Proposal and Progress:

VQL Debugger with LLM, RAG, and Optimization Strategies

Sau Lai YIP

Abstract

Building on the study of databases and the preparation through literature review conducted in the previous UROP session, I have further studied related papers and proposed the framework for a VQL (Visualization Query Language) debugger, which primarily consists of a fine-tuned code-LLM (Large Language Model for Code) and a RAG (Retrieval-Augmented Generation) component, augmented by commonly practiced optimization strategies for LLMs. I have implemented the major component of this debugger by adapting a published pipeline de signed for SQL (Structured Query Language) error correction. Considering the syntax differences between VQL and SQL, I explored two adjustment methods, *VQL PyDict* and *SQL PyDict* + *VisBin PyDict*, and compared them through experiments. Based on the results, the upcoming plan is outlined.

1. Introduction

As data mining technologies develop and the popularity of deep-learning solutions increases, datasets are becoming increasingly complex and large-scale. Therefore, effective visualization is crucial for data analysis. As the language interface is a user-friendly paradigm, it can promisingly reduce the complexity for visualization tasks [1]. Although increasing efforts are dedicated to addressing the NL2Vis (Natural Language to Visualization) task, syntax and semantic errors remain unavoidable. Apart from improving NL2Vis solutions, developing error correction approaches is an alternative direction worth exploring.

One intermediate representation of NL2Vis is VQL (Visualization Query Language), which has a syntax highly similar to the widely used SQL (Structured Query Language). Given the extensive history and substantial efforts in text-to-SQL and SQL correction, it is effective and promising to address NL2Vis errors through VQL debugging, by adapting methods primarily used for SQL.

Most NL2Vis approaches are deep-learning-based [2], which usually effectively addresses syntax constraints but suffers from limited generalization across unseen databases or domains. Inspired by the remarkable generalizability of Large Language Models (LLMs) [2] and the initial addressing of syntax constraints by NL2Vis approaches, it is worthwhile to explore LLM-based debugging. This could potentially benefit from the complementary effects of deep-learning-based and LLM-based approaches.

2. Literature Review & Methodology

2.1. Overall Framework

Drawing on successful practices from previous works, I have proposed a framework for a VQL debugger using Code-LLM, RAG technique, and optimization strategies for LLMs. Based on their significance and source references, the framework is segmented into five components (see *Figure 1*): ***Corrector*** (green), ***Retriever*** (purple), ***Schema Encoder*** (orange), ***Detector*** (blue), and ***Action Planner*** (grey).

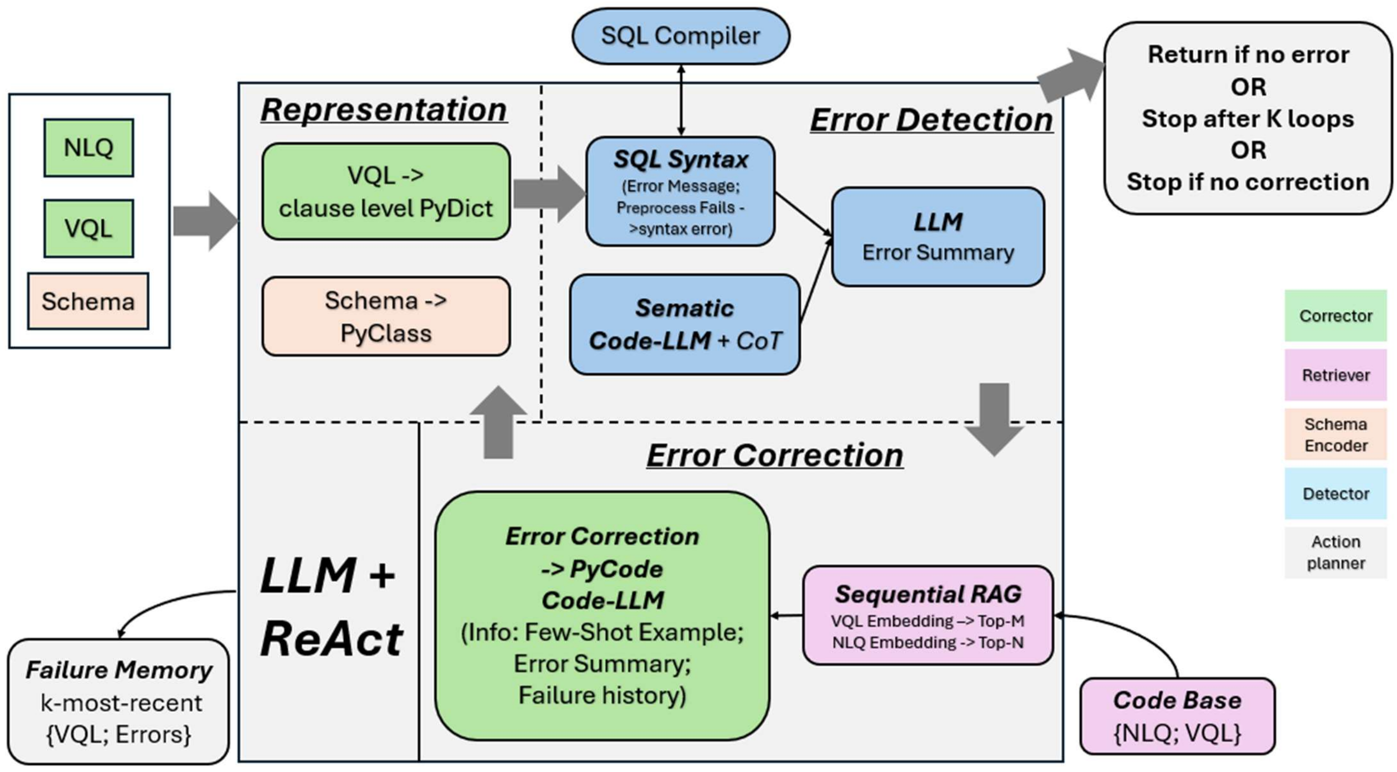
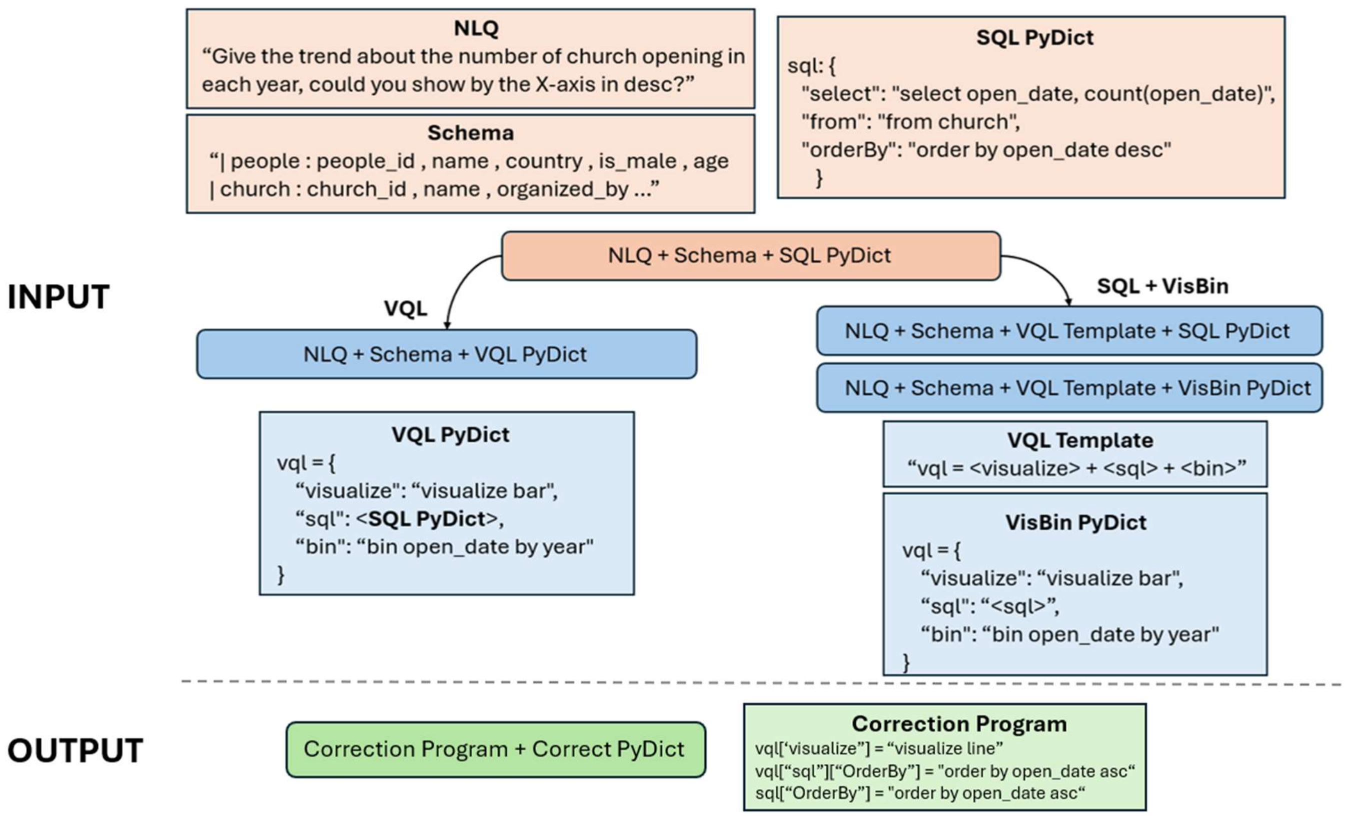


Figure 1. Overall framework of the VQL debugger.

2.2. Corrector


NLQ
"Give the trend about the number of church opening in each year, could you show by the X-axis in desc?"
Schema
"| people : people_id , name , country , is_male , age | church : church_id , name , organized_by ..."
SQL PyDict
INPUT
NLQ + Schema + SQL PyDict
VQL
SQL + VisBin
NLQ + Schema + VQL PyDict
NLQ + Schema + VQL Template + SQL PyDict
NLQ + Schema + VQL Template + VisBin PyDict
VQL PyDict
VQL Template
"vql = <visualize> + <sql> + <bin>"
VisBin PyDict
OUTPUT
Correction Program + Correct PyDict
Correction Program

```
sql: {
  "select": "select open_date, count(open_date)",
  "from": "from church",
  "orderBy": "order by open_date desc"
  }
```

```
vql = {
  "visualize": "visualize bar",
  "sql": <SQL PyDict>,
  "bin": "bin open_date by year"
}
```

```
vql = {
  "visualize": "visualize bar",
  "sql": "<sql>",
  "bin": "bin open_date by year"
}
```

```
vql['visualize"] = "visualize line"
vql["sql"]["OrderBy"] = "order by open_date asc"
sql["OrderBy"] = "order by open_date asc"
```

Figure 2. The two approaches to adjust the pipeline for SQL error correction for VQL. Oranges: PyDict representation method used in the original pipeline. Blue: adjusted version for VQL. Green: Python program representation in the output.

The Corrector is the fundamental component of the debugger, designed by adapting [3], an LLM-based pipeline for SQL corrections, considering the high similarity between SQL and VQL.

The original pipeline utilizes a clause-level representation for SQL using a *PyDict* (Python dictionary), where syntax keywords serve as keys, and the corresponding SQL query contents as values. To construct such a *PyDict* representation, the SQL is initially disambiguated and parsed into an abstract syntax tree (AST) using the Spider context-free grammar. The re-represented query, along with the corresponding natural language query (NLQ) and schema information, forms the input. The schema is depicted as the table name followed by the column names. The targeted edit in the output is structured as a *Python program*, aligning with the *PyDict* representation of the VQL. *Figure 2* illustrates this original representation approach in orange.

This method has proven more accurate thanks to the clause-level representation compared to previous token-level approaches [4]. Furthermore, the edit can be efficiently parsed through the direct execution of the *Python program* to update the dictionary. Given these advantages and performance, the *Corrector* component is adapted from this pipeline.

The primary differences between SQL and VQL are a visualization part at the beginning and an optional binning part at the end. Accordingly, two adjustment methods are proposed: (1) ***VQL PyDict***, which replaces SQL entirely with VQL in the *PyDict* representation to fine-tune one Code-LLM; (2) ***SQL PyDict + VisBin PyDict***, which constructs separate *PyDicts* for SQL and non-SQL parts ("visualize" and "bin"), fine-tunes two versions, and combines them as the *Corrector*.

(1) VQL PyDict

To differentiate the SQL part from the two non-SQL parts in the representation, as well as to effectively reuse the original implementation, *VQL PyDict* employs a key "sql" to correspond to the *SQL PyDict* in the original design and adds two additional key-value pairs for "visualize" and "bin" parts respectively (see blue parts on the left in Figure 2). This encapsulation method also simplifies the adjustment for edit representation, requiring only additional parsing to switch between "vql['sql']" and "sql" at the beginning of each program (refer to the differences between the last two lines in "*Correction Program*" in *Figure 2*).

(2) SQL PyDict + VisBin PyDict

Given that the encapsulation in method 1 may increase the complexity of the PyDict, potentially affecting Code-LLM's understanding of the representation, another approach, *SQL PyDict + VisBin PyDict*, is proposed to explore the effectiveness of separating the SQL and non-SQL parts. Two versions of Code-LLM are fine-tuned with two partial representations: one for the SQL part, using the identical SQL PyDict from the original design, and the other for the "visualize" and "bin" parts, replacing the actual value for the key "sql" in the *VQL PyDict* (introduced in method 1) with the string "<sql>". An additional *SQL Template* is included in the input to delineate the position of the three parts using strings "<visualize>", "<sql>", and "<bin>" respectively. This second adjustment method is depicted by the blue parts on the right side in *Figure 2*.

2.3. Retriever

The performance of LLMs is typically enhanced by utilizing in-context demonstrations [3]. Consequently, an RAG component is incorporated into the debugger to retrieve similar, correct VQL examples. The encoding approach for this retriever draws inspiration from [5], which retrieves VQL as templates and modifies them for NL2Vis. Their retriever encodes schema using a GNN [6] and NLQ using an LSTM in parallel. Given that incorrect VQL can partially reflect the structure of the correct VQL, it potentially offers more significant insights in terms of similarity compared to the NLQ. Therefore, a sequential architecture is employed in this VQL debugger. The VQL is encoded using a GNN that processes the corresponding AST. To streamline the training process, an off-the-shelf language model (e.g., BERT [7] and DeBERTa [8]) is utilized to encode the NLQ. Initially, M examples are retrieved based on structural similarity, as captured by the VQL encoder. These are then narrowed down to N examples based on semantic similarity in the NLQ. These examples are included in the input prompt for in-context learning.

2.4. Schema Encoder

As indicated by [2], the complexity of the schema (or table) representation significantly influences LLMs' performance in the NL2Vis task. This study evaluates various representation strategies, including table serialization, which is also the one used in [3], natural language summarization, markup formats (XML, Markdown, and CSV), and programming representations (SQL, Python). Their results demonstrate that programming approaches outperform other methods. Therefore, to enhance the original representation in [3], this work adopts the Python programming approach from [2], which also aligns with the Python representation of VQL and edits.

2.5. Detector & Action Planner

Besides in-context demonstrations, various optimization strategies are commonly employed in LLM-orchestrated architectures. [2] explores the effectiveness of several more strategies for the NL2Vis task, including CoT (Chain of Thought), which is applied by prompting LLM to first develop a sketch, as well as role-playing and code interpretation.

Another strategy, ReAct (Reasoning and Action Planning) [9], has been effectively implemented in *RTLFixer* [10], a debugger for Verilog code. This framework incorporates a feedback loop in which the LLM refers to execution messages from the compiler and the current status to determine necessary actions, including calling the compiler, returning answers, and performing RAG.

There is also an LLM-based SQL debugger for reference. The *SQLFixAgent* [11] is a looping SQL debugger composed of three LLM-based agent modules: *SQLRefiner*, *SQLReviewer*, and *QueryCrafter*. The *SQLReviewer* identifies errors through a step-by-step verification following the rubber duck debugging principle. This process makes resource use more efficient by ensuring that only SQL with detected errors progresses to later stages. The *QueryCrafter* generates candidate queries, which are then selected or revised by the *SQLRefiner*. The *SQLRefiner* also maintains a failure memory to track the latest records of unsuccessful corrections.

Building on the works mentioned, a *Detector* and *Action Planner* are incorporated to further enhance the debugger. The *Detector* interacts with a SQL compiler (e.g., SQLite3) to collect syntax errors, while a Code-LLM detects semantic errors through code interpretation and step-by-step cross-verification with the NLQ. The results from these steps are then summarized as error descriptions by an LLM, which are included in the input prompt to aid in identifying critical parts.

An *Action Planner* is also included to manage the procedure iteratively. It selects actions based on the results from error detection and compares new corrections with records in the failure memory. Actions may include returning correct VQL, reporting failure, or exiting due to step limitations. The *Action Planner* aims to reduce inference time by filtering correct VQL and to enhance accuracy through iterative corrections.

3. Experiments

The implementation of Corrector has now been completed. To compare the two methods of adjusting [3] for VQL, experiments were conducted using different amounts of data and backbone Code-LLMs.

3.1. Metrics

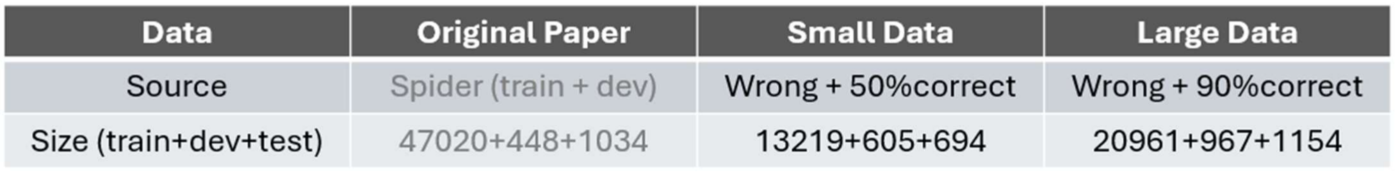
Each VQL is evaluated based on three components: (1) SQL, assessed by exact matching (EM) and execution matching (EX) as per the Spider dataset; (2&3) "visualize" and "bin", evaluated using exact matching. The overall VQL accuracy for EM and EX is calculated by integrating (2&3) into (1).

3.2. Dataset

[3] generates erroneous SQL from the Spider training set using 5-fold cross-validation. Due to the absence of an existing dataset for VQL error correction, the same method is employed using NVBench [12] as the source dataset and codeT5-small [13] as the NL2Vis parser. In each fold, the parser undergoes training for five epochs. VQL with failures in either SQL, "visualize," or "bin" parts are recorded as wrong generations. A total of 2461 erroneous VQLs are generated.

A certain percentage of correct data is reserved as a retrieval base for future use, while the remaining data is combined with the erroneous data. In [3], the test set is formed using the Spider development set. Since NVBench does not have an initial split, the test set is created from 5% of the combined data. Following [3], data from a set of pre-selected databases are collected and excluded correct ones to form the development set. Inputs and outputs are structured as outlined in Section 2. Particularly for the outputs, based on the practice in [3], incorrect VQLs are amended to match the gold standard, while correct ones are revised from scratch.

Two datasets are prepared with different reserved percentages for the retrieval base, with 50% for Small Data and 10% for Large Data (details are in *Table 1*).



Data	Original Paper	Small Data	Large Data
Source	Spider (train + dev)	Wrong + 50%correct	Wrong + 90%correct
Size (train+dev+test)	47020+448+1034	13219+605+694	20961+967+1154

Table 1. Statistics of datasets.

3.3. Settings

Two backbone Code-LLMs, codeT5-Small and CodeT5-Base [13], are utilized for the original pipeline and the two adjustment methods. The Code-LLM is trained over 10 epochs for each experiment, adhering to the original settings in [3].

3.4. Results

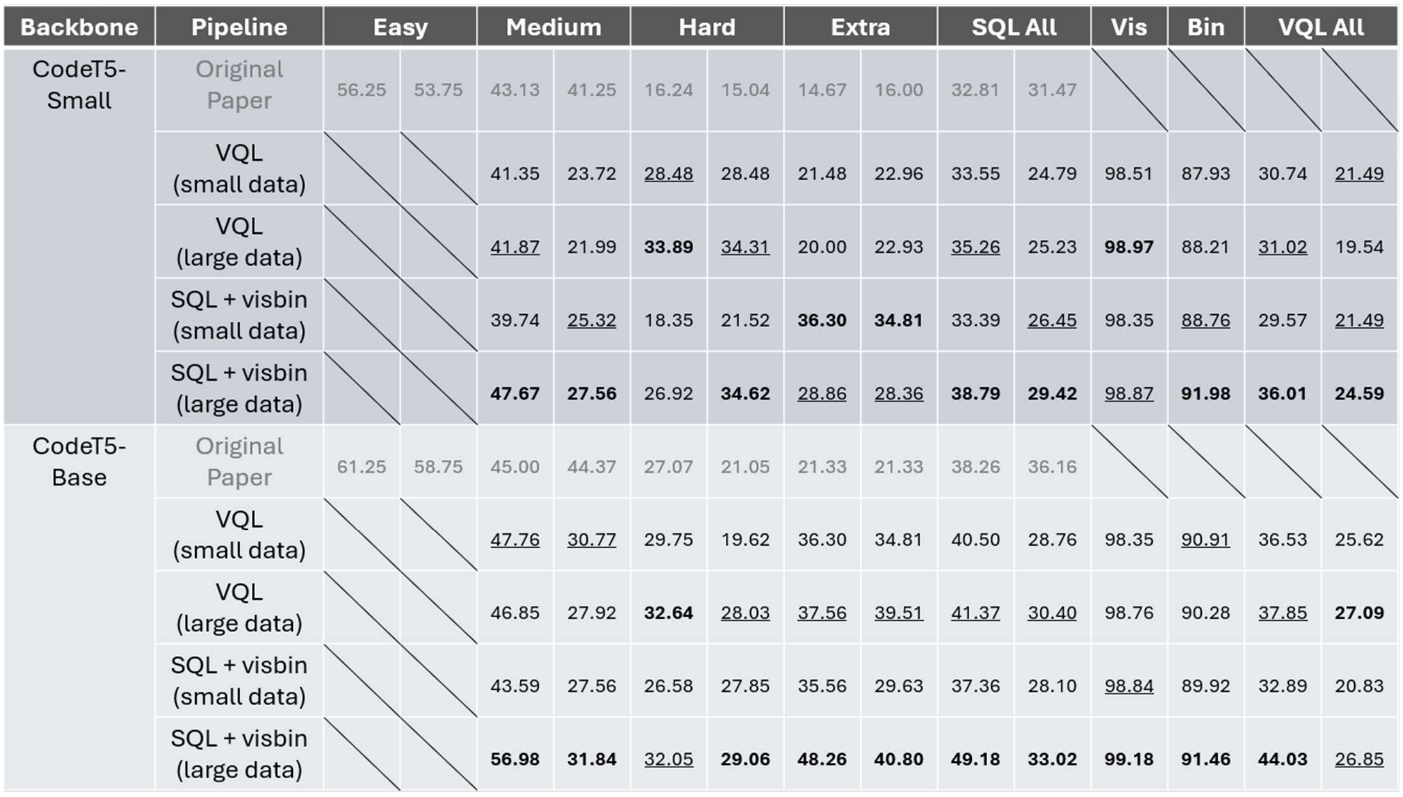
The results from the development set (see *Table 2*) and test set (see *Table 3*) are inconsistent, which may be due to the different construction approaches. The development set is constructed in an out-of-domain setting, as there is no training data from the same databases as those in the development sets, posting particular challenges in terms of generalizability.

The results for the development set (see *Table 2*) indicate that method 2 (*SQL + VisBin*) significantly outperforms method 1 (*VQL*). This suggests that the complexity introduced by encapsulation negatively impacts performance in an out-of-domain context. Additionally, this preference indicates that a simpler representation and problem-solving approach may enhance the generalizability of LLMs.

Conversely, method 1 exhibits superior performance in the test set. This could be because uniformity of representation holds more importance than simplicity in a within-domain context. Since method 2 only partially addresses VQL, this might lead to confusion concerning the unaddressed parts in NLQ.

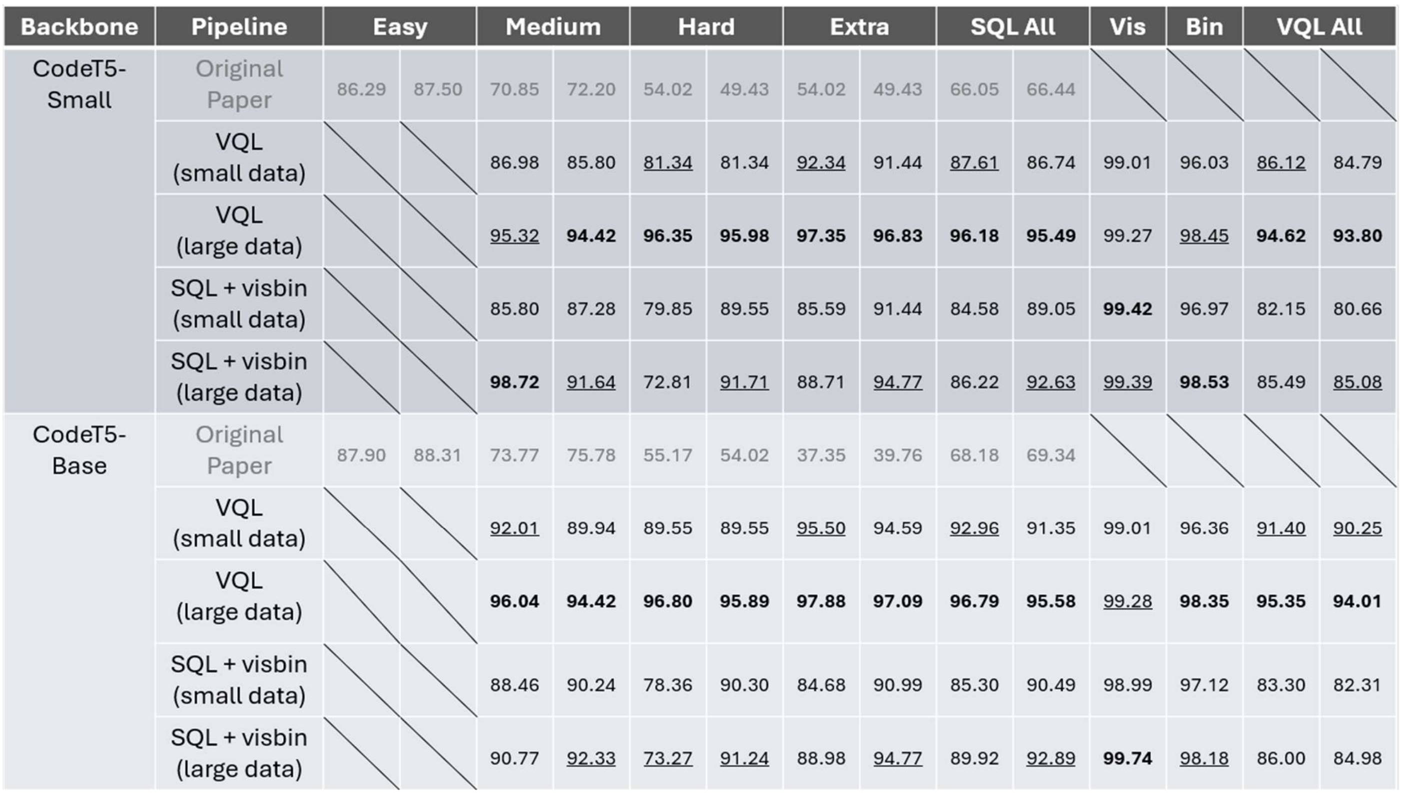
Given that real-world applications are more likely to be out-of-domain, method 2 is generally more favourable. Furthermore, it is observed that method 1 generates slightly more inexecutable edit programs, indicating that complexity in representation may compromise stability.

Results for both sets improve with the use of a larger dataset. Since the size of the retrieval base may correlate with the effectiveness of few-shot demonstrations, further experiments will be conducted to determine the optimal balance of proportions.



Backbone	Pipeline	Easy		Medium		Hard		Extra		SQL All		Vis	Bin	VQL All	
CodeT5-Small	Original Paper	56.25	53.75	43.13	41.25	16.24	15.04	14.67	16.00	32.81	31.47				
	VQL (small data)			41.35	23.72	28.48	28.48	21.48	22.96	33.55	24.79	98.51	87.93	30.74	21.49
	VQL (large data)			41.87	21.99	**33.89**	34.31	20.00	22.93	35.26	25.23	**98.97**	88.21	31.02	19.54
	SQL + visbin (small data)			39.74	25.32	18.35	21.52	**36.30**	**34.81**	33.39	26.45	98.35	88.76	29.57	21.49
	SQL + visbin (large data)			**47.67**	**27.56**	26.92	**34.62**	28.86	28.36	**38.79**	**29.42**	98.87	**91.98**	**36.01**	**24.59**
CodeT5-Base	Original Paper	61.25	58.75	45.00	44.37	27.07	21.05	21.33	21.33	38.26	36.16				
	VQL (small data)			47.76	30.77	29.75	19.62	36.30	34.81	40.50	28.76	98.35	90.91	36.53	25.62
	VQL (large data)			46.85	27.92	**32.64**	28.03	37.56	39.51	41.37	30.40	98.76	90.28	37.85	**27.09**
	SQL + visbin (small data)			43.59	27.56	26.58	27.85	35.56	29.63	37.36	28.10	98.84	89.92	32.89	20.83
	SQL + visbin (large data)			**56.98**	**31.84**	32.05	**29.06**	**48.26**	**40.80**	**49.18**	**33.02**	**99.18**	**91.46**	**44.03**	26.85

Table 2. Results for development sets. In each criterion for SQL, the left column represents EX, and the right column represents EM. The best performances are in bold, and the second best are underlined.



Backbone	Pipeline	Easy		Medium		Hard		Extra		SQL All		Vis	Bin	VQL All	
CodeT5-Small	Original Paper	86.29	87.50	70.85	72.20	54.02	49.43	54.02	49.43	66.05	66.44				
	VQL (small data)			86.98	85.80	81.34	81.34	92.34	91.44	87.61	86.74	99.01	96.03	86.12	84.79
	VQL (large data)			95.32	**94.42**	**96.35**	**95.98**	**97.35**	**96.83**	**96.18**	**95.49**	99.27	98.45	**94.62**	**93.80**
	SQL + visbin (small data)			85.80	87.28	79.85	89.55	85.59	91.44	84.58	89.05	**99.42**	96.97	82.15	80.66
	SQL + visbin (large data)			**98.72**	91.64	72.81	91.71	88.71	94.77	86.22	92.63	99.39	**98.53**	85.49	85.08
CodeT5-Base	Original Paper	87.90	88.31	73.77	75.78	55.17	54.02	37.35	39.76	68.18	69.34				
	VQL (small data)			92.01	89.94	89.55	89.55	95.50	94.59	92.96	91.35	99.01	96.36	91.40	90.25
	VQL (large data)			**96.04**	**94.42**	**96.80**	**95.89**	**97.88**	**97.09**	**96.79**	**95.58**	99.28	**98.35**	**95.35**	**94.01**
	SQL + visbin (small data)			88.46	90.24	78.36	90.30	84.68	90.99	85.30	90.49	98.99	97.12	83.30	82.31
	SQL + visbin (large data)			90.77	92.33	73.27	91.24	88.98	94.77	89.92	92.89	**99.74**	98.18	86.00	84.98

Table 3. Results for test sets. In each criterion for SQL, the left column represents EX, and the right column represents EM. The best performances are in bold, and the second best are underlined.

4. Upcoming Plan

4.1. Dataset and Parsers

Currently, errors are generated using CodeT5, the same language model (LLM) used for error correction. Concerns may arise that an LLM could be biased towards its own generation. To address this, the debugger should also be evaluated using errors from other NL2Vis parsers. One approach is to adapt sequence-to-sequence architectures for the NL2Vis task. Potential candidates include Seq2Vis [12], Transformer [14], and ncNet [1], which have been used as baselines in [5]. Alternatively, employing a different LLM for error generation and correction could be considered. Such evaluations would help assess the generalizability of the framework.

4.2. Component Development and Further Evaluation

Current experimental results suggest that the second adjustment approach is more favourable for better generalizability and stability. However, the first method should not be dismissed, as the integration of other components, which leverage in-context learning and optimization strategies for LLMs, may enhance generalizability and stability. It is possible that the negative impacts of representation complexity could be mitigated by incorporating these additional components.

5. Conclusion

This report initially examined the current state of the NL2Vis task and the importance of VQL error correction. Drawing on effective practices from previous works, an LLM-based VQL debugger consisting of a *Corrector*, *Retriever*, *Schema Encoder*, *Detector*, and *Action Planner* was proposed. The report then discussed the experiments and outcomes of the two implemented methods for adjusting [3] for VQL. Finally, the upcoming plan was outlined based on current progress and observations.

6. Acknowledgment

I extend my sincere gratitude to my supervisor, Prof. WONG, for his inspiring guidance and constructive feedback throughout this process. Special thanks to Yuanfeng SONG, the first author of [5], for generously sharing the implementation of RGVisNet and providing invaluable suggestions regarding the NL2Vis parsers.

References

[1] Y. Luo, N. Tang, G. Li, J. Tang, C. Chai, and X. Qin, “Natural Language to Visualization by Neural Machine Translation,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 1, pp. 217–226, Nov. 2021, doi: 10.1109/tvcg.2021.3114848.

[2] Y. Wu *et al.*, “Automated Data Visualization from Natural Language via Large Language Models: An Exploratory Study,” *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–28, May 2024, doi: 10.1145/3654992.

[3] Z. Chen *et al.*, “Text-to-SQL Error Correction with Language Models of Code,” *arXiv.org*, May 22, 2023. https://arxiv.org/abs/2305.13073

[4] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, “CoditT5: Pretraining for Source Code and Natural Language Editing,” *arXiv.org*, Aug. 10, 2022. https://arxiv.org/abs/2208.05446

[5] Y. Song, X. Zhao, R. C. Wong, and D. Jiang. RGVisNet: A Hybrid Retrieval-Generation Neural Framework Towards Automatic Data Visualization Generation. *In Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22)*. Association for Computing Machinery, New York, NY, USA, 1646–1655. 2022. https://doi.org/10.1145/3534678.3539330.

[6] F. Scarselli, M. Gori, A. c. Tsoi, M. Hagenbuchner, and G. Monfardini. 2008. The graph neural network model. *TNN* (2008).

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” arXiv.org, Oct. 11, 2018. https://arxiv.org/abs/1810.04805

[8] P. He, X. Liu, J. Gao, and W. Chen, “DeBERTa: Decoding-enhanced BERT with Disentangled Attention,” arXiv.org, Jun. 05, 2020. https://arxiv.org/abs/2006.03654

[9] S. Yao et al. 2022. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*.

[10] Y. Tsai, M. Liu, H. Ren. RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models. https://arxiv.org/abs/2311.16543v3, 2023. doi:10.48550/arXiv.2311.16543.

[11] J. Cen, J. Liu, Z. Li, J. Wang. SQLFixAgent: Towards Semantic-Accurate Text-to-SQL Parsing via Consistency-Enhanced Multi-Agent Collaboration. https://arxiv.org/abs/2406.13408v2, 2024. doi:10.48550/arXiv.2406.13408.

[12] Y. Luo, J. Tang, and G. Li, “nvBench: A Large-Scale Synthesized Dataset for Cross-Domain Natural Language to Visualization Task,” *arXiv.org*, Dec. 24, 2021. https://arxiv.org/abs/2112.12926

[13] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation,” *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Jan. 2021, doi: 10.18653/v1/2021.emnlp-main.685.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. 2017. Attention is all you need. In *NIPS*.