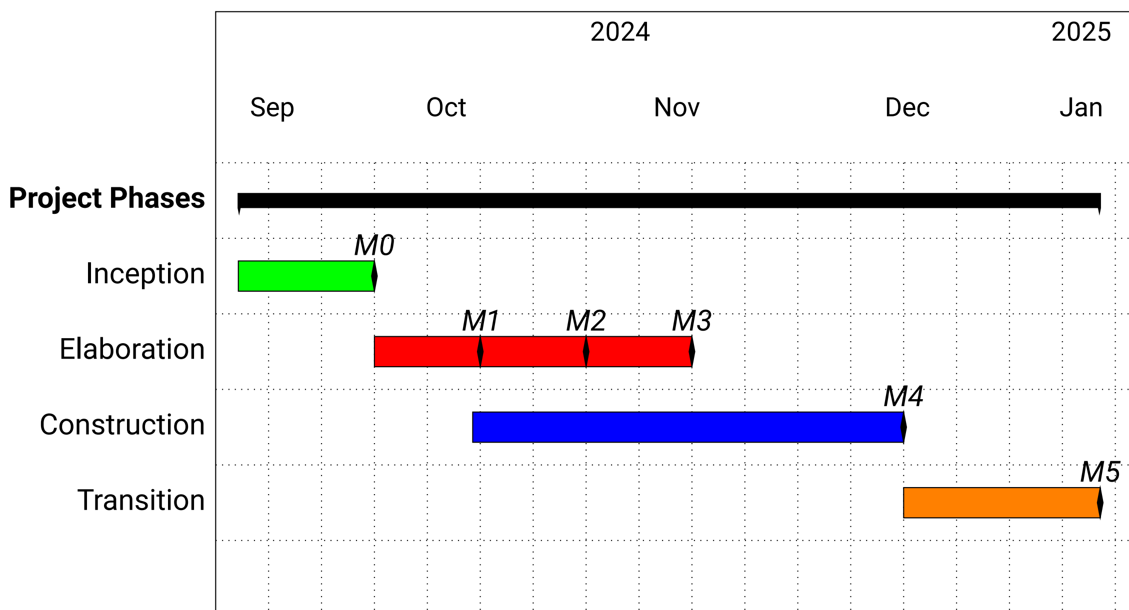


Figure 2.1.: Project Timeline

2.5. Process and Meetings

Project meetings were an integral part of the Scrum process, with sprint planning sessions every two weeks.

2.6. Organization and Roles

Due to the higher expected workload of Benjamin, it was decided that Benjamin would be responsible for the implementation of the PoC (chatbot application), while Fiona would focus on the initial testing scripts. The testing scripts were then finished in a joint effort, after being further

extended by Benjamin. Fiona was tasked with generating the queries for Pagila, while Benjamin was responsible for the DGDB.

2.7. Risk Management

Risk management is vital for project success, ensuring potential issues are identified, assessed, and mitigated. Below is an overview of identified risks, their descriptions, proposed mitigation strategies, and current status. The risk assessment matrix illustrates the impact and likelihood of each risk before and after implementing mitigation measures.

Nr.	**Risk Description**	**Mitigation**	**Status**
R1	Self-hosting LLMs requires powerful hardware	Request servers from OST and DGT (AzureCloud) with dedicated GPUs.	Resolved
R2	Data privacy concerns of DGT	Use DGT servers for hosting critical data - never transfer the DGDB to OST servers	Resolved
R3	Chosen approaches not performing as well as expected, thus needing replacement	Implement modular design for easy replacement slow components	Resolved
R4	Illness of team members	Ensure regular work status updates in issue management system	Resolved

Table 2.1.: Risk Assessment and Mitigation

Impact	Very Unlikely	Unlikely	Possible	Probable	Very Likely
Critical				R1	R2
Serious			R3		
Moderate			R4		
Minor					
Negligible					

Table 2.2.: Project risk assessment matrix before mitigation

Impact	Very Unlikely	Unlikely	Possible	Probable	Very Likely
Critical					
Serious					
Moderate			R3		
Minor					
Negligible			R4	R1	R2

Table 2.3.: Project risk assessment matrix after mitigation

Despite the initial concerns, only a single risk materialized during the project, since OST was taking down the powerful dedicated GPU server (R1). We were however able to run the tests before the server was taken down, therefore rendering the requested servers from DGT unnecessary. It was nevertheless a wise foresight.

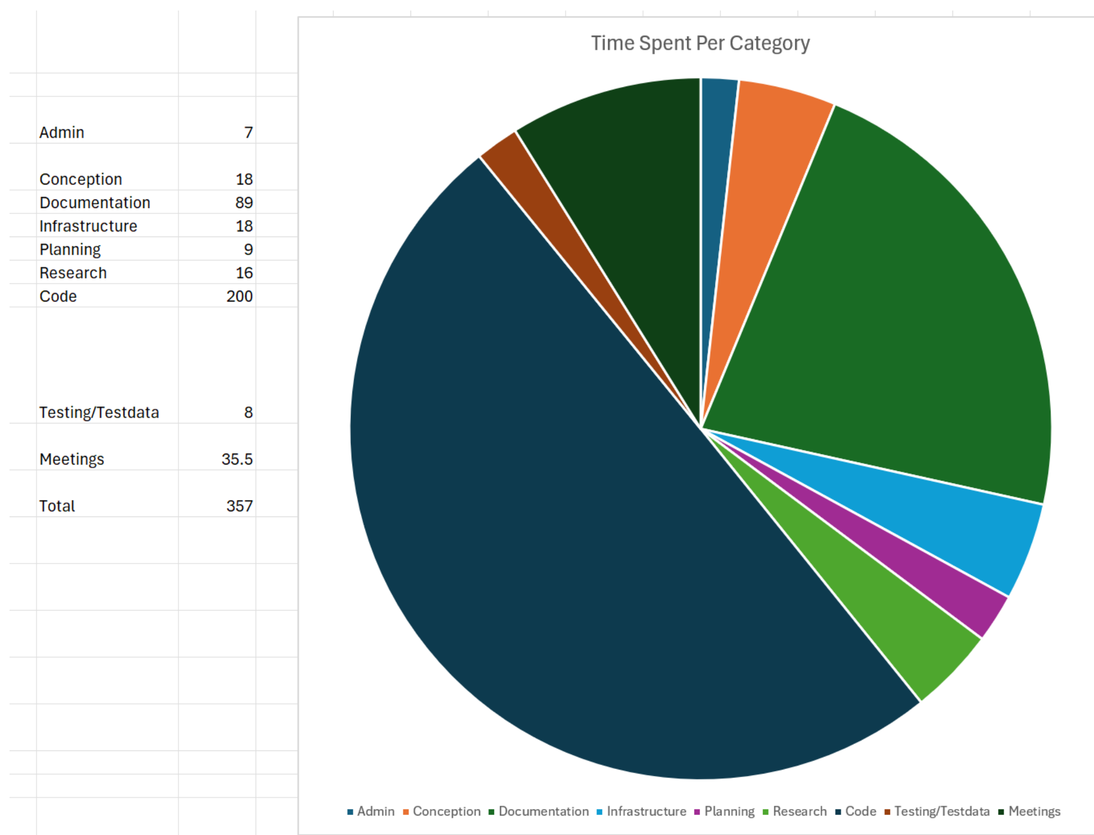
3. Project Monitoring

3.1. Timetracking

Time tracking was conducted to ensure efficient use of resources and to maintain an accurate record of the hours spent on various tasks throughout the project. The pie chart below illustrates the distribution of time across different project activities such as coding, documentation, meetings, research, and other miscellaneous tasks. This detailed tracking helped in identifying areas where more effort was needed and ensured appropriate workload distribution.

3.1.1. Benjamin

Since Benjamin worked on the testing script and implemented the PoC by himself, the time spent coding makes up the majority of his time.



Category	Hours
Admin	7
Conception	18
Documentation	89
Infrastructure	18
Planning	9
Research	16
Code	200
Testing/Testdata	8
Meetings	35.5
Total	357

Figure 3.1.: Timetracking Chart for Benjamin

3.1.2. Fiona

The following graphic depicts the time spent by Fiona on the project, also distinguising the time spent by different categories.

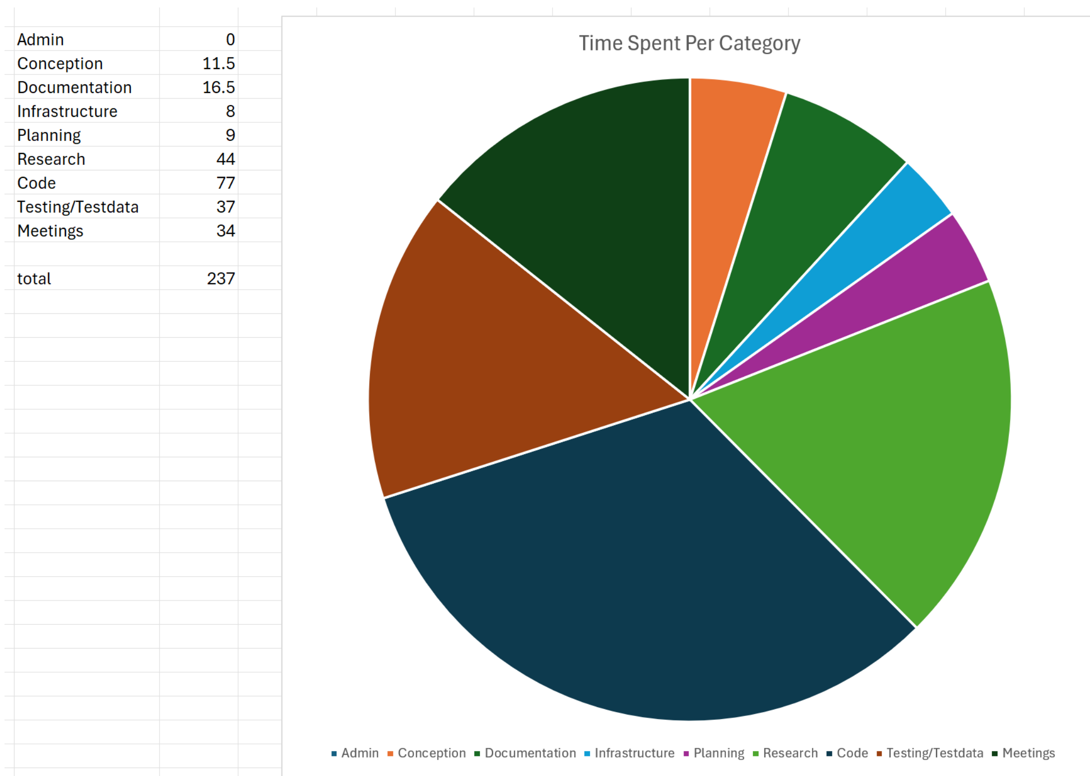


Figure 3.2.: Timetracking Chart for Fiona

3.1.3. Total

The following chart depicts the total time spent on the project by both team members. Since this has been the bachelor thesis of Benjamin, his expected workkload was 12 ECTS points, while Fiona's workload was 8 ECTS points resulting in aproximately 360 and 240 hours of work respectively.

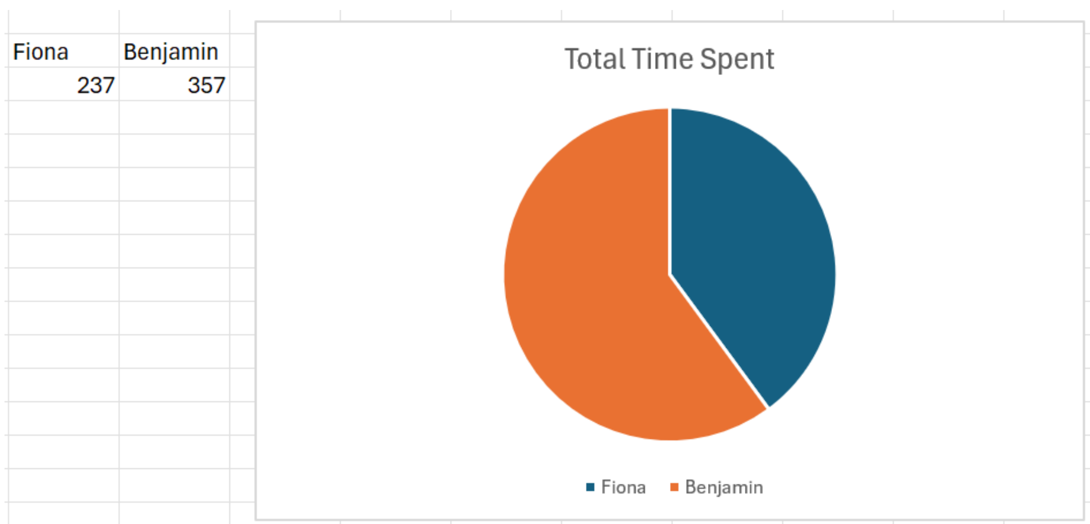


Figure 3.3.: Timetracking Chart for Total

3.2. Project Management

Two-week sprints were used for project management, following the Agile methodology. This approach allowed for regular assessment of progress and quick adjustments to the project plan. Each sprint began with a planning meeting to outline tasks and set goals, followed by standups to discuss progress and address any impediments. The burndown chart below illustrates the progress of the project across multiple sprints. The chart shows the total scope of work and the amount of work completed over time. It should however be taken with a grain of salt, as often times the amount of work was underestimated. Furthermore, some issues were not accounted for in the burndown chart, as we pivoted when new feedback was introduced by the stakeholders. Since the burndown was automatically generated by Azure Devops, it was not possible to adjust it to the new circumstances.

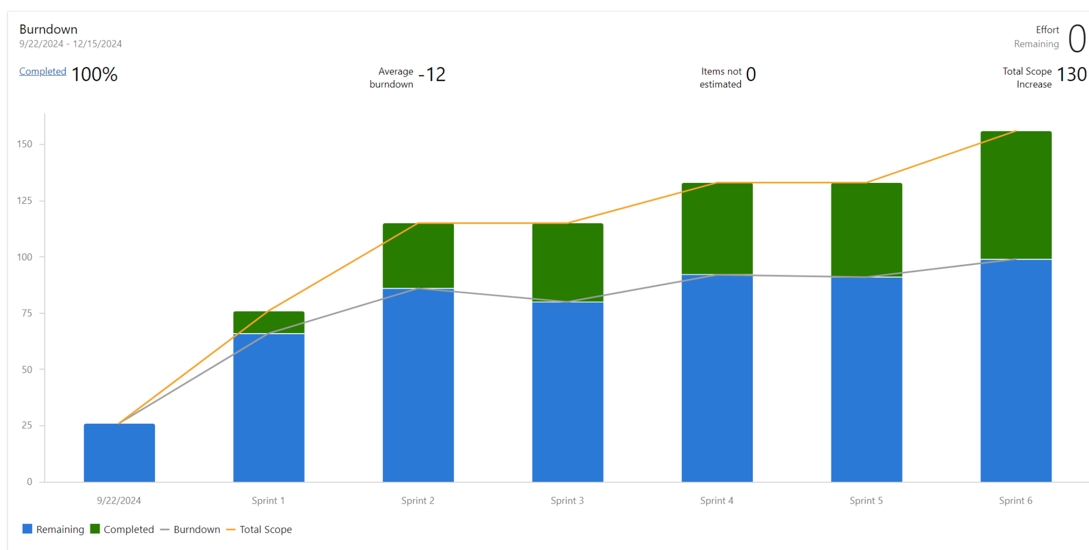


Figure 3.4.: Burndown Chart

Part IV.

Appendix

Acronyms

AI Artificial Intelligence. 2, 9, 10, 15, 18

DGDB DataGovernance Technologies Database. 7, 14, 27, 35, 43–45, 47

DGT DataGovernance Technologies. 2, 3, 5, 14, 38, 50, 51

LLM large language model. v, 2, 3, 5–7, 9, 10, 13–16, 18, 30, 33, 36–47, 51, 53

NL Natural Language. 3, 43, 44

NL-to-SQL Natural Language to Structured Query Language. 2, 15

OST Eastern Switzerland University of Applied Sciences. 2–5, 45, 51

PoC proof of concept. v, 2–5, 8, 10, 11, 14, 15, 18, 38, 40

RAG Retrieval Augmented Generation. 15, 18, 33, 34, 36, 40–42, 44, 46, 49, 50

RAG User Interface. 36

SQL Structured Query Language. 2–5, 14–16, 18, 26, 28–30, 34, 40–42, 44, 50, 52

SUD System under Development. 5, 25, 27, 28, 30, 31

Glossary

Azure DevOps Azure DevOps is a set of development tools used for software development. It is a cloud-based service that provides version control, reporting, requirements management, project management, automated builds, testing and release management capabilities.. 58, 60

ChromaDB ChromaDB is a database that stores the embeddings of RAG.. 59

Git Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files.. 59

LaTeX LaTeX is a high-quality typesetting system; it includes features designed for the production of technical and scientific documentation. LaTeX is the de facto standard for the communication and publication of scientific documents.. 58

Miniconda Miniconda is a free minimal installer for conda. It is a small, bootstrap version of Anaconda that includes only conda, Python, the packages they depend on, and a small number of other useful packages, including pip, zlib and a few others.. 59

Ollama Ollama is a tool that allows easy hosting of open source models.. 18, 59

PEP8 PEP8 is a coding convention for the Python programming language. It is a set of rules that specify how to format Python code for maximum readability.. 58

Pull Request A pull request is a method of submitting contributions to an open development project. It is often used in web development and software development, and it is a way of managing changes to a codebase.. 59

Pylint Pylint is a tool that checks for errors in Python code, tries to enforce a coding standard and looks for code smells. It can also look for certain type errors, it can recommend suggestions about how particular blocks can be refactored and can offer you details about the code's complexity.. 58, 59

Venv Venv is a tool that creates isolated Python environments. It allows you to manage separate package installations for different projects.. 59

Visual Studio C++ Build Tools Visual Studio C++ Build Tools is a standalone build tools package that allows you to build C++ applications.. 59

Bibliography

[1] Farid M. Kedwan. *NLP Application: Natural Language Questions and SQL using Computational Linguistics*. Routledge, 2024. ISBN: 9781032538358. URL: https://www.routledge.com/NLP-Application-Natural-Language-Questions-and-SQL-using-Computational-Linguistics/Kedwan/p/book/9781032538358.

[2] Microsoft. *Fine-Tuning Large Language Models*. Last Accessed: 2025-01-10. 2025. URL: https://learn.microsoft.com/en-us/ai/playbook/technology-guidance/generative-ai/working-with-llms/fine-tuning.

[3] OpenAI. *OpenAI Structured Output Documentation*. Last Accessed: 2025-01-10. 2025. URL: https://platform.openai.com/docs/guides/structured-outputs.

[4] Zimmermann Rohrer. *Natural Language to GraphQL*. Last Accessed: 2025-01-10. 2024. URL: https://eprints.ost.ch/id/eprint/1215/1/FS%202024-BA-EP-Rohrer-Zimmermann-Natural%20Language%20to%20GraphQL.pdf.

[5] Denis Rothman. *RAG-Driven Generative AI: Build custom retrieval augmented generation pipelines with LlamaIndex, Deep Lake, and Pinecone*. 2024. ISBN: 9781836200918; 1836200919. URL: https://libgen.li/file.php?md5=932d6a159ffeb7e270f6a964f11619b8.