UNIVERSITY OF BUCHAREST

FACULTY OF
MATHEMATICS AND
INFORMATICS

INFORMATICS SPECIALIZATION

Master's Thesis

TEXT TO ENTITY FRAMEWORK

Graduate
Andrei-Cristian Rusu

Scientific coordinator
Păduraru Ciprian Ionuț

Bucharest, September 2024

Rezumat

Această teză dezvoltă o soluție Text2EntityFramework (Text2EF), avansând conceptul de Text2SQL prin utilizarea Mapării Obiect-Relationale (ORM) pentru interacțiuni mai versatile cu bazele de date. În timp ce Text2SQL convertește interogările în limbaj natural în declarații SQL, aceasta necesită o înțelegere profundă a schemelor de baze de date. Text2EF simplifică acest proces prin generarea de interogări ORM, aliniate mai bine cu practicile moderne de dezvoltare orientată pe obiecte.

Am transformat cu succes 87% din interogările datasetului Spider folosind SQL2EF și am generat un set de date pentru fine-tuningul Llama3 cu 8B parametri. Aceasta a crescut precizia interogărilor generate de la 31,26% la 51,46%, aproape un salt de 65%, fără fine-tuning extensiv sau inginerie de prompturi.

Cercetarea noastră arată potențialul abordărilor Text2ORM de a spori productivitatea dezvoltatorilor, conectând limbajul natural cu operațiunile de baze de date în cadrele software contemporane.

Abstract

This thesis develops a Text2EntityFramework (Text2EF) solution, advancing the Text2SQL concept by using Object-Relational Mapping (ORM) for more versatile database interactions. Although Text2SQL translates natural language queries into SQL, it frequently necessitates an extensive understanding of the database schemas. Text2EF simplifies this by generating ORM-based queries that better align with modern object-oriented development practices.

We successfully transformed 87% of queries in the Spider dataset using SQL2EF and generated a dataset to fine-tune Llama3 with 8B parameters. This improved the accuracy of generated queries from 31.26% to 51.46%, nearly a 65% increase without extensive fine-tuning or prompt engineering.

Our research demonstrates the potential of Text2ORM approaches to enhance developer productivity by bridging natural language with database operations within contemporary software frameworks.

Contents

Chapter 1

Introduction

1.1 Background

In the last couple of years, software development practices have increasingly emphasized the importance of abstraction and automation. Tools like Object-Relational Mapping (ORM) frameworks, such as Entity Framework (EF), have become integral to modern software development, enabling developers to interact through high-level, object-oriented code with databases, rather than raw SQL queries. This shift is part of a broader trend in software engineering focused on simplifying complex, error-prone tasks by providing more intuitive, developer-friendly tools.

Simultaneously, the NLP field made great strides, particularly in the area of computer comprehension and generation of human language. A notable advancement in this area is the development of techniques that translate queries in natural language into SQL commands, a process known as Text2SQL. This innovation holds the potential to make database interactions more accessible to non-technical users, allowing them to retrieve and manipulate data using plain language without the need to write complex SQL queries.

However, despite the potential of Text2SQL, it remains closely tied to the use of raw SQL, which may be less attractive in development environments that favor ORMs. SQL, being a declarative language, often requires an extensive understanding of the database schemas and query optimization. In contrast, ORMs abstract these complexities, allowing developers to focus on business logic rather than the intricacies of database management.

1.2 Motivation

Despite the advances in Text2SQL, a clear gap exists in integrating NLP with ORM frameworks like Entity Framework. While SQL remains a powerful tool, many modern applications prefer using ORMs due to their advantages in terms of maintainability, readability, and integration with object-oriented programming practices. This raises an

intriguing question: why not extend the capabilities of Text2SQL to generate ORM-compatible code, thereby directly interfacing with the database in a way that aligns with contemporary development practices?

This thesis is motivated by the recognition that translating natural language directly into ORM-based queries could bridge this gap. By doing so, it would enable more seamless integration between NLP-generated database queries and the object-oriented codebases prevalent in modern software development.

The task of Text-to-Entity Framework (Text2EF) involves converting natural language input directly into Entity Framework queries, effectively bypassing the need for intermediate SQL representations. This approach not only better aligns with modern development practices but also has the potential to simplify the development process, improve code maintainability, and make database operations more accessible to developers who may not be experts in SQL.

Furthermore, this motivation is grounded in the desire to push the boundaries of NLP applications in software engineering. By exploring the feasibility and effectiveness of Text2EF, this thesis aims to contribute to the ongoing discourse on leveraging NLP in the context of software development, offering a novel perspective on the intersection of natural language understanding and database management.

1.3 Objectives

This thesis' primary objective is to develop and evaluate a method for translating natural language queries into Entity Framework (EF) code, thereby extending the capabilities of existing Text2SQL models to generate ORM-compatible queries. To achieve this, the thesis will focus on several key objectives.

Firstly, the thesis aims to design and implement a methodology for converting SQL queries into EF code (SQL2EF), the main reason being that there are no Text2EF datasets out there.

Secondly, it seeks to create and evaluate a Text2EF dataset by transforming existing Text2SQL datasets - particularly, the Spider dataset.

Thirdly, the thesis will experiment with a Llama3 model, both with and without fine-tuning, to assess its ability to generate accurate EF code.

Fourthly, the thesis will analyse the experiment outcomes and point out its advantages and disadvantages.

Finally, based on the findings, the thesis will propose avenues for future research, including enhancements to the SQL2EF transformation process, expansion of the dataset, and further integration with more advanced NLP models.

1.4 Structure

This thesis is structured into seven key chapters, with each chapter focusing on a distinct aspect of the research. The following chapters cover the following topics:

The second chapter, Literature Review, examines relevant literature for the Text2EF task, identifying gaps in the current research and establishing the context for this study.

The third chapter, Technologies and Methodologies, describes the technologies and methodologies selected for this research. It provides a rationale for choosing specific tools, frameworks, and techniques, detailing their contribution to the development and evaluation of the Text2EF task.

The fourth chapter, Methodology, outlines the methodology used to develop the SQL2EF transformation algorithm, the creation of the Text2EF dataset, and the evaluation methods used to assess the model's effectiveness.

The fifth chapter, Experiments and Results, presents the findings from the conducted experiments. It includes a detailed analysis of the performance of the SQL2EF algorithm as well as an evaluation of the model's effectiveness.

The sixth chapter, Implementation, delves into the technical aspects of this research. It explains the process of conducting Entity Framework query testing and details the methods used to convert SQL to Entity Framework code.

Finally, the seventh chapter, Conclusions, summarizes the thesis's contributions, reflects on the significance of the research, and offers concluding remarks.

1.5 Contributions

This thesis makes several significant contributions to the field of NLP and software engineering. First, it introduces the first `SQL2EF` algorithm, a novel approach for transforming SQL queries into Entity Framework representations. This algorithm paves the way for new methodologies in automatic database schema generation from natural language. Second, this work presents the first `Text2EF dataset`, specifically designed to facilitate the training and evaluation of models capable of converting textual descriptions into Entity Framework code. These contributions provide the foundation for future advancements in automatic code generation and database design.

Chapter 2

Literature review

2.1 Introduction to Semantic Parsing and Text2SQL

Semantic parsing has emerged as a pivotal area within the field of NLP, involving the transformation of natural language into machine-readable representations. Among the various subfields, Text2SQL has gained significant attention. Text2SQL represents the task of converting questions in natural language into structured SQL queries, enabling direct interaction with relational databases through natural language. This task presents substantial challenges, particularly due to the complexity of natural language, the necessity for precise database interactions, and the cross-domain nature of databases, which often vary significantly in structure and schema.

2.1.1 The Importance of the Spider Dataset

The *Spider* dataset [4] is recognized as one of the most impactful contributions to the Text2SQL field. It was created to overcome the limitations of earlier datasets, which were often restricted to single domains or simpler queries. Spider is a human-annotated, large-scale dataset, specifically crafted to aid in complex and cross-domain semantic parsing tasks. It consists of 10,181 questions paired with 5,693 unique and intricate SQL queries, covering 200 databases across multiple domains. This extensive range makes Spider a vital resource for developing and rigorously testing models that need to manage the complexity and diversity found in real-world database queries.

2.1.2 Contributions

This dataset introduces several critical features that significantly advance the Text2SQL task.

Complexity and Diversity: Unlike earlier datasets, Spider includes a broad spectrum of SQL queries, encompassing nested queries, joins, aggregations, and other sophisticated

SQL structures. This level of complexity is essential for training models capable of handling the intricate logic often required in practical applications.

Cross-Domain Capability: The inclusion of databases from various domains ensures that models trained on Spider can generalize effectively across different contexts. This cross-domain aspect is crucial for developing robust models that must function in environments where the database schema may be unknown or may differ significantly from one instance to another.

Human-Labeled Data: The dataset is entirely human-labeled, ensuring a high standard of annotation quality. This is particularly important for training models that need to comprehend nuanced language constructs and translate them accurately into SQL queries.

2.1.3 Impact on Text2SQL Research

The introduction of the Spider dataset has had a profound impact on the Text2SQL research landscape. It has become the benchmark for evaluating model performance, stimulating the development of more sophisticated algorithms designed to tackle the complexities introduced by the dataset. Researchers have leveraged Spider to explore advanced techniques such as schema linking, entity recognition, and enhanced natural language understanding, expanding the limits of what Text2SQL models are capable of achieving.

2.1.4 Relevance to Text2EF Task

In the context of this thesis, which focuses on transforming a Text2SQL dataset into a Text2EntityFramework (Text2EF) dataset, the Spider dataset provides a foundational reference. By analyzing the challenges and solutions presented in the Spider dataset, we can draw critical parallels to the challenges involved in generating Entity Framework (EF) models from natural language inputs. The insights derived from the Text2SQL task—particularly those related to handling complex queries, ensuring cross-domain generalization, and maintaining high annotation quality—are directly applicable to the development of a Text2EF application.

The transformation from Text2SQL to Text2EF introduces new dimensions, such as mapping natural language to Entity Framework syntax rather than SQL queries. However, the fundamental challenges of understanding and parsing natural language, dealing with complex logic, and ensuring adaptability across various domains remain consistent. The methodologies and approaches developed in response to the Spider dataset provide valuable guidance as we address the unique challenges inherent in Text2EF, particularly in managing complex entity relationships and domain-specific models.

2.2 LLMs and LLaMA 3

2.2.1 Introduction

LLMs have achieved remarkable progress in the realm of NLP, showcasing abilities that span from basic text prediction to complex tasks like code creation and logical reasoning. These models, predominantly based on transformer architectures, are trained on extensive datasets, enabling them to recognize intricate language patterns, understand context, and generate text that closely resembles human writing. The emergence of LLMs has transformed NLP, providing models that not only comprehend language but also generate it with a sophistication that was once beyond reach.

2.2.2 The Evolution of Transformer Architecture

The foundation of LLMs is built on the self-attention mechanism, introduced for the first time by Vaswani et al. (2017) [2]. This mechanism enables the model to assess the significance of words relative to the other words in the sentence, which is crucial for tasks requiring contextual understanding. Over time, the evolution of models has led to the development of transformer-based architectures, including GPT, BERT, and more recent advancements like GPT-3 and GPT-4, resulting in increasingly sophisticated LLM's.

2.2.3 Introduction to LLaMA 3

LLaMA 3 [1], the latest development in this series, marks a substantial advancement in the capabilities of LLMs. This generative model is designed to push the limits of both scale and performance in LLMs. Building on the groundwork of its predecessors, LLaMA 3 incorporates numerous innovations, including enhanced data preprocessing methods, more efficient training algorithms, and an architecture optimized for understanding and generating complex text. Trained on a broad and diverse dataset, LLaMA 3 is better equipped to generalize across various domains.

2.2.4 Importance of LLaMA 3 for the Text2EF Task

Within the scope of this thesis, LLaMA 3's relevance is particularly significant due to its ability to handle tasks requiring deep linguistic and contextual understanding, such as converting a Text2SQL dataset into a Text2EF (Entity Framework) dataset. This transformation necessitates not only syntactic changes but also a comprehension of the underlying semantics of the data, a task that LLaMA 3 is well-suited to accomplish.

2.3 SQLNet: Converting Natural Language into Structured Queries without the Need for Reinforcement Learning

2.3.1 Introduction

The task of converting natural language questions into SQL has been a persistent challenge in the field of NLP. This problem is especially difficult because it involves transforming unstructured, often ambiguous, natural language into a structured, precise format like SQL. Solving this challenge holds considerable potential for enhancing human-computer interaction, as it would allow users to access information from databases without the need to be proficient in database query languages.

In this context, the paper "SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning" by Xu et al. (2017) [3] presents a novel approach to the problem of Text2SQL conversion, introducing SQLNet, a neural network-based model designed to generate SQL queries directly from questions written in natural language. This work is highly relevant to the broader field of Text2SQL tasks and provides a foundation upon which further research, including the transformation of Text2SQL datasets into Text2Entity Framework (Text2EF) datasets, can be built.

2.3.2 Model Architecture and Key Contributions

SQLNet distinguishes itself from prior approaches by avoiding the use of reinforcement learning, which was a common method employed in earlier models to train the system to generate correct SQL queries. Reinforcement learning, while powerful, introduces significant complexity in training and often suffers from issues such as reward sparsity, which can hinder the model's performance and convergence.

Instead, SQLNet employs a novel sketch-based approach, where the structure of the SQL query is predicted separately from its components. Specifically, SQLNet breaks down the SQL query generation task into predicting the key components of the SQL structure, such as SELECT, WHERE, and aggregation functions, before filling in the specific details such as column names and conditions. This decomposition allows SQLNet to focus on accurately predicting each part of the SQL query, reducing the overall error rate and improving performance on complex queries.

The primary contributions of SQLNet include the introduction of a sketch-based decoder, which simplifies the SQL generation process and enhances the model's capacity to handle a broad spectrum of queries, the elimination of reinforcement learning from the training process, which simplifies model training and improves reliability, and a demonstrated improvement over previous SOTA models on standard Text2SQL benchmarks,

showcasing the efficacy of the proposed approach.

2.3.3 Relevance to Text2EF Task

The methodologies and innovations presented in SQLNet are directly relevant to the task of Text2EF. The sketch-based approach employed by SQLNet can serve as a guiding framework for similar decompositional methods in the Text2EF task, where the generation of Entity Framework (EF) code from natural language may require breaking down the problem into predicting different aspects of the EF structure separately before assembling them into a coherent whole.

Furthermore, SQLNet's avoidance of reinforcement learning highlights the potential for simplifying the training process in Text2EF applications. By leveraging techniques that do not rely on reinforcement learning, we can achieve more stable and reliable training outcomes, which is crucial for developing robust models capable of handling the complex nature of EF generation tasks.

The success of SQLNet in the Text2SQL domain underscores the importance of focusing on model architecture and task decomposition when addressing the challenges of natural language to code transformation tasks.

Chapter 3

Technologies and Methodologies Used

This section offers an overview of the core methodologies and technologies employed in the development of the Text2EF application. The project leverages several programming languages and tools, each serving a distinct purpose in the workflow.

3.1 Programming Languages and Tools

3.1.1 C# and Entity Framework

Entity Framework (EF) is an open-source Object-Relational Mapping (ORM) framework designed for .NET applications, enabling developers to work with databases through .NET objects. In this project, C# was utilized to create a minimalist Entity Framework project that served as a controlled environment for testing and validating the transformation of SQL queries into EF-compatible expressions. The choice of C# and EF was driven by the widespread use of EF in enterprise environments, making it a suitable target for transforming SQL queries.

3.1.2 Rust and SQL2EF Application

Rust was the primary programming language used to develop the SQL2EF application, the core component responsible for converting SQL queries into EF-compatible expressions. Rust was chosen for its performance, safety guarantees, and powerful tooling ecosystem, which made it ideal for handling the complex transformations required by this project. The application relies heavily on the ‘sqlparser‘ crate, an external Rust library designed to parse SQL queries into their abstract syntax tree. The AST representation of SQL queries forms the basis for translating these queries into EF LINQ expressions.

3.1.3 Python and Bash Scripting

Python and Bash were employed for various auxiliary tasks that supported the main development process. Python scripts played a crucial role not only in the preprocessing and postprocessing stages of the dataset transformation - merging the split Spider datasets and creating the final Text2EF datasets - but also in training and testing the Llama3 models. The ease of handling data with Python, along with its rich ecosystem of libraries, made it the ideal choice for these tasks. Additionally, Bash scripts were utilized primarily for automating the scaffolding of the SQLite databases from the dataset.

3.2 Workflow and Integration

The workflow of the Text2EF project followed a systematic approach, starting with the preparation and consolidation of the dataset. The original Spider dataset, which was initially split, was merged into a single, unified dataset. This step was critical for ensuring consistency and ease of use in subsequent phases.

Following the dataset preparation, a minimalist Entity Framework (EF) project was set up. This involved scaffolding the SQLite files from the Spider dataset, establishing the necessary structure for working within the EF context.

With the EF project in place, the SQL2EF converter was built. This tool was responsible for converting SQL queries from the dataset into Entity Framework-compatible LINQ expressions.

Once the SQL2EF converter was operational, the Text2EF dataset was constructed. This dataset was tested within the EF project sandbox to ensure that the converted queries executed correctly and produced the expected results. This validation step was crucial for verifying the correctness of the SQL2EF application.

Subsequently, datasets for training and testing the Llama3 model were created, by providing a rich context for Llama3, facilitating effective training. The Llama3 model was then trained and tested, both with and without fine-tuning, to assess its performance and ability to generalize from the Text2EF dataset.

3.3 Challenges and Considerations

The transformation of SQL queries into EF-compatible expressions presents several challenges, particularly in handling the differences in abstraction between SQL, which is a declarative language, and LINQ, which is more tightly integrated with object-oriented paradigms. Moreover, the project had to account for various SQL dialects and their idiosyncrasies. While the ‘sqlparser‘ crate provided a robust foundation for parsing SQL

queries, careful handling was required to ensure that the AST could be accurately transformed into EF expressions.

Chapter 4

Methodology

4.1 The dataset

For this task, we have selected the dataset introduced in the paper titled ”Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text2SQL Task” [4]. This dataset is extensive, covering complex and cross-domain Text2SQL scenarios. It includes 10,181 questions and 5,693 unique, intricate SQL queries distributed across 200 databases, each comprising multiple tables. Each entry within the dataset consists of the database name, a natural language query, and the corresponding SQL query. Additionally, each database is accompanied by an SQLite file that contains the schema and a set of entries, which will later be used to verify the outcomes of the SQL queries against the generated EF code.

```
{
  "db_id": "cre_Drama_Workshop_Groups",
  "question": "What is the description and code of the type of
   ↪  service that is performed the most often?",
  "query": "SELECT T1.Service_Type_Description,
   ↪  T1.Service_Type_Code FROM Ref_Service_Types AS T1 JOIN
   ↪  Services AS T2 ON T1.Service_Type_Code = T2.Service_Type_Code
   ↪  GROUP BY T1.Service_Type_Code ORDER BY COUNT(*) DESC LIMIT 1"
}
```

Listing 1: Dataset entry example

In the original format, the dataset comes splitted in ‘dev‘ and two ‘train‘ files - one created by the people that created Spider, and one by the other contributors. We have merged all these datasets in one, for the purpose of being able to create a more appropriate split for our task at hand.

4.2 Entity Framework

We had set up a simple project that was used for testing the correctness of the generated queries by algorithmically generating the code that would run the SQL query against the generated EF code and checking if the results were the same. We also set up a script to scaffold the SQLite files and generate the associated models and context files for each of the databases.

4.2.1 Testing the queries

In the process of transforming a Text2SQL dataset into a Text2EF one, a critical aspect involves ensuring that the Entity Framework (EF) code generated from natural language queries produces results consistent with those from corresponding SQL queries. However, directly comparing SQL queries and EF code execution results presents significant challenges due to differences in how SQL and LINQ (Language-Integrated Query, used by EF) handle data types, formatting, and execution contexts. To address this, an algorithm for comparing the two results was developed. The solution involves standardising and normalising the results from both SQL and EF before comparison.

One of the key challenges in comparing SQL and LINQ results is the inherent differences in data types and formatting. For instance, SQL might return a VARCHAR as a string, while the corresponding EF query could return it as a Decimal or Int. To mitigate these issues, the testing method includes normalization logic to align the formats and data types between the SQL and LINQ results.

Despite the normalization steps, there are inherent limitations to this comparison approach. Certain edge cases, such as the conversion of empty strings to zero values or the automatic type conversion done by EF, require careful consideration. In some instances, SQL queries might return no results, while the corresponding LINQ query might return a single result with a default value (e.g., zero). These situations necessitate additional checks within the testing framework to ensure that such cases are handled appropriately.

4.3 SQL2EF

We have developed an algorithm written in Rust, based on the ‘sqlparser‘ crate, that effectively transforms SQL `SELECT` statements into Entity Framework code, by creating the associated AST (Abstract Syntax Tree) of the query and parsing it.

4.3.1 Schema mapping

In the process of transforming SQL datasets into Entity Framework (EF) datasets for our Text2EF application, one of the key challenges is aligning the differences between SQL

```json
{
  "context": "BrowserWebContext",
  "tables": {
    "browser": {
      "table": "Browsers",
      "columns": {
        "id": {
          "name": "Id",
          "field_type": "Int",
          "is_optional": false,
          "column_type": "Int"
        },
        "name": {
          "name": "Name",
          "field_type": "String",
          "is_optional": true,
          "column_type": null
        }
      }
    }
  }
}
```

Listing 2: Schema mapping example

database schemas and the structure expected by C# code in EF. These differences arise because SQL and C# use different conventions and data types, meaning that the names of tables and columns in the database often do not directly match with those used in EF models. This is where schema mapping comes into play.

Schema mapping is the process of creating a correspondence between the schema of a SQL database (i.e., its tables, columns, and data types) and the Entity Framework models in C#. Essentially, it acts as a bridge between the two very different structures. Without schema mapping, it would be difficult, if not impossible, to automatically transform SQL-based data interactions into their EF equivalents. This is because of how the data is represented and interacted with in SQL, which is quite different from how it's done in C#. For example, a column named user_id in a SQL table might be mapped to UserId in EF, and a SQL VARCHAR type might correspond to both a string or an integer in C#.

4.3.2 Core Functionalities

The application currently supports a comprehensive set of SQL functionalities and constructs, effectively translating them into the EF context. The core operations begin with

projections, where the application handles the selection of fields from a database table, including those with aliases. It extends support to various aggregating functions such as `min`, `max`, `count`, `avg`, and `sum`, allowing for complex calculations within queries. Furthermore, binary operations, including expressions such as `A - B`, are accurately translated to their EF equivalents.

Projections can include a combination of these elements, and the application is capable of managing the `DISTINCT` keyword. Additionally, wildcard selections are supported, translating the SQL `*` to EF's corresponding constructs.

Filtering of data through `WHERE` clauses is another critical feature, with the application supporting a wide range of comparison operators, including both unary and binary types such as ==, !=, <, and >. The translation of SQL's `LIKE` function and `IN` and `BETWEEN` clauses are handled too.

A significant aspect of the `WHERE` clause functionality is the handling of subqueries. The application supports the inclusion of subqueries, allowing complex conditions such as `WHERE x > (SELECT ...)` to be correctly represented within EF. Additionally, when dealing with aggregated fields within these clauses, the application intelligently determines whether the field has already been selected and reuses it if possible, otherwise calculating it as needed. Alias management within `WHERE` clauses is also handled.

The application also supports grouping operations via the `GROUP BY` clause, enabling the aggregation of data based on one or more fields.

When it comes to handling joins, the application is versatile and robust. It supports inner joins and can interpret different SQL join syntaxes, whether they involve chaining multiple tables together or applying conditions across multiple `ON` clauses. The order of joins is determined based on the conditions specified, ensuring a correct syntax. The application also manages implicit joins by translating them into EF's `SelectMany` method, and it handles joins involving multiple fields by comparing field pairs using constructs such as `new { Pair1 = , Pair2 = ... }`.

In addition to these fundamental operations, the application supports a variety of set operations between two SQL `SELECT` statements, including `INTERSECT`, `UNION`, `IN`, `NOT IN`, and `EXCEPT`.

Key SQL keywords such as `LIMIT`, `ORDER BY`, and `HAVING` are also supported. The `LIMIT` keyword is mapped to the `Take` method in EF, allowing the query to return a specified number of records. The application carefully manages the `ORDER BY` clause, particularly in cases involving multiple fields or ordering by functions, ensuring the correct syntax is applied for each case. The `HAVING` clause is handled with attention to detail, considering both scenarios where the field is selected first and then filtered, and where it is calculated within the `HAVING` clause itself.

4.3.3 Complex and Interesting Cases

Several complex cases highlight the sophistication and depth of the SQL2EF application.

For example, in projection operations, SQL automatically sorts fields when aggregating data using `min/max`. This behavior necessitates additional sorting within EF to ensure the correct results are returned. The application addresses this by applying an `.OrderBy()` on the aggregated field before performing a `.First()` operation, ensuring consistency with SQL's behavior.

Another notable case involves type casting. When performing operations such as averaging on a field that is a `VARCHAR` in SQL but is treated as a `string` in C#, the application needs to cast the field to a numeric type such as `double` to ensure a valid syntax.

Handling optional fields presents another challenge, particularly when performing set operations like `INTERSECT` or `EXCEPT` between queries where one includes optional fields, and the other does not. The application solves this issue by implementing a check on the optional fields, using `.HasValue` to ensure that only non-null fields are selected, then selecting the fields with `.Value`, thus making the field non-optional within the EF context.

In scenarios where set operations are performed on a single item, such as `A.Intersect(B)` with only one selected item, EF requires a different approach. The application wraps the EF code within a `new List<return type> { ... }` to correctly execute the operation.

Field aliasing and deduplication are also critical aspects handled by the application. When fields from different tables share the same name (e.g., `T1.Name` and `T2.Name`), the application assigns unique names based on the table alias, such as `T1Name` and `T2Name`, ensuring that the generated EF query is unambiguous and error-free.

4.4 Training and Testing with Llama3

4.4.1 Dataset Preparation

To effectively train and test the Llama3 model, we created two datasets from the final Text2EF dataset by applying an 80-20 split to each individual database. By using this approach we ensure that the model is trained on queries from each database, providing a more balanced learning experience and preventing scenarios where the model encounters an unfamiliar database for the first time. Each dataset entry was formatted to include the necessary context, such as the database schema, models, and the natural language query.

The datasets were uploaded to `HuggingFace`, a widely used platform for sharing datasets, models, and other AI resources within the machine learning community.

4.4.2 Model Training with UnslothAI

Overview of the Training Process

To fine-tune our Text2EF model, we utilized *Unsloth*, an advanced AI tool specifically designed to enhance training speed and memory efficiency. Unsloth achieves these improvements by leveraging a combination of sophisticated mathematical techniques and custom GPU kernels, implemented using OpenAI's Triton language. These optimizations allow for more efficient use of computational resources, enabling the training of large models with reduced overhead. We have selected the `Meta-Llama-3.1-8B` model, which offers a balanced trade-off between performance and computational demands. This model was chosen due to its suitability for our specific task, providing a solid foundation for further fine-tuning.

Introduction to SFTTrainer

The training process was carried out using a tool called `SFTTrainer`, which is designed to streamline the *Supervised Fine-Tuning (SFT)* of LLMs. `SFTTrainer` enhances the fine-tuning process by allowing the model to better adapt to specific datasets while maintaining efficiency in computational resources.

This tool provides extensive control over various hyperparameters that are crucial to the model's learning process. Among these hyperparameters are batch size, gradient accumulation steps, warmup steps, learning rate and the total number of training steps. By fine-tuning these parameters, we optimized the training process to meet the particular demands of our task.

Hyperparameter	**Value**
Batch Size	2
Gradient Accumulation Steps	4
Learning Rate	2e-4
Warmup Steps	5
Max Steps	60
Optimizer	`adamw_8bit`

Table 4.1: Key SFTTrainer Hyperparameters for Initial Testing

The **Batch Size** was set to 2, meaning that two samples were processed before updating the model's parameters. We selected a smaller batch size in the initial phase to manage memory efficiently. The **Gradient Accumulation Steps** were configured to 4, meaning gradients were accumulated over four steps before performing a backpropagation, effectively treating the batch size as 8 (2 samples $\times$ 4 steps).

The **Learning Rate** was initially set to 2×10^{-4}, which is relatively high for early experiments, allowing the model to make substantial updates during each step. This was

combined with **Warmup Steps** of 5, which helps in gradually increasing the learning rate during the first 5 steps, thus minimizing instability during the initial training phase. Additionally, the **Max Steps** were set at 60, meaning the model underwent 60 gradient updates before this phase of training concluded. We employed the `adamw_8bit` optimizer, a memory-efficient variant of the Adam optimizer, particularly suited for handling large models like `Meta-Llama-3.1-8B`.

Adjustments and Further Fine-Tuning

Following the initial testing phase, we made several adjustments to further explore the model's potential. Specifically, we increased the warmup steps to 500 and extended the maximum training steps to 200. These changes were aimed at providing a more gradual increase in the learning rate and allowing the model more time to refine its understanding and improve its performance. The extended warmup phase supports stable convergence by allowing the model to adapt slowly to the learning process, while the additional training steps gave the model more opportunities to enhance its capabilities.

Parameter-Efficient Fine-Tuning (PEFT)

To optimize the training process further, we implemented *Parameter-Efficient Fine-Tuning (PEFT)*. PEFT is a strategy that focuses on updating a small, selected subset of model parameters rather than all of them. This approach significantly reduces computational requirements and is particularly advantageous when fine-tuning large models, as it allows for effective adaptation with minimal resource usage.

In our PEFT implementation, we configured several key parameters to balance computational efficiency with fine-tuning effectiveness. The PEFT parameters used are detailed in the following table:

PEFT Parameter	**Value**
Rank (`r`)	16
LoRA Alpha (`lora_alpha`)	16
LoRA Dropout (`lora_dropout`)	0
Bias (`bias`)	None
Target Modules	`o_proj, v_proj, k_proj, q_proj, up_proj, down_proj, gate_proj`
Gradient Checkpointing	Unsloth

Table 4.2: Key PEFT Parameters for Fine-Tuning

The **Rank** parameter (`r`) was set to 16, determining the dimensionality of the low-rank matrices used in the fine-tuning process. A rank of 16 was chosen to balance the complexity of updates with computational efficiency.

The **LoRA Alpha** (`lora_alpha`) was also set at 16, which scales the updates applied during fine-tuning, ensuring proportional and effective changes to the model's parameters. The **LoRA Dropout** (`lora_dropout`) was set to 0, meaning no dropout was applied, allowing all parameter changes to be fully utilized.

The **Bias** was set to "None," which was an optimized choice for our specific use case, meaning the model's bias terms were not updated during fine-tuning. This further reduced the number of adjustable parameters, enhancing the efficiency of the PEFT process.

The fine-tuning targeted specific **Modules** within the model, namely `q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, and `down_proj`. These modules are integral to the projection layers within the Transformer architecture, playing a key role in the attention mechanism.

Finally, we enabled **Gradient Checkpointing** with the "Unsloth" option. This technique lowers memory consumption by retaining only a portion of the intermediate activations during training, allowing for processing larger batch sizes and longer sequence lengths without surpassing GPU memory limits.

Chapter 5

Experiments and Results

5.1 SQL2EF

The SQL2EF application was tested using a comprehensive dataset consisting of 4285 unique SQL queries. The objective was to transform these queries into equivalent Entity Framework (EF) code and evaluate the correctness of the transformation by comparing the results generated by the EF queries against those produced by the original SQL queries.

Result Type	**Count**	**Percentage**
Total Queries	4579	-
Passed	3984	87.00%
Build failed	275	6.01%
Code failed	120	2.62%
Algorithm failed	200	4.37%

Table 5.1: Results of the SQL2EF algorithm ran on Spider

Out of the 4285 queries, 3984 were successfully transformed into EF code, and these transformations not only compiled correctly but also returned identical results when executed against the database. This high success rate indicates that the application is capable of handling a substantial portion of real-world SQL queries with precision.

However, the transformation process encountered several challenges. A total of 275 queries were correctly transformed into EF code, but the build failed in C#. These failures were primarily due to cases that the current version of the application has not yet been designed to handle, such as complex relationships or unsupported features in the C# environment.

Additionally, there were 200 queries that did not pass through the SQL2EF pipeline at all, due to reasons such as syntax elements not accounted for when parsing the AST. These cases highlight the current limitations of the application's parsing logic.

Moreover, 120 queries were successfully transformed, and the EF code compiled without errors. However, issues emerged during the testing phase, including incorrect result

comparisons by the EF testing code, discrepancies in how SQL and EF handle certain operations, and for 81 of these queries, the discrepancies were attributed to underlying SQL errors, such as invalid data in the database.

These results do not undermine the overall success of the SQL2EF application. Rather, they highlight specific areas for further development and refinement. The cases that currently cause build failures, skipped pipeline processing, or incorrect query outputs provide valuable insights into the application's current boundaries and inform the next steps in its evolution.

5.2 Text2EF

We initially ran the model on the prepared dataset to evaluate its performance. The results were assessed across three distinct scenarios: first, using the model without any fine-tuning; then, with a minimal amount of fine-tuning applied; and finally, with additional fine-tuning and warmup steps.

As can be seen in Figures 5.1 and 5.2, the loss for the fine-tuned model started at 0.5598 and ended at 0.0951, while for the fine-tuned model with additional steps and warmup, the loss started at 0.5494 and ended at 0.0514.

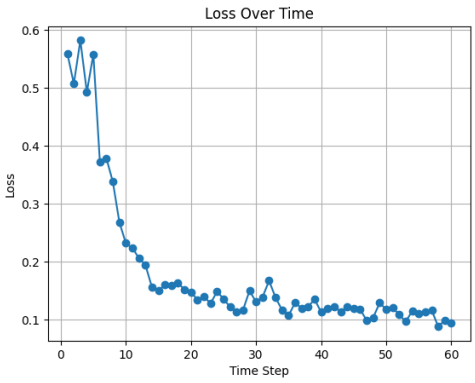

Figure 5.1: The loss for the fine-tuned model.

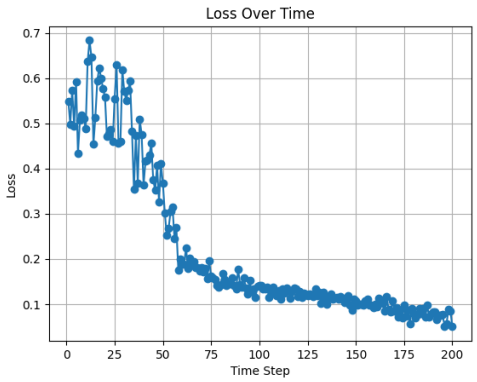

Figure 5.2: The loss for the fine-tuned model with additional steps and warmup.

5.2.1 Performance Analysis

The initial evaluation of the model, as shown in Table 5.2, reveals a clear improvement in performance as the model undergoes fine-tuning and additional training. The simple model, which had not been fine-tuned, exhibited a high percentage of total errors (68.74%), with a significant portion being syntactic errors (43.07%). Semantic errors

were also notable, contributing to 25.67% of the total errors, resulting in a passing percentage of only 31.26%.

Metric	Simple Model	Fine-Tuned Model	Fine-Tuned + More Steps
Passed (%)	31.26	47.78	51.46
Total Errors (%)	68.74	52.22	48.54
Syntactic Errors (%)	43.07	21.22	14.99
Semantic Errors (%)	25.67	31.00	33.55

Table 5.2: Performance Metrics of `Meta-Llama-3.1-8B` on Text2EF Dataset

Fine-tuning the model on the Text2EF dataset led to a marked reduction in total errors to 52.22%. This improvement was primarily driven by a substantial decrease in syntactic errors, which dropped to 21.22%. However, the semantic errors slightly increased to 31.0%, indicating that while the model's syntactic capabilities improved, challenges in understanding the semantics persisted. Consequently, the passing percentage improved significantly to 47.78%, nearly a 50% increase compared to the simple model.

Further fine-tuning, including additional training steps and a warmup phase, resulted in continued performance gains, with the total error percentage dropping to 48.54%. Syntactic errors decreased further to 14.99%, highlighting the model's improved grasp of syntax. However, this improvement came with an increase in semantic errors to 33.55%, suggesting that while the model became more adept at handling the syntactic structures of C# Entity Framework (EF), it still struggled with fully understanding the underlying semantics. The highest passing percentage of 51.46% reflects these overall gains, yet also underscores the ongoing challenge of semantic comprehension in the Text2EF task.

5.2.2 Error Analysis

A detailed analysis of the generated queries revealed that the most common error type was related to generating incorrect results, accounting for 60.7% of the failed queries. This indicates that the model, despite improvements, often failed to correctly map the input text to the appropriate EF entities and relationships, leading to erroneous outputs (see Figure 5.3).

Another frequent error was the occurrence of CS1061 errors, which happen when the model attempts to call a method or access a class member that does not exist. This suggests that the model struggles with correctly associating schema mappings with the appropriate EF classes and methods. For example, even when the schema mapping explicitly states that an SQL table like "neighbourhood" is mapped to "Neighbourhoods," the model occasionally fails to apply this mapping correctly, resulting in syntactically or semantically invalid queries (as illustrated in Figure 5.4).

Additionally, there were errors related to misunderstanding the C# syntax, such as generating a completely incorrect `.Where` clause like the following:

```
....Where(group => group.Select(row => row.Stars).Count() >= 2).ToList();
```

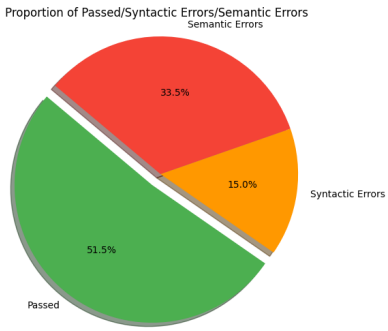

Figure 5.3: Proportion of Passed Cases Versus Syntactic and Semantic Errors

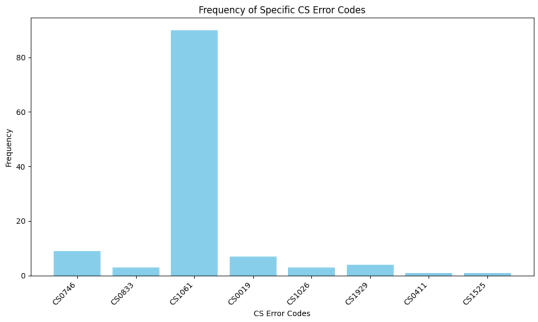

Figure 5.4: Distribution of Different Error Codes

There were also instances where the model generated nonsensical constructs, such as multiple `.Select` statements chained together without any meaningful structure or purpose.

Further, the model exhibited errors that suggest a combination of schema miscomprehension and C# syntax misunderstanding. For instance, CS0746 errors occurred, which are typically caused by missing or incorrectly placed constraints in method calls, and CS0019 errors, which involve incompatible type comparisons, such as comparing strings to objects directly without proper conversion or parsing.

The model also generated CS0833 errors, which arise when there are conflicting or duplicate property definitions within the same context, and CS1929 errors, which occur when aggregation functions are incorrectly applied without considering the types of fields being aggregated. These errors highlight the model's difficulties in correctly interpreting and generating EF-specific C# code, particularly in complex query scenarios.

5.2.3 Implications and Future Work

The findings from this analysis indicate that while the `Meta-Llama-3.1-8B` model demonstrates potential for the Text2EF task, significant challenges remain, particularly in semantic comprehension. The observed trade-offs between reducing syntactic errors and increasing semantic errors suggest that the model is beginning to understand the structural aspects of EF queries, but it still lacks a deep understanding of the underlying domain logic.

This analysis suggests that further improvements could be achieved through better prompt engineering and additional fine-tuning focused specifically on C# EF syntax and semantics. Training the model with a larger and more diverse set of EF queries, including

complex and edge cases, could help address some of the semantic errors observed. Moreover, incorporating more sophisticated schema-mapping techniques and context-awareness into the model's training process might enhance its ability to correctly associate and generate accurate EF queries.

Chapter 6

Implementation

6.1 EF Testing

The implementation of Entity Framework (EF) testing in our SQL2EF application is a critical component, aiming to validate the accuracy of LINQ queries generated from textual input against their SQL equivalents. The testing mechanism has been designed to handle a variety of scenarios, ensuring that the generated LINQ queries return results that are consistent with those of the corresponding SQL queries. However, the process is not without its challenges, particularly in areas such as data normalization, type conversion, and result comparison.

6.1.1 Normalization of Date and Time Values

One of the significant challenges in the EF testing framework is the normalization of date and time values. Currently, all dates are normalized to the "yyyy-MM-dd HH:mm:ss" format. While this approach standardizes the representation of dates, it can lead to discrepancies when the SQL query is intended to retrieve dates in a specific format. This mismatch between the expected format and the normalized format can cause the test to fail, not due to a logical error, but because of a formatting inconsistency.

In practice, SQL queries often require dates to be in specific formats that align with business rules or reporting requirements. Therefore, while our normalization step ensures consistency across different data sources, it may inadvertently introduce errors in scenarios where the SQL query demands a different date format. To mitigate this, future iterations of the testing framework could incorporate logic to detect and respect the intended date format, allowing for a more flexible comparison between LINQ and SQL results.

6.1.2 Handling Type Conversions

Another challenge lies in the handling of type conversions, particularly when dealing with primitive data types such as strings, integers, and decimals. For instance, a VARCHAR field in SQL might be interpreted as a string, but when scaffolded into the EF model, it could be converted into a different type such as an integer or decimal. This type conversion, although necessary for EF operations, can lead to inconsistencies when comparing LINQ results to SQL results.

In the current implementation, we attempt to address this issue by normalizing certain types of data, such as decimals, where the decimal point might be represented differently in LINQ results (e.g., a comma instead of a dot). We also account for potential null values in SQL results that might correspond to zero or false in LINQ results. These adjustments, while effective to some extent, are not comprehensive and may require further refinement to handle all edge cases.

6.1.3 Comparison of Results

The core of the EF testing process is the comparison of results obtained from the LINQ queries and their SQL counterparts. The comparison logic is designed to be robust, accounting for differences in the number of results, the structure of the returned data, and the types of values.

In scenarios where the LINQ query returns a single field or a single row, a direct comparison is made between the LINQ and SQL results. However, when dealing with more complex queries that return multiple fields or rows, the comparison becomes more intricate. The framework attempts to align the keys and values from both sets of results, ensuring that they match not only in value but also in type and format.

An additional layer of complexity is introduced when SQL results return an empty set or null values, which might correspond to non-null default values in LINQ results. The framework currently includes logic to handle these cases, such as converting null SQL results to default values in LINQ, but this approach may not cover all potential scenarios. Future enhancements could involve more sophisticated logic that considers the context of the data being compared, rather than relying on generic type-based conversions.

6.1.4 Caveats and Future Work

While the EF testing framework is effective in many cases, there are inherent caveats that need to be addressed in future work. The reliance on normalization and type-based comparisons can sometimes obscure underlying issues in the data or the queries themselves. Additionally, as the SQL and LINQ queries get more complex, the likelihood of encountering edge cases that are not adequately handled by the current framework also

increases.

Future improvements could focus on refining the normalization process, allowing for more context-sensitive handling of data formats, and enhancing the type conversion logic to better align with the nuances of different data types. Furthermore, expanding the comparison logic to include more sophisticated checks could have a significat impact on the accuracy and reliability of the testing framework.

Overall, while the EF testing framework has been effective for most LINQ queries, there's still room for improvement to ensure it can handle the full range of cases that might arise in a real-world Text2EF application.

6.2 SQL2EF

This section details the process of transforming SQL queries into Entity Framework (EF) expressions. This transformation involves several critical steps that ensure the accuracy and functionality of the generated EF code. Each subsection below addresses a specific aspect of this transformation process.

6.2.1 Alias and Identifier Handling

One of the core challenges in transforming SQL to EF is managing table aliases and identifiers. In SQL, table aliases are often used to simplify query writing, but in EF, it is essential to maintain a correct mapping between these aliases and the actual table names. We implement a `hashmap` to maintain this mapping, which is vital for ensuring that column references are correctly resolved during the transformation.

When column names are used without explicit table aliases in SQL, the query parser must infer the table from which the column originates. This becomes particularly challenging when multiple tables are involved in a query. Our approach automatically identifies the correct table for such columns by referencing the schema mapping for the tables involved.

Another critical aspect is case sensitivity. SQL is often case-insensitive, but EF is case-sensitive. Therefore, we normalize the case of all identifiers and aliases to ensure consistent behavior across the transformation. This normalization process also extends to handling scenarios where the same alias is used in different cases, such as `T1` and `t1`, ensuring that such ambiguities do not lead to incorrect EF expressions.

6.2.2 Schema Mapping

The process of schema mapping is essential for transforming SQL queries into EF expressions. This involves mapping the SQL schema to the corresponding EF schema by analyzing the `Context.cs` file, which contains metadata about the mappings between

SQL column names and EF fields. Additionally, we analyze the model files to extract further information about the columns, such as data types, optionality, and relationships.

The schema mapping process relies on a series of regular expressions to extract this data from the files. The extracted data is then organized in the order in which fields are defined, which is crucial for handling operations like `SELECT *`.

6.2.3 Projection Handling

Projection in SQL refers to the columns selected in a query, which we handle differently depending on the complexity of the query.

When the query involves a single function, such as `SELECT SUM(x)`, we identify specific cases like `COUNT(*)`, `COUNT(column)`, and `COUNT(DISTINCT column)`. These cases are transformed into the appropriate EF methods such as `.Count()`, `.Select(column).Count()`, and `.Select(column).Distinct().Count()` respectively. Similar transformations are applied for other aggregate functions like `SUM`, `AVG`, and others.

In cases where the query selects all columns using `SELECT *`, we generate an EF projection that selects each field explicitly in the correct order. For instance, this might look like `.Select(row => new { row.Field1, row.Field2, ... })`. This approach ensures that all columns are included in the result set, preserving the original query's intent.

For more complex queries that involve calculated fields, aggregated fields, or compound expressions, we maintain a record of these fields throughout the transformation process. This allows us to reuse them intelligently in other sections of the query, such as in `WHERE` or `GROUP BY` clauses.

Also, when multiple aggregated fields are present, we apply an additional `.GroupBy(row => 1)` to ensure proper aggregation across all records.

6.2.4 Filtering and Conditional Clauses

Filtering in SQL is typically achieved through `WHERE` and `HAVING` clauses. In EF, these must be translated into equivalent expressions while preserving the logical order of operations, especially when complex conditions are involved.

SQL allows for a variety of logical operations, such as `AND` and `OR`, that must be correctly parenthesized to ensure the correct order of evaluation. We handle these conditions recursively, ensuring that compound conditions like `A == 2 && (B >= 1 || C LIKE '%2')` are accurately reflected in the EF expressions.

6.2.5 Join Handling

Joining tables is one of the most complex aspects of SQL to EF transformation due to the variety of join syntaxes available in SQL. We focus on translating inner joins, which can be expressed in several ways in SQL.

In SQL, the order of join operations can affect the correctness of the query. For example, in a query like `X JOIN Y JOIN Z ON Z.a = Y.a AND X.b = Y.b`, we must reorder the joins so that each join has access to the necessary tables. This is achieved by analyzing the dependencies between tables and restructuring the joins accordingly.

Depending on the number of constraints in a join, we handle the transformation differently. For joins without constraints, we use EF's `SelectMany` syntax. For joins with a single constraint, we directly apply the condition in the EF join clause. When multiple constraints are present, we use the `new` syntax to create composite objects that represent the join conditions. For instance, `X JOIN Y ON X.a = Y.a AND X.b = Y.b` becomes `new { Pair1 = X.a, Pair2 = X.b }` and `new { Pair1 = Y.a, Pair2 = Y.b }` in EF.

To simplify the transformation process, we propagate all joined tables throughout the query, even though EF might allow for more optimized projections. This approach ensures that all necessary fields are available for subsequent operations.

6.2.6 Handling Keywords

SQL queries often include keywords like `ORDER BY` and `LIMIT`, which have direct counterparts in EF but require careful handling.

In SQL, multiple `ORDER BY` clauses can be used, which EF handles differently for the first clause and subsequent ones. The first clause is translated to `.OrderBy` or `.OrderByDescending`, while additional clauses are translated to `.ThenBy` or `.ThenByDescending`. When an `ORDER BY` clause refers to a calculated or aggregated field, we check whether the field has already been selected. If not, we compute it on-the-fly.

The `LIMIT` clause in SQL, which restricts the number of records returned, is directly translated into EF's `.Take(number)` method, being a very straightforward transformation.

6.2.7 Group By

The `GROUP BY` clause in SQL is used to group rows that share identical values in certain columns. In Entity Framework (EF), it's important to carefully track these grouped fields throughout the transformation process. This tracking enables us to reference the grouped fields in the projection step by using `.Key`.

Chapter 7

Conclusions

In this thesis, we have made significant strides in advancing the field of Text2EntityFramework (Text2EF) by creating a Rust-based application capable of transforming Text2SQL datasets into Text2EF datasets. Specifically, we successfully transformed 87% of the queries in the dataset, resulting in the first publicly available Text2EF dataset. This contribution is a crucial step forward in bridging the gap between NLP and software engineering, particularly in automating the generation of EF code from natural language queries.

7.1 Contributions and Implications

The primary contribution of this work lies in the development of a novel dataset and the initial exploration of the Text2EF task using SOTA language models. The successful transformation of the Spider dataset into a Text2EF dataset provides a valuable resource for the research community, enabling further exploration and experimentation in this emerging area. Moreover, our experiments using the Llama3.1-8B model, though preliminary, demonstrate the potential of LLMs in generating EF code directly from textual descriptions.

This work has important implications for the automation of software development tasks, particularly in the context of data-driven applications. By automating the generation of EF code, developers can interact with the databases with less effort and reduced time, leading to increased productivity and potentially higher software quality. Additionally, the Text2EF task opens new avenues for integrating NLP techniques into software engineering, thereby enhancing the capabilities of modern development tools.

7.2 Limitations

Despite these contributions, several limitations of the current work must be acknowledged. First, our experiments were confined to the Spider dataset, limiting the generalizability of

our findings. While Spider is a comprehensive and challenging Text2SQL dataset, there exist other datasets, such as WikiSQL, that could further diversify and enrich the Text2EF dataset. Expanding the dataset to include these additional sources would likely enhance the robustness of the Text2EF models and provide a more comprehensive evaluation.

Furthermore, our approach to text generation was limited to a single model, Llama3.1-8B, with minimal fine-tuning and prompt engineering. While this provided a useful starting point, it is evident that further experimentation with different models, hyperparameters, and prompt strategies could yield better performance. Techniques such as those proposed in SQLNet, where each part of the query is generated separately, could also be adapted to the Text2EF task, potentially providing an increase in the accuracy and coherence of the generated EF code.

In the SQL2EF task, the limitations of our C# testing framework also warrant attention. Specifically, the inability to determine whether incorrect query results were due to semantic errors or testing inaccuracies poses a challenge.

Moreover, the failure of 446 generated queries to build, particularly those involving many-to-many relationships, highlights an area where the current approach falls short. Addressing these issues will be essential for improving the overall reliability and accuracy of the SQL2EF pipeline.

7.3 Future Directions

Several potential directions for future research and development emerge from this work. First, expanding the Text2EF dataset by incorporating additional Text2SQL datasets, such as WikiSQL, would provide a richer and more diverse training ground for future models. This could lead to more generalized Text2EF solutions that perform well across different types of queries and databases.

Secondly, exploring different text generation models and techniques is a promising direction. Experimenting with various LLMs, fine-tuning strategies, and advanced prompt engineering methods could substantially improve the syntactic and semantic accuracy of the generated EF code. Additionally, adopting a modular approach to text generation, inspired by architectures like SQLNet, could further enhance the precision and reliability of the outputs.

Finally, addressing the limitations in the SQL2EF testing framework is crucial for advancing this line of research. Developing more sophisticated testing methods to distinguish between semantic errors and testing inaccuracies would provide clearer insights into model performance. Furthermore, enhancing the handling of complex SQL constructs, such as many-to-many relationships, will be vital for achieving more robust and reliable SQL2EF transformations.

7.4 Concluding Remarks

In summary, this thesis serves as a comprehensive representation of foundational effort in the exploration of the Text2EF task, contributing both a novel dataset and initial insights into the application of LLMs for this purpose. While the work is still in its early stages, the results obtained thus far are promising and suggest that with continued research and development, Text2EF has the potential to become a powerful tool for automating software development tasks. Future work in this area will undoubtedly build on the foundations laid here, leading to more sophisticated and accurate models, further narrowing the gap between natural language and software code.

Bibliography

[1] Abhimanyu Dubey et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need." In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[3] Xiaojun Xu, Chang Liu, and Dawn Song. *SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning*. 2017. arXiv: 1711.04436 [cs.CL]. URL: https://arxiv.org/abs/1711.04436.

[4] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. 2019. arXiv: 1809.08887 [cs.CL]. URL: https://arxiv.org/abs/1809.08887.