Proposal and Progress:

VQL Debugger with Code-LLM and Optimization Strategies

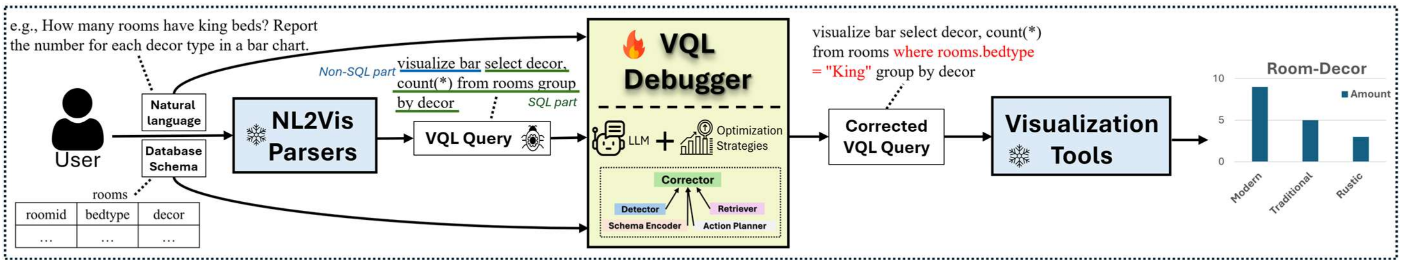
Abstract

Efficient data analysis and interpretation are essential in today's information era. NL2Vis (Natural Language to Visualization) technologies offer accessible interfaces to utilize visualization tools by generating visualization queries accepted to such tools from natural language queries. Despite their potential, existing NL2Vis parsers often encounter errors due to the inherent gap between natural language and visualization queries. In addition to advancing NL2Vis systems, exploring debugging methods to correct NL2Vis-generated queries offers a promising solution. Therefore, we introduce a VQL (Visualization Query Language) debugger, considering that VQL is a query language commonly used as the input format for visualization tools and the output format for NL2Vis parsers. Building on existing research, our VQL debugger employs pre-trained Code-LLMs (Large Language Models for Code) combined with common optimization strategies for deploying LLMs.

Our VQL debugger comprises five components. The *Corrector* is primarily responsible for debugging, aided by enriched database schema information from the *Schema Encoder*, syntax and semantic errors identified by the *Detector*, relevant examples retrieved by the *Retriever* from a codebase, and the correction history recorded by the *Action Planner* during an iterative debugging process. The architecture of each component has been designed by referring to previous successes in related work.

The *Corrector* has been developed by adapting a SQL (Structured Query Language) correction pipeline, leveraging the syntax similarities between VQL and SQL. A VQL query includes a SQL query, which can extract data from databases with specified constraints, along with non-SQL components that specify visualization attributes such as diagram types. Based on the differences, two adaptation methods have been developed: one treating a VQL as a whole and the other treating the SQL and non-SQL parts separately. These methods have been compared through experiments and analysis from various perspectives. Based on this initial study, a plan and focus for the implementation and evaluation of the other components have been specified for further work.

In summary, our contributions include: (1) introducing a VQL debugger; (2) examining the adaptability of effective practices for other programming languages to VQL; (3) exploring how to optimize pre-trained LLMs (Large Language Models) for VQL debugging. If successfully developed with satisfactory performance, our debugger could improve companies' efficiency in data interpretation and productivity by: (1) shortening professionals' drafting time of visualization queries; (2) lowering or even eliminating programming requirements for non-professionals to generate and refine visualizations via a simple natural language interface.



1. Introduction

In today's information era, efficiently analyzing and interpreting complex datasets is increasingly crucial. NL2Vis (Natural Language to Visualization) technologies offer accessible interfaces to utilize visualization tools by generating visualization queries accepted to such tools from natural language queries. Despite its promise, existing NL2Vis parsers are susceptible to errors due to the inherent gap between natural language and structured visualization queries. Beyond advancing NL2Vis systems, exploring debugging methods to correct NL2Vis-generated visualization queries presents another promising avenue.

VQL (Visualization Query Language) is one of the structured query languages commonly used by visualization tools to generate diagrams and often serves as an output format for NL2Vis parsers. A VQL query comprises a SQL (Structured Query Language) query, which is used to extract data from databases with certain constraints, along with two non-SQL components (the visualization and binning parts) that specify visualization attributes. Given SQL's widespread use and study in areas such as text-to-SQL (i.e., generating SQL from natural language) and SQL correction, along with the minor differences between VQL and SQL, exploring VQL debugging is promising. Building on existing research, we present a VQL debugger that uses pre-trained Code-LLMs (Large Language Models for Code) with common optimization strategies for deploying LLMs.

Our contributions include: (1) introducing a VQL debugger; (2) examining the adaptability of correction approaches from other programming languages to VQL; and (3) exploring how to optimize LLMs for VQL debugging tasks. If successfully developed with satisfactory performance, our debugger could improve companies' efficiency in data interpretation and productivity by: (1) shortening professionals' drafting time of visualization queries; (2) lowering or even eliminating programming requirements for non-professionals to generate and refine visualizations via a simple natural language interface.

2. Methodology

2.1. Overall Framework

Building on successful practices from previous works, I propose a framework for a VQL debugger using Code-LLM and optimization strategies for LLMs, including prompt engineering, RAG (retrieved-augmented generation), CoT (chain-of-thought), and ReAct (Reasoning and Action Planning). The framework is divided into five components based on their significance and source references (see Figure 1): ***Corrector***, ***Retriever***, ***Schema Encoder***, ***Detector***, and ***Action Planner***.

2.2. *Corrector*

The *Corrector* is the core component of the debugger. Thanks to the high similarity between SQL and VQL, this component is adapted from an LLM-based pipeline for SQL corrections [3]. The original pipeline uses a clause-level representation for SQL with PyDict (Python dictionary), where syntax keywords are keys and the corresponding clause contents are values. To effectively obtain the PyDict representation, a SQL is initially parsed into an abstract syntax tree (AST) using the Spider context-free grammar. The re-represented SQL, natural language query (NLQ), and schema, which are depicted as the table name followed by column names, form the input prompt. The edit in the output is structured as Python programs, aligning with the PyDict representation. Figure 2 illustrates this original representation approach in orange.

As justified in [3], this clause-level representation is more accurate than previous token-level approaches [4]. Moreover, the edit can be efficiently parsed by directly executing the Python program to update the PyDict. Given these advantages, the *Corrector* component is developed by adapting this pipeline.

The main differences between SQL and VQL are a visualization part at the beginning and an optional binning part at the end. Two adjustment methods are proposed:

- ***VQL PyDict***, which represents a VQL with one PyDict and fine-tunes one Code-LLM.
- ***SQL PyDict + VisBin PyDict***, which constructs separate PyDicts for SQL and non-SQL parts ("visualize" and "bin" parts), fine-tunes two versions of Code-LLM and combines them as the *Corrector*.

2.2.1. VQL PyDict

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

2.2.2. ***SQL PyDict + VisBin PyDict***

Since the encapsulation in the previous method may complicate the PyDict, affecting Code-LLM's understanding, another approach, *SQL PyDict + VisBin PyDict*, explores the separation of SQL and non-SQL parts. Two versions of Code-LLM are fine-tuned with partial representations: one for SQL, using the original *SQL PyDict*, and another for "visualize" and "bin" parts, replacing the "sql" value in the *VQL PyDict* (method 1) with the string "<sql>". An additional VQL template in the input delineates the three parts using "<visualize>", "<sql>", and "<bin>". This second method is depicted by the blue parts on the right side in Figure 2.

2.3. ***Retriever***

LLM performance is often improved with in-context demonstrations [3]. Thus, an RAG component is added to the debugger to retrieve similar VQL examples. The *Retriever*'s encoding method is inspired by RGVisNet [5], which retrieves VQL as templates and adapts them for NL2Vis. It uses a GNN [6] for schema encoding and an LSTM for NLQ encoding in parallel. Since the incorrect VQL may partially reflect the correct structure, providing better similarity insights than NLQ, this VQL debugger uses a sequential architecture. A GNN encodes the VQL by processing its AST. To simplify training, an off-the-shelf language model (e.g., BERT [7], DeBERTa [8]) encodes the NLQ. Initially, M examples are retrieved based on structural similarity from the VQL encoder and narrowed to N examples based on NLQ semantic similarity. These examples are included in the input prompt for in-context learning.

There are two strategies for providing examples. Each retrieved data point can present with (1) the **gold VQL**, or (2) the **edit programs** from a wrong prediction to the gold VQL. The first one is more token economic while the second one is more consistent with the debugging task. Their effects will be compared through experiments.

2.4. *Schema Encoder*

According to [2], schema complexity significantly affects LLM performance in NL2Vis tasks. This study evaluates various representation strategies, including table serialization (used in [3]), natural language summarization, markup formats (XML, Markdown, CSV), and programming representations (SQL, Python). Results show programming approaches perform best. Thus, to improve the original representation in [3], this work adopts the Python programming approach from [2] to enrich the schema representation, aligning with the Python representation of VQL and edits.

2.5. *Detector & Action Planner*

In addition to providing guidance in the input prompt, there are optimization strategies to facilitate the reasoning process in LLM-orchestrated architectures. [2] investigates several strategies for NL2Vis tasks, such as CoT (Chain of Thought) that prompts LLM to sketch first, role-playing, and code interpretation. Another strategy, ReAct (Reasoning and Action Planning) [9], is effectively used in RTLFixer [10], a Verilog code debugger. This framework uses a feedback loop where the LLM refers to compiler execution messages and current status to select appropriate actions from calling the compiler, returning answers, and performing RAG.

An LLM-based SQL debugger, SQLFixAgent [11], serves as a reference for these two components. It is a looping SQL debugger with three LLM-based agent modules: SQLRefiner, SQLReviewer, and QueryCrafter. SQLReviewer identifies errors through step-by-step verification following the rubber duck debugging principle, ensuring only SQL with detected errors progresses further. QueryCrafter generates candidate queries, selected or revised by SQLRefiner, which also maintains a failure memory to track unsuccessful corrections.

Building on these works, a *Detector* and *Action Planner* are added to enhance the debugger. The *Detector* communicates 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 analysis. An LLM summarizes these results as error descriptions, which are included in the input prompt to assist in identifying critical parts. The *Action Planner* manages the process iteratively, selecting actions based on error detection results and comparing new corrections with failure memory records. Actions may include returning correct VQL, reporting failure, or exiting due to step limits. The *Action Planner* aims to reduce inference time by filtering correct VQL and improve accuracy through iterative corrections.

3. Results

3.1. VQL Debugging Dataset

The VQL debugging data is derived from nvBench [12] using 5-fold cross-validation as outlined in [3]. NvBench is divided into 5 equal sets. An NL2Vis parser (CodeT5-Small) is trained for 1 epoch on 4 sets and evaluated on 1 set. For each inference, the top 10 responses are sampled, and erroneous ones are collected to form the VQL debugging dataset. This process is repeated five times, with a different set as the dev set each time. A total of 171,837 erroneous VQL queries are synthesized from 25,386 NL2Vis data points.

The VQL debugging dataset is split into training, development, and test sets in two ways:

- **Out-of-distribution (denoted as *outof*) setting**: Datapoints are split by databases following [3]. For each gold VQL in the dev and test sets, the most confident erroneous VQL is used. If it is not processable or does not exist, an empty VQL is used. In the training set, all processable errors are included, defaulting to an empty VQL if none exist for a gold VQL. To match [3]'s data volume, 50% are randomly selected from each set, resulting in: training (47,003), dev (558), and test (1,183).
- **Within-distribution (denoted as *within*) setting**: Data points are randomly divided into the same portion (i.e. 47,003 for training, 558 for dev, and 1,183 for test). All processable errors are included, with an empty VQL used if none exist.

3.2. Metrics

A VQL is considered as three parts:

- **SQL**, evaluated by EX (execution matching) and EM (exact matching) following Spider.
- **Vis** (the "visualize" part), evaluated by EM.
- **Bin** (the "bin" part), evaluated by EM.

The two metrics for VQL are:

- **VQL EX** = SQL EX && Vis EM && Bin EM.
- **VQL EM =** SQL EM && Vis EM && Bin EM.

3.3. Settings

Two Code-LLM backbones, CodeT5-Small and CodeT5-Base [13], are employed. Each is trained over 10 epochs, with other hyperparameters following the original settings in [3].

3.4. Experiments on *Corrector*

The two methods, *VQL PyDict* and *SQL PyDict + VisBin PyDict*, were evaluated in both *outof* and *within* settings. The result is shown in Tables 1&2. Overall, *VQL PyDict* emerged as the preferred method.

In the *outof* setting, *VQL PyDict* outperformed *SQL PyDict + VisBin PyDict*, despite a minor discrepancy. Specifically, *VQL PyDict* surpassed the latter by 2.11% in EX and 3.38% in EM on average. Conversely, in the within setting, *VQL PyDict* slightly underperformed but remained comparable to *SQL PyDict + VisBin PyDict*, with only a marginal deficit of 0.85% in EX and 0.22% in EM on average. It is important to note that the *outof* setting is significantly more challenging than the *within* setting. This is evidenced by the substantial gap in average performance, with scores in the *outof* setting mostly below 50%, while those in the *within* setting approach 100%. Given that real-world scenarios are more likely to resemble the *outof* setting, *VQL PyDict* is preferable in terms of **performance**.

Additional factors differentiate the two methods. One is **unity/consistency**. *VQL PyDict* aligns more closely with the NLQ, whereas *SQL PyDict* and *VisBin PyDict* only partially correspond to it. The advantage of *SQL PyDict + VisBin PyDict* lies in its **simplicity**, as it is structurally less complex. The results indicate different preferences for these features across settings: unity/consistency is more critical in the *outof* setting, while simplicity is favored in the *within* setting. Another benefit of *VQL PyDict* is its **economy**, as it requires only one model to be trained, compared to the two models needed for *SQL PyDict + VisBin PyDict*. This adds another layer of support for choosing the VQL PyDict method.

These analytical insights suggest potential improvements for the *SQL PyDict + VisBin PyDict* method. For instance, it might be advantageous to operate on the two PyDicts simultaneously rather than through independent pipelines, or to employ a simpler approach for handling the VisBin part, given that it is less complex.

4. Future Work

4.1. Dataset and Parsers

Currently, errors are generated using CodeT5, the same language model (LLM) employed for error correction. This raises concerns about potential bias towards its own outputs. To mitigate this, one strategy is to synthesize errors from other NL2Vis parsers, such as sequence-to-sequence architectures like Seq2Vis [12], Transformer [14], and ncNet [1], which are used as baselines in [5]. Alternatively, using a different Code-LLM (e.g. Qwen2.5-Coder [15], Code Llama [16]) for error generation and correction could be explored. These evaluations will also help determine the framework's generalizability.

4.2. Other Components

The remaining components (*Retriever*, *Corrector*, *Schema Encoder*, and *Action Planner*) will be developed, and an ablation study will be conducted to evaluate each of them. Given the limited potential for improvement under the *within* setting, the focus will be on the *outof* setting. The experiments will examine the impact of balancing detail and content length on performance when introducing these components, as additional information can provide valuable guidance but also increase content length.

In particular, two additional issues need to be studied for the *Retriever*:

- **Providing Gold VQLs vs. Providing Edit Programs**: Experiments will be conducted to compare these two strategies, which have been discussed previously.
- **Fine-Tuning vs. In-Context Learning**: Some other Code-LLMs, such as Qwen2.5-Coder and Code Llama, are larger than Code-T5 (with CodeT5-Small containing 60M parameters and CodeT5-Base containing 220M), which are expected to be more powerful. Therefore, it is worthwhile to investigate whether few-shot in-context learning is comparable to fine-tuning for such models, which may provide a more economical approach. Notably, when using in-context learning alone, few-shot examples will be in the form of edit programs.

5. Acknowledgement

I extend my sincere gratitude to my supervisor, Prof. Raymond WONG, for his patient guidance.

6. 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*.

[15] B. Hui *et al.*, "QWen2.5-Coder Technical Report," *arXiv.org*, Sep. 18, 2024. https://arxiv.org/abs/2409.12186

[16] B. Rozière *et al.*, "Code llama: Open Foundation Models for code," *arXiv.org*, Aug. 24, 2023. https://arxiv.org/abs/2308.12950

Appendix

Figures/ Tables

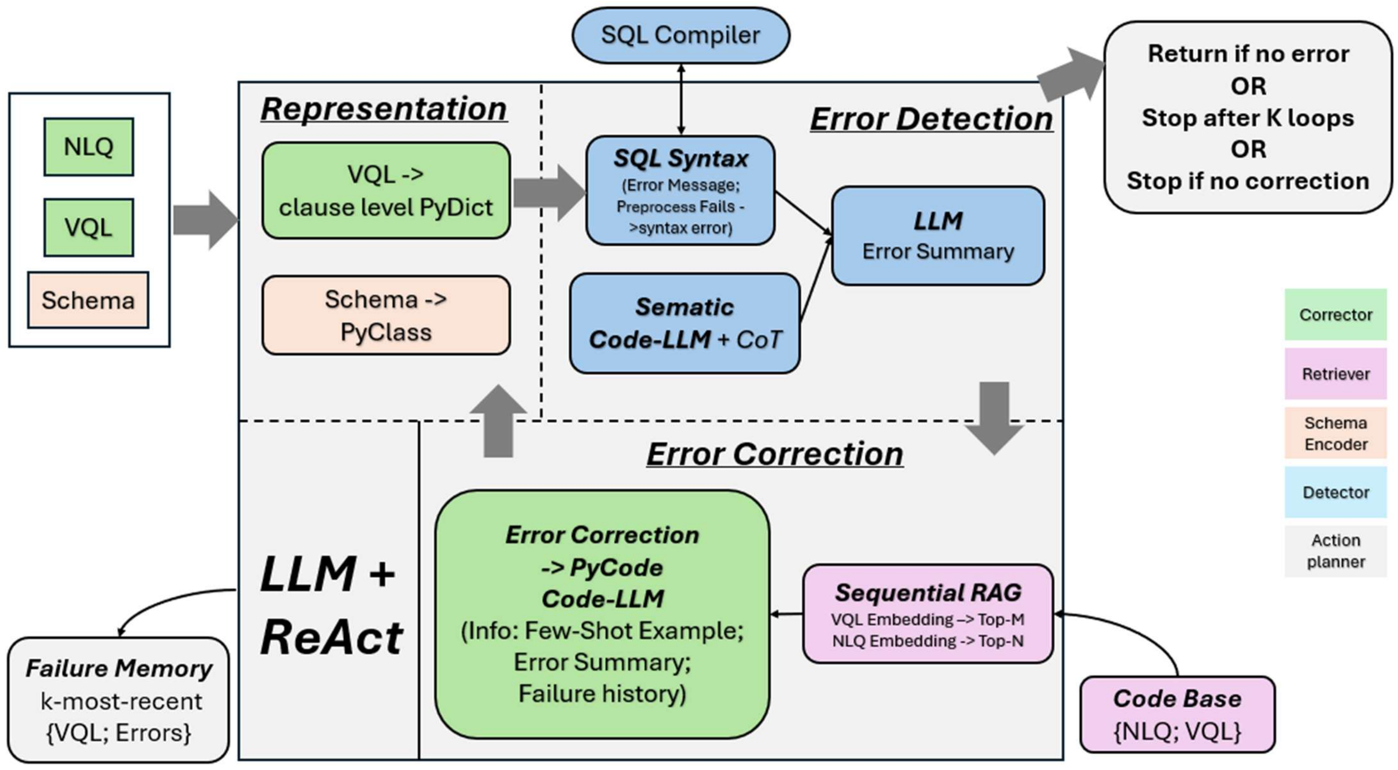
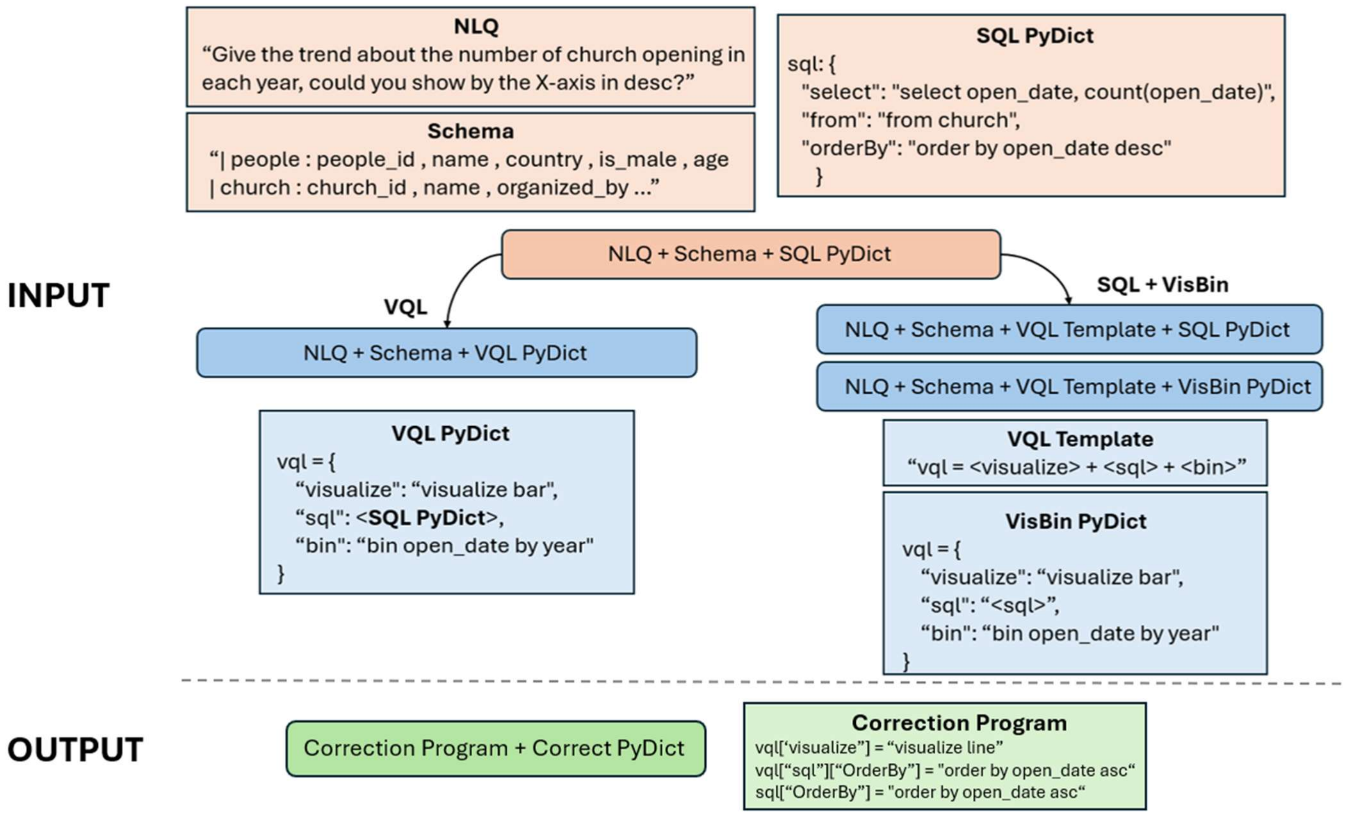

NLQ
VQL
Schema
Representation
VQL -> clause level PyDict
Schema -> PyClass
SQL Compiler
Error Detection
SQL Syntax (Error Message; Preprocess Fails -> syntax error)
Sematic Code-LLM + CoT
LLM Error Summary
Return if no error OR Stop after K loops OR Stop if no correction
Corrector
Retriever
Schema Encoder
Detector
Action planner
Error Correction
LLM + ReAct
Error Correction -> PyCode Code-LLM (Info: Few-Shot Example; Error Summary; Failure history)
Sequential RAG
VQL Embedding -> Top-M
NLQ Embedding -> Top-N
Code Base {NLQ; VQL}
Failure Memory k-most-recent {VQL; Errors}

Figure 1. The overall framework of the VQL debugger.


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: components used in the PyDict representation method of the original pipeline. Blue: adjusted version for VQL. Green: Python program representation in the output.

Outof			**VQL Eval** (EX|EM)								
Set	**Backbone**	**Pipeline**	**Easy**	**Medium**		**Hard**		**Extra**		**All**	
Dev			(0)	(307)		(141)		(110)		(558)	
	CodeT5-Small	**VQL**		**33.88**	**18.89**	**23.40**	**23.40**	**31.82**	**29.09**	**30.82**	**22.04**
		SQL+VisBin		31.92	13.68	17.73	15.60	30.00	26.36	27.96	16.67
	CodeT5-Base	**VQL**		**37.13**	**17.92**	**24.82**	**21.99**	**39.09**	**35.45**	**34.41**	**22.40**
		SQL+VisBin		35.83	14.98	24.11	20.57	37.27	31.82	33.15	19.17
Test			(0)	(755)		(161)		(267)		(1183)	
	CodeT5-Small	**VQL**		41.72	39.74	14.91	11.80	38.58	**38.58**	37.36	35.67
		SQL+VisBin		**43.18**	**41.59**	**16.77**	**14.91**	**39.70**	**38.58**	**38.80**	**37.28**
	CodeT5-Base	**VQL**		**55.23**	**55.23**	17.39	13.66	**56.55**	**59.93**	**50.38**	**49.28**
		SQL+VisBin		49.14	48.08	**19.25**	**14.29**	47.19	44.94	44.63	42.77

Within			**VQL Eval** (EX|EM)								
Set	**Backbone**	**Pipeline**	**Easy**	**Medium**		**Hard**		**Extra**		**All**	
Dev			(0)	(302)		(92)		(164)		(558)	
	CodeT5-Small	**VQL**		94.37	94.37	90.22	91.30	**96.95**	**96.95**	94.44	**94.62**
		SQL+VisBin		**95.03**	**94.70**	**92.39**	**93.48**	95.73	95.12	**94.80**	**94.62**
	CodeT5-Base	**VQL**		96.03	97.02	90.22	91.30	**98.17**	96.95	95.70	96.06
		SQL+VisBin		**97.68**	**97.68**	**94.57**	**94.57**	**98.17**	**97.56**	**97.31**	**97.13**
Test			(0)	(609)		(245)		(329)		(1183)	
	CodeT5-Small	**VQL**		93.27	**93.92**	89.39	**89.80**	**97.26**	**96.96**	93.58	**93.91**
		SQL+VisBin		**93.43**	92.45	**90.61**	**89.80**	96.35	96.05	**93.66**	92.90
	CodeT5-Base	**VQL**		94.25	94.58	**96.61**	91.02	96.66	96.66	94.17	94.42
		SQL+VisBin		**95.57**	**95.24**	93.47	**93.06**	**96.96**	**96.96**	**95.52**	**95.27**

Tables 1&2. Results for the out-of-distribution and within-distribution setting. For each comparison of the two methods (i.e. VQL PyDict and SQL PyDict + VisBin PyDict), the higher score is bold. The statistics in braces show the exact number of data points belonging to the hardness level. The hardness is determined based on the SQL part following the Spider evaluation used by the referred pipeline.