RetrySQL: Text-to-SQL Training with Retry Data for Self-Correcting Query Generation

Alicja Rączkowska*, **Riccardo Belluzzo***, **Piotr Zieliński***, **Joanna Baran***, **Paweł Olszewski***

Machine Learning Research
Allegro.com
{alicja.raczkowska, riccardo.belluzzo, piotr.c.zielinski, joanna.baran, pawel.olszewski}@allegro.com

Abstract

The text-to-SQL task is an active challenge in Natural Language Processing. Many existing solutions focus on using black-box language models extended with specialized components within customized end-to-end text-to-SQL pipelines. While these solutions use both closed-source proprietary language models and coding-oriented open-source models, there is a lack of research regarding SQL-specific small generative models. At the same time, recent advancements in self-correcting generation strategies show promise for improving the capabilities of existing architectures. The application of these concepts to the text-to-SQL task remains unexplored. In this paper, we introduce *RetrySQL*, a new approach to training text-to-SQL generation models. We prepare reasoning steps for reference SQL queries and then corrupt them to create *retry data* that contains both incorrect and corrected steps, divided with a special token. We continuously pre-train open-source coding models with this data and demonstrate that retry steps yield an improvements of up to 4 and 9 percentage points for overall and challenging execution metrics, respectively, as compared to pre-training without *retry data*. We showcase that the self-correcting behavior is learned by the model and the increase in downstream accuracy metrics is a result of this additional skill. Finally, we incorporate *RetrySQL*-trained models into the full text-to-SQL pipeline and showcase that they are competitive in terms of Execution Accuracy with proprietary models that contain orders of magnitude more parameters. *RetrySQL* demonstrates that self-correction can be learned in the text-to-SQL task and provides a novel way of improving generation accuracy for small SQL-oriented language models.

Code — https://github.com/allegro/RetrySQL
Extended version — https://arxiv.org/abs/2507.02529

1 Introduction

The task of translating natural language questions to SQL queries is a major challenge for machine learning models. The complexity stems from the need of relating often ambiguous user input to abstract entities, relations and values that are present in relational databases (Li et al. 2023). Even prominent Large Language Models (LLMs), such as GPT-4o (Team 2024b) or Gemini 1.5 (Team 2024a), struggle with approaching human performance in leading text-to-SQL benchmarks: BIRD (Li et al. 2023) and SPIDER 2.0 (Lei et al. 2024). The same is true for models tuned specifically for coding tasks (Li et al. 2024; Chen et al. 2024; Talaei et al. 2024). Thus, there exists a need for more advanced solutions that can bridge that gap and provide reliable SQL queries even in difficult real-world scenarios.

The text-to-SQL task can be divided into three main steps (Maamari et al. 2024): retrieval, generation and correction. Many existing approaches try to tackle these steps at the same time, in a single end-to-end pipeline (Maamari et al. 2024; Talaei et al. 2024; Pourreza and Rafiei 2023, 2024), utilizing relatively large LLMs. In this work, we focus only on the generation step and show how it can be improved with a novel approach to model pre-training of small LLMs - text-to-SQL systems need to operate quickly in real-world scenarios.

Specifically, we teach the model to self-correct during the generation itself. While previous work did use self-correction in the sense of post-processing, applied in the correction step (Pourreza and Rafiei 2023), we enforce the self-correcting behavior at an earlier point. This sort of active knowledge-based self-correction is an ongoing research area when it comes to LLMs (Ye et al. 2024a,b). While recent work in slow thinking reasoning systems, such as DeepSeek-R1 (Team 2025), shows that self-correction can be learned in a reinforcement learning setup, other lines of research suggest that it is possible to obtain the self-correction ability with specific data augmentations and a standard autoregressive pre-training objective (Ye et al. 2024a,b) or during supervised fine-tuning (Muennighoff et al. 2025). It has been shown that augmenting training data for grade-school math solution generation with so-called *retry data* leads to increased generation accuracy (Ye et al. 2024b). The applicability of this approach to other tasks and models has not been explored as of yet.

We introduce *RetrySQL*, a novel text-to-SQL generation module training paradigm that incorporates *retry data* in the training process and teaches the resulting model to self-correct. *RetrySQL* first augments the training data with reasoning steps that explain the sequence of operations required for obtaining the solution SQL query (**Fig. 1a**). Then,

*These authors contributed equally.

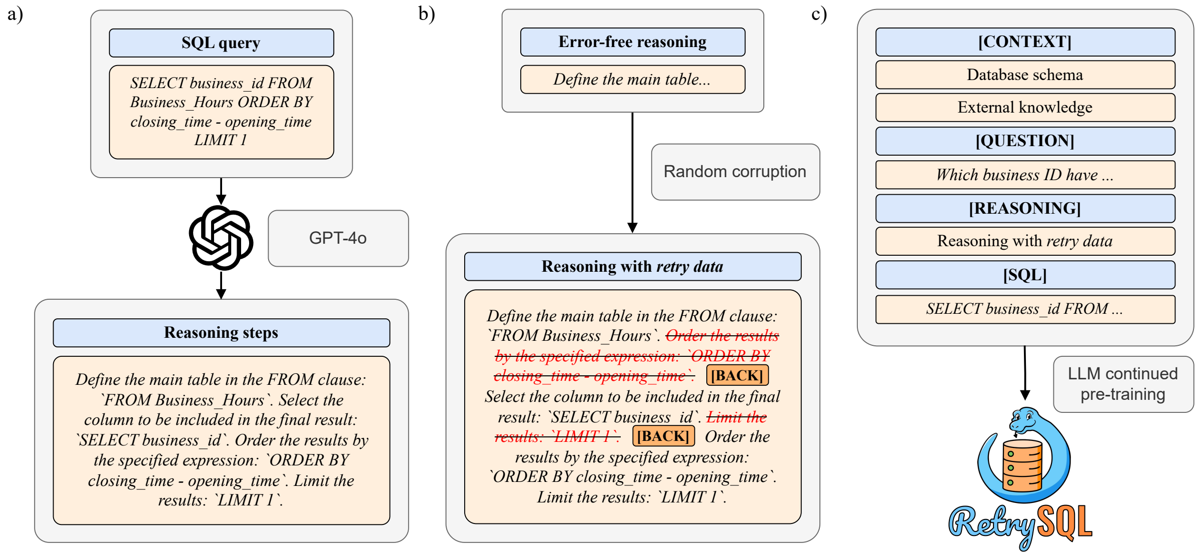

Figure 1: *RetrySQL* overview. **(a)** Reasoning step generation. For each SQL query in the training dataset, we generate a series of reasoning steps using GPT-4o. **(b)** Preparation of *retry data*. For each set of reasoning steps, we apply random perturbations, treated as errors, by replacing some steps with different ones. We follow these errors with special **[BACK]** tokens and amend them with correct steps. **(c)** We take an open-source LLM and continue its pre-training with training examples that contain *retry data* injected into reasoning steps. The resulting *RetrySQL*-trained model learns the ability to self-correct, which improves its capabilities in generating correct SQL queries from natural language questions.

retry data is generated by corrupting the order of these reasoning steps (**Fig. 1b**). The *retry data* is incorporated into the training examples and we perform continued pre-training of open-source coding-oriented LLMs, which results in a *RetrySQL*-trained models that are capable of self-correction (**Fig. 1c**). We present a set of experiments across multiple strategies of generating reasoning corruptions. We demonstrate that using *retry data* yields superior generation results when compared to training data with error-free reasoning steps.

To showcase that the self-correcting behavior is indeed learned by the model, we provide an analysis of output token confidence. We show that the max softmax score is on average lower for tokens before correction than for those after. Similarly, incorrect tokens display higher variance of softmax scores across beam search passes than the corrected ones. This illustrates that the model becomes uncertain as it makes a mistake and then self-corrects itself with higher confidence.

Our results corroborate the recent findings regarding the self-correction ability in language models (Ye et al. 2024b), demonstrating that the improvement in generation accuracy coming from the inclusion of *retry data* in pre-training is a universal law. It is applicable not only to the grade school math reasoning problem and GPT-2, but also to the text-to-SQL domain and larger, more modern Transformer-based decoder-only models. These findings suggest that *retry data* could be adapted to even more domains, especially if reasoning steps can be added to the training examples.

While the main focus of our work is on the SQL generation step in isolation, we also showcase that relatively small, 1.5B-parameter open-source coding models trained with our *RetrySQL* paradigm are competitive with much larger closed-source proprietary LLMs when used as a part of the full text-to-SQL pipeline. We share all code for *retry data* generation, as well as model training and evaluation.

In summary, our key contributions are the following:

- We introduce *RetrySQL*, a novel text-to-SQL training paradigm that makes use of reasoning steps enhanced with *retry data*.
- We show that using *retry data* in pre-training is beneficial to the generation process, as indicated by the Execution Accuracy metric calculated for the BIRD and SPIDER benchmark datasets.
- We demonstrate that *RetrySQL*-trained models have the ability to self-correct as they generate reasoning steps for the output SQL queries.
- We illustrate that within a simple end-to-end text-to-SQL pipeline, *RetrySQL*-trained 1.5B-parameter open-source coding models are competitive with proprietary models such as GPT-4o-mini and GPT-4o, which opens new possibilities for employing such small models in real-world text-to-SQL systems.

2 Related Work

Early text-to-SQL methods relied on sequence-to-sequence frameworks, using models like Graph Neural Networks, Recurrent Neural Networks, and pre-trained Transformers for encoding queries and schemas (Cai et al. 2021; Cao et al. 2021; Hwang et al. 2019), while employing slot-filling or auto-regressive decoding to generate SQL queries (Choi et al. 2020; Wang et al. 2020). Recently, the field has shifted with the emergence of LLMs, which are currently leading in the most popular benchmarks (Shi et al. 2024). While initial efforts were focused on optimizing prompt designs that leveraged in-context learning (Nan et al. 2023; Gao et al. 2023) and multi-stage prompting (Pourreza and Rafiei 2023), the current state-of-the-art is represented by LLM-based pipelines. These latest approaches integrate LLMs in more complex sequences of processing stages, with separate components for schema linking, self-correction, self-debugging, and self-consistency (Talaei et al. 2024; Pourreza et al. 2024; Lee et al. 2024; Maamari et al. 2024; Sun et al. 2024).

Compared to closed-source model prompting approaches, open-source model fine-tuning for the text-to-SQL task remains relatively unexplored (Shi et al. 2024). Many of the existing works favor parameter-efficient fine-tuning over full-parameter fine-tuning due to the former's superior training efficiency and lower training costs (Shi et al. 2024; Chen et al. 2024; Zhang et al. 2024b,a). While the majority of practitioners choose to use powerful general-purpose LLMs as their base models (Pourreza and Rafiei 2024; Xie et al. 2024; Shi et al. 2024), promising results have also been shown by adapting coding LLMs to the text-to-SQL domain (Li et al. 2024; Chen et al. 2024; Talaei et al. 2024), demonstrating that starting from a model already heavily pre-trained on coding tasks, with SQL-related training data, leads to higher performance in benchmark evaluations.

The usage of small LLMs for the text-to-SQL task is an important practical concern, as response time and costs are crucial factors in real-world applications. Existing work explored models as small as 7B parameters (Pourreza and Rafiei 2024), but the utilization of even smaller LLMs remains relatively unexplored.

It has been shown that language models trained to follow chain of thought (CoT) (Wei et al. 2023) steps excel at solving problems that involve math and symbolic reasoning (Mirzadeh et al. 2024; Sprague et al. 2024; Ye et al. 2024a,b; Muennighoff et al. 2025). Recent work in the domain of LLM theory paves the way for the discovery of LLM universal laws, including the ability to self-correct (Ye et al. 2024a,b). Test-time compute methods are an active area of research as well, with some methods indicating that reinforcement learning approaches might not be needed for eliciting self-reflection in LLMs (Madaan et al. 2023; Muennighoff et al. 2025). When it comes to small reasoning models, it has been shown that data curation (Gunasekar et al. 2023) and knowledge distillation (Mitra et al. 2023) are effective techniques for improving model performance.

The intersection of the text-to-SQL task and reasoning with self-correction for small ($\sim$1B parameters) LLMs has not been explored as of yet. *RetrySQL* addresses this gap by providing a method for teaching such small text-to-SQL models to self-correct, which greatly improves their capabilities.

3 Methodology

In this section, we describe our *RetrySQL* training paradigm in detail. We augment the training dataset (**Section 3.1**) with synthetically generated reasoning steps (**Section 3.2**). Then, we define the *retry data* generation process, in which reasoning steps are corrupted with random errors and then corrected (**Section 3.3**).

3.1 Training Data

In order to assess the efficacy of RetrySQL, we utilized two popular text-to-SQL benchmark datasets: BIRD (Li et al. 2023) and SPIDER (Yu et al. 2019). We selected them due to their relatively large training sets. For BIRD, the training data includes 9428 examples, each consisting of the database name, natural language question, external evidence, and the ground truth SQL query (SQLite dialect). In addition, the metadata for each database is available as well, consisting of a full list of tables, columns and table relations. We discovered that relations for one table, *mondial_geo*, are defined incorrectly. We excluded it from our pipeline, which left us with 9135 training examples. For SPIDER, the training data consists of 8659 examples, with a similar structure to BIRD, excluding the external evidence. We did not use the newer SPIDER 2.0 (Lei et al. 2024) dataset due to its focus on agentic evaluation - it does not include any training examples.

For the generation process, we needed to incorporate the schema information together with the question and external knowledge. To this end, we parsed the ground truth queries and prepared the matching Data Definition Language (DDL) statements, which served as the schema linking data. Importantly, unless stated otherwise, we incorporated so-called *perfect* schema linking in our experiments (i.e. no redundant links). We were interested primarily in studying the SQL generation process in isolation, without the additional task of finding relevant schema connections. We matched column and table names in each ground truth SQL query to the corresponding database metadata and built minimal required schema links.

We used DDL for schema representation because it provides a concise notation that includes table and column names, together with data types and relations. Moreover, it keeps an SQL-focused context for the model, without needing to explain specific data formats in the prompt. This approach is commonly used in existing text-to-SQL pipelines (Shi et al. 2024).

3.2 Reasoning Step Generation

Previous work showed that the usage of *retry data* in model pre-training is effective only if we also instruct the model to follow a chain of reasoning steps (Ye et al. 2024b). To this end, we needed to procure reasoning steps for each of our training examples. The training datasets do not contain this data, so we used GPT-4o for generating synthetic reasoning steps (**Fig. 1a**). Enhancing language model training data with synthetically generated components is a newly emerging trend (Abdin et al. 2024). We used a prompt that highlighted the need of reasoning steps being in a format resembling solution reasoning chains from the dataset used in previous research on self-correction (Ye et al. 2024a) (**Fig. S3**). We verified the correctness of the output formatting and also the semantic validity for a small subset of all examples, consisting of 100 instances sampled uniformly at random, thus representing a wide range of databases and query difficulties. We found that there were no cases with erroneous reasoning steps. Consequently, we assumed that the full dataset is similarly error-free. Full manual verification was not necessary, since we did not aim for the reasoning steps to be perfectly accurate. We ultimately wanted to generate SQL queries, and the reasoning steps were meant to serve as an additional training signal to the verified ground truth from the training datasets.

3.3 Retry Data Generation

We generated SQL-specific *retry data* by corrupting the solution steps prepared beforehand (Ye et al. 2024b) (**Fig. 1b**). We considered several variants of perturbations: forward single (denoted as *FS*), forward and back single (*FBS*), forward multiple (*FM*), forward and back multiple (*FBM*). Given a sequence of reasoning steps of length N, for each step r_i in that sequence we select uniformly at random (with probability p_{retry}) another step $r_{error} \in S$, where S is a set of candidate corruptions. The selection is done either once (for *FS* and *FBS*) or multiple times (for *FM* and *FBM*). For *FS* and *FM*, S consists of elements $r_{i+1}, \dots, r_N$ (i.e. future steps). For *FBS* and *FBM*, S contains elements $r_1, \dots, r_{i-1}, r_{i+1}, \dots, r_N$ (i.e. future and past steps). After selecting an element r_{error} from S, we follow it with the **[BACK]** token and then with the correct r_i itself. As such, each r_i can be replaced with the following:

- (r_{error}, **[BACK]**, r_i), for *FS* and *FBS*,
- (r_{error}, **[BACK]**, ... , r_{error}, **[BACK]**, r_i), for *FM* and *FBM*.

We generated training dataset variants for *FS*, *FBS*, *FM*, *FBM* and $p_{retry} \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$. To steer the training, we introduced additional tokens **[CONTEXT]**, **[QUESTION]**, **[REASONING]** and **[SQL]**. We combined the database schema, external knowledge, reasoning steps and the ground truth SQL query using these tokens as delimiters (**Fig. S2**).

4 Experiments

In this section, we outline the experimental setup for our study. We delineate the preliminary linear probing task (**Section 4.2**), and then describe the baseline models that we used in our experiments (**Section 4.1**). We explain our inference procedure, as well as evaluation metrics used to measure the effectiveness of all models (**Section 4.3**).

4.1 Baseline Models

We evaluated several baselines in addition to the models trained with *retry data*. We assessed zero-shot performance of GPT-4o-mini[1], GPT-4o[1], Gemini-1.5-flash[2] and Gemini-1.5-pro[2], as well as several open-source models: Llama-3.2 1.5B, Qwen2.5 1.5B, Qwen2.5-Coder 1.5B (Qwen et al. 2025) and OpenCoder 1.5B (Huang et al. 2024).

For our experiments with *retry data*, we chose from among them two coding-oriented models: OpenCoder 1.5B and Qwen2.5-Coder 1.5B. Such small model sizes allowed us to more effectively utilize our compute resources and sped up the experimentation process.

For the details on the training setup and hyperparameters, see **Appendix A.1**. For the specifics of proprietary model inference, see **Appendix A.2**.

4.2 Linear Probing Dataset

Before conducting the pre-training experiments with *retry data*, we first validated if the OpenCoder model has an innate, hidden capability of distinguishing *correct* and *incorrect* reasoning steps. If that were the case, then providing *retry data* at training time would allow the model to unlock the ability to self-correct. It has been shown previously that models pre-trained on grade school math data with correct solution steps exhibit regretful patterns in their internal states (Ye et al. 2024b).

In order to verify the above hypothesis, we designed a preliminary linear probing task. We parsed the *retry data* and categorized the training examples based on the presence of the **[BACK]** token. Each reasoning step r_i was marked as either:

- *incorrect*, if it was followed by the **[BACK]** token;
- *correct*, if it was followed by another step.

Then, we took the original BIRD data samples and extended them with reasoning step sequences ending with either *correct* or *incorrect* steps, again with the addition of special **[CONTEXT]**, **[QUESTION]** and **[REASONING]** tokens to divide the input sections. We used the Retry *FS* 0.3 dataset variant as the source for the reasoning steps. In this way, we extracted $15k$ examples in total, keeping the proportion between *correct* and *incorrect* instances balanced.

We used this data to train a classification model, in which a binary classification head categorized *correct* and *incorrect* reasoning steps (for more details, see **Appendix A.6**).

4.3 Inference Process and Evaluation Metrics

For the purpose of measuring the effectiveness of *retry data* in text-to-SQL generation, we utilized the Execution Accuracy (EX) metric. For BIRD, the development dataset contains a total of 1534 examples. The data format is the same as the training set, with the addition of a difficulty value for each example. For SPIDER, the development data numbers 1034 examples, each with a difficulty level as well.

During inference, we used the same format as during training, but omitted the part of each sequence after the **[REASONING]** token. Thus, we wanted the model to first generate the reasoning steps, and then the **[SQL]** token, followed by the actual SQL query.

During inference we used the best checkpoint of each trained model variant. We used beam search multinomial sampling with 4 beams as a decoding strategy (following (Ye et al. 2024b)), with $temperature = 0.5$, $top\_k = 50$ and $top\_p = 1.0$. We limited the number of new tokens to 1024. Each evaluation example was processed 5 times, for the purpose of measuring the model variance. The results

[1]API version: 2023-03-15-preview

[2]Stable version: 002

were post-processed by removing the **[SQL]** and preceding tokens, leaving only the SQL query.

We then calculated the EX metric in the following manner. First, the generated SQL query as well as the ground truth query for a given question were executed in an SQLite database containing the development data. Then, resulting sets of rows were compared and the ratio of matching rows was saved. Finally, the match ratios for all examples were averaged. For BIRD, we report EX values over all examples, or over just the simple, moderate or challenging ones. For SPIDER, we report five EX values: overall, easy, medium, hard and extra hard.

5 Results

In this section, we assess the effectiveness of *RetrySQL* for training SQL generation models. We showcase the baseline results for both proprietary and open-source LLMs (**Section 5.1**). We demonstrate through linear probing that the baseline OpenCoder model can recognize *incorrect* reasoning steps (**Section 5.2**). We then show that training with *retry data* improves Execution Accuracy compared to training with error-free reasoning steps (**Section 5.3**). We also explain the self-correction behavior with an analysis of model confidence around the **[BACK]** tokens (**Section 5.4**). Finally, we describe the evaluation of the full text-to-SQL pipeline (**Section 5.5**).

5.1 Zero-shot Baselines

We evaluated a selection of models in zero-shot mode (**Tab. S1**). We observe that all small open-source models fall behind proprietary models, with Qwen2.5-Coder 1.5B being the closest to GPT-4o-mini. General-purpose models (Llama-3.2 1B and Qwen2.5 1.5B) scored worse than coding-specific models (Qwen2.5-Coder 1.5B and OpenCoder 1.5B). Based on these results we selected Qwen2.5-Coder 1.5B and OpenCoder 1.5B for further experiments.

5.2 Detecting Regretful Patterns Through Linear Probing

We took the baseline OpenCoder 1.5B model and performed a linear probing experiment with its frozen weights. The linear probing model detected *incorrect* steps with average $balanced\_accuracy = 82\%$ and $f1\_score = 71\%$. Since the detection accuracy was significantly higher than 50% (random guess), we can conclude that the probing signals most likely came from the pre-trained weights, and not from the fine-tuned classification layer (Ye et al. 2024a). To support our findings, we visualize the internal state embeddings of OpenCoder using a t-SNE projection (**Fig. 2**). The plot illustrates the separability of the model's internal states when it comes to predicting *correct* versus *incorrect* reasoning steps. These results demonstrate that OpenCoder has an innate capability to self-correct, which is in line with previous research positing that this ability is a universal law for all Transformer models (Ye et al. 2024b). This justifies the usage of *retry data*, which should enable the model to backtrack as it generates an *incorrect* step and then *retry* once again.

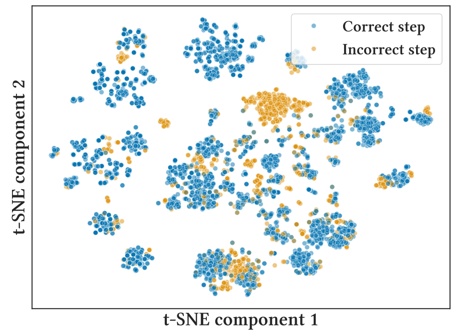

Figure 2: t-SNE projection of OpenCoder's internal state embeddings for the linear probing task. Blue points represent embeddings corresponding to *correct* reasoning steps, while orange points indicate embeddings for *incorrect* steps. The clusters of orange points indicate that the OpenCoder model differentiates a large portion of the *incorrect* steps from the *correct* ones, highlighting the innate, yet hidden, ability to detect mistakes in the reasoning process.

5.3 Retry Data Improves SQL Generation Metrics

To test if the considered models can learn the ability to self-correct, we continued the pre-training process with reasoning-enhanced training data, both error-free and with *retry data*. Compared to the zero-shot baselines, models continuously pre-trained with error-free data yielded impressive improvements in generation accuracy metrics (**Tab. 1**). These results are expected, as during training we provided the model with previously unseen domain-specific text-to-SQL training samples.

The *retry data* results show us that models continuously pre-trained with such corrupted samples lead to improved accuracy metrics when compared to the error-free continued pre-training (**Tab. 1**, **Tab. S2**). Out of four approaches to *retry data* preparation, the *FS* variant proved to be the best overall. The highest improvements in overall generation accuracy were observed for p_{retry}=0.2 and 0.3. For the results pertaining to the other variants, see **Appendix A.3**.

The effectiveness of *RetrySQL* transfers across models and benchmark datasets. For OpenCoder 1.5B, the overall Execution Accuracy improved by ∼4 p.p. and and 3.1 p.p., measured on BIRD and SPIDER benchmarks, respectively. For Qwen2.5-Coder 1.5B, the overall improvements were 0.4 p.p. and 3.93 p.p., on BIRD and SPIDER, respectively.

While the $\text{EX}_{overall}$ metric does show the effectiveness of using *retry data* for improving text-to-SQL generation accuracy in general, it does not show the full picture. *RetrySQL* works particularly well for the hardest evaluation samples (**Tab. 1**). Previous research postulated that the advantage of the self-correction ability can be observed especially for complex out-of-distribution evaluation examples, which require the longest solution reasoning sequences (Ye et al. 2024b). We found this to be the case for most model and benchmark combinations. It is worth noting that the number of operations necessary to explain SQL queries is not

Model	BIRD				
	Dataset variant	EX_{simple}	$EX_{moderate}$	$EX_{challenging}$	$EX_{overall}$
OpenCoder 1.5B	– (zero-shot)	47.14 ± 0.45	27.63 ± 0.44	17.52 ± 0.55	38.44 ± 0.43
	error-free	62.70 ± 0.07	43.53 ± 0.14	39.45 ± 0.28	54.71 ± 0.08
	Retry *FS* 0.2	**68.22 ± 0.12**	**45.47 ± 0.14**	40.28 ± 0.34	**58.70 ± 0.09**
	Retry *FS* 0.3	68.00 ± 0.00	44.91 ± 0.26	**43.31 ± 0.28**	58.68 ± 0.06

Model	SPIDER					
	Dataset variant	EX_{easy}	EX_{medium}	EX_{hard}	EX_{extra}	$EX_{overall}$
OpenCoder 1.5B	error-free	90.24 ± 0.18	73.59 ± 0.10	**69.88 ± 0.31**	56.63 ± 0.00	74.25 ± 0.08
	Retry *FS* 0.2	**91.53 ± 0.00**	80.27 ± 0.00	66.09 ± 0.00	**60.12 ± 0.27**	**77.35 ± 0.04**
	Retry *FS* 0.3	89.60 ± 0.18	**80.94 ± 0.00**	62.76 ± 0.26	59.52 ± 0.27	76.52 ± 0.08

Model	BIRD				
	Dataset variant	EX_{simple}	$EX_{moderate}$	$EX_{challenging}$	$EX_{overall}$
Qwen2.5-Coder 1.5B	– (zero-shot)	46.77 ± 0.44	24.78 ± 0.39	11.45 ± 0.94	36.78 ± 0.44
	error-free	**65.47 ± 0.13**	45.04 ± 0.00	30.21 ± 0.28	55.96 ± 0.10
	Retry *FS* 0.2	63.96 ± 0.09	45.47 ± 0.00	35.17 ± 0.00	55.65 ± 0.05
	Retry *FS* 0.3	63.91 ± 0.08	**46.64 ± 0.22**	**39.10 ± 0.00**	**56.36 ± 0.05**

Model	SPIDER					
	Dataset variant	EX_{easy}	EX_{medium}	EX_{hard}	EX_{extra}	$EX_{overall}$
Qwen2.5-Coder 1.5B	error-free	**92.34 ± 0.00**	74.75 ± 0.11	65.52 ± 0.00	48.80 ± 0.00	73.25 ± 0.05
	Retry *FS* 0.2	90.40 ± 0.16	**80.94 ± 0.00**	**69.66 ± 0.43**	**55.18 ± 0.30**	**77.18 ± 0.06**
	Retry *FS* 0.3	91.13 ± 0.00	77.76 ± 0.09	66.55 ± 0.23	53.01 ± 0.00	75.11 ± 0.08

Table 1: Execution Accuracy for OpenCoder 1.5 and Qwen2.5-Coder 1.5 trained with *RetrySQL*. All results are expressed in percentages, with mean and standard deviation over 5 multinomial beam search generations. The best results are marked in bold. Results with *retry data* that improve upon the error-free training are indicated with an underline.

perfectly correlated with the difficulty level (**Fig. S1**). As such, many challenging examples can be solved with relatively short queries, which do not require the model to generate long reasoning step sequences. This might explain why for OpenCoder 1.5B and BIRD the results are spread more evenly across difficulty levels.

5.4 Does *RetrySQL* Know That It Makes Mistakes?

The EX results show us that training with *RetrySQL* increases the number of correct SQL queries compared to models trained without *retry data* (**Tab. 1**). However, it could be the case that the additional tokens present in reasoning steps in *retry data* simply act as robust augmentation and the model does not learn to self-correct. The underlying model behavior needs to be studied in more detail. To this end, we analyzed the confidence scores returned by the model trained with the Retry *FS* 0.3 dataset in proximity (radius of 10 tokens) to **[BACK]** tokens. We took max softmax scores from these tokens, and then calculated the mean and standard deviation per token across 10 multinomial beam search passes. Finally, we averaged these metrics separately for tokens before and after each **[BACK]** token.

It is evident that the mean of max confidence scores for predicted tokens differs between tokens preceding the **[BACK]** token and those after it (**Fig. 3a**). In other words, as the model is making mistakes, it is less confident in its predictions than after it self-corrects itself and starts to generate correct tokens. This shows that the ability to self-correct is an active part of the generation process.

Similarly, the standard deviation of confidence scores across beam search passes differs between tokens before and those after the **[BACK]** token (**Fig. 3b**). For the incorrect tokens, the variance is on average much higher than for the ones after the self-correction. This indicates a reduction in model uncertainty - before the **[BACK]** token each beam search pass returns significantly different results, the model is not decided what to choose. Conversely, after self-correction, the results become much more consistent and certain - the model catches the error and commits to the correction.

Both of these results show that the self-correcting ability is a learned behavior, resulting from the inclusion of *retry data* in the training process. This new skill is evident when we analyze the generated reasoning steps and SQL queries. The error-free model can hallucinate non-existent tables or JOIN conditions in its reasoning and then include them in the SQL query. The *RetrySQL*-trained model generates correct SQLs in these cases. For examples of such model outputs, see **Appendix A.9**. For an additional analysis of generated reasoning steps, see **Appendix A.4**.

5.5 RetrySQL-trained Model in an End-to-end Text-to-SQL Pipeline

All experiments presented thus far focused strictly on the generation step, with *perfect* pre-computed schema link-

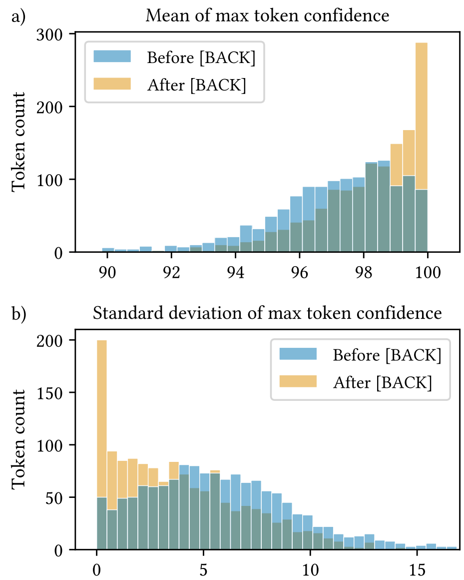

Figure 3: Distribution of token confidence before and after **[BACK]** tokens. **(a)** Mean of max token confidence across 10 beam search passes. It can be seen that the confidence score is on average much higher for tokens after the **[BACK]** token, indicating that the model is uncertain as it makes mistakes, but is confident after self-correction. **(b)** Standard deviation of max token confidence across 10 beam search passes. The variance of model predictions is much higher as it makes mistakes than after self-correction.

ing. However, in a real text-to-SQL pipeline, perfect schema linking is not available. In order to validate if *RetrySQL*-trained models are competitive with existing proprietary models in the full end-to-end pipeline setting, we performed an additional set of experiments. We observed that the *RetrySQL*-trained models (OpenCoder 1.5B, BIRD training data) compare favorably to much larger models (**Tab. S5**): they achieve $\mathrm{EX}_{overall}$ of up to 51.36, compared to 32.53 and 54.99 for GPT-4o-mini and GPT-4o, respectively. For the complete description of the pipeline, together with details on our schema linking methodology and full evaluation results, see **Appendix A.5**. These results indicate that using *retry data* in conjunction with our *RetrySQL* method produces 1.5B-parameter models that are competitive with much larger proprietary models ($\sim$8B and $\sim$200B parameters for GPT-4o-mini and GPT-4o (Abacha et al. 2025), respectively). This is a promising outcome, showing that incorporating self-correction in the generation stage might be a way forward for future text-to-SQL end-to-end pipelines.

6 Limitations

In our experiments we used the training data from the BIRD and SPIDER benchmark datasets, which contain a limited number of training examples. This is different than what has been done in previous work on self-correction with *retry data* (Ye et al. 2024b). There, the training examples were generated on demand as the training went on, to fill a preset number of training steps. We did not have a setup for generating synthetic data in that way, and had to rely on the curated training examples from BIRD. However, a direct comparison to our results is not obvious, as the problem setting of grade school math is very different to our text-to-SQL task. In addition, the metric used in that work was a direct measure of correctness, while in our case we used an indirect Execution Accuracy metric computed in relation to database values, which were not present in the training data. We leave synthetic training data generation as a topic for future work.

We did not consider evaluating *RetrySQL* scaling with model size, because we specifically wanted to address the problem of small model performance in the text-to-SQL task. In addition, there are some challenges related to larger models. According to LLM scaling laws (Kaplan et al. 2020), in order to avoid overfitting when model size is increased, the data volume has to increase accordingly. As indicated above, we did not have access to large annotated text-to-SQL datasets. Furthermore, full-parameter training of larger models is quite intensive in terms of compute resources.

For the full pipeline experiments, we utilized only a relatively simple LLM-based schema linking stage and did not include a correction stage at the end. This is not an ideal strategy, as there are many optimizations that could be applied to these stages. However, the main part of our research focused on the generation stage and these other elements remained out of scope for us. Moreover, because the *RetrySQL* training paradigm teaches the generation model to *self*-correct, the additional correction step becomes less important. We leave building a fully optimized end-to-end text-to-SQL pipeline, with the most recent approaches to schema linking and query selection, as a topic for future research.

7 Conclusions

In this paper, we presented *RetrySQL*, a novel approach to training text-to-SQL generation models. Our solution utilizes reasoning steps with *retry data* in the training examples, which teaches the generation model to self-correct itself as it produces its output. We show that using such data for the continued pre-training of a coding LLM leads to improved Execution Accuracy metrics when compared to models pre-trained without *retry data*. We provide an explainability analysis for our results - we show that as the *RetrySQL* model makes mistakes, it is less confident in its predictions than after it self-corrects itself. Finally, we showcase that incorporating *RetrySQL*-trained 1.5B-parameter models into a relatively simple end-to-end text-to-SQL pipeline produces results that are competitive with much larger closed-source proprietary LLMs such as GPT-4o-mini and GPT-4o, making them much more viable for real-world deployment. We hope that our *RetrySQL* training paradigm will lead to further developments in text-to-SQL models, especially in the context of self-correcting generation.

References

Abacha, A. B.; wai Yim, W.; Fu, Y.; Sun, Z.; Yetisgen, M.; Xia, F.; and Lin, T. 2025. MEDEC: A Benchmark for Medical Error Detection and Correction in Clinical Notes. arXiv:2412.19260.

Abdin, M.; Aneja, J.; Behl, H.; Bubeck, S.; Eldan, R.; Gunasekar, S.; Harrison, M.; Hewett, R. J.; Javaheripi, M.; Kauffmann, P.; Lee, J. R.; Lee, Y. T.; Li, Y.; Liu, W.; Mendes, C. C. T.; Nguyen, A.; Price, E.; de Rosa, G.; Saarikivi, O.; Salim, A.; Shah, S.; Wang, X.; Ward, R.; Wu, Y.; Yu, D.; Zhang, C.; and Zhang, Y. 2024. Phi-4 Technical Report. arXiv:2412.08905.

Cai, R.; Yuan, J.; Xu, B.; and Hao, Z. 2021. SADGA: Structure-Aware Dual Graph Aggregation Network for Text-to-SQL. In *Advances in Neural Information Processing Systems*, volume 34, 7664–7676. Curran Associates, Inc.

Cao, R.; Chen, L.; Chen, Z.; Zhao, Y.; Zhu, S.; and Yu, K. 2021. LGESQL: Line Graph Enhanced Text-to-SQL Model with Mixed Local and Non-Local Relations. In Zong, C.; Xia, F.; Li, W.; and Navigli, R., eds., *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2541–2555. Online: Association for Computational Linguistics.

Chen, X.; Wang, T.; Qiu, T.; Qin, J.; and Yang, M. 2024. Open-SQL Framework: Enhancing Text-to-SQL on Open-source Large Language Models. arXiv:2405.06674.

Choi, D.; Shin, M.; Kim, E.; and Shin, D. R. 2020. RYAN-SQL: Recursively Applying Sketch-based Slot Fillings for Complex Text-to-SQL in Cross-Domain Databases. *CoRR*, abs/2004.03125.

Gao, D.; Wang, H.; Li, Y.; Sun, X.; Qian, Y.; Ding, B.; and Zhou, J. 2023. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. arXiv:2308.15363.

Gunasekar, S.; Zhang, Y.; Aneja, J.; Mendes, C. C. T.; Giorno, A. D.; Gopi, S.; Javaheripi, M.; Kauffmann, P.; de Rosa, G.; Saarikivi, O.; Salim, A.; Shah, S.; Behl, H. S.; Wang, X.; Bubeck, S.; Eldan, R.; Kalai, A. T.; Lee, Y. T.; and Li, Y. 2023. Textbooks Are All You Need. arXiv:2306.11644.

Huang, S.; Cheng, T.; Liu, J. K.; Hao, J.; Song, L.; Xu, Y.; Yang, J.; Liu, J. H.; Zhang, C.; Chai, L.; Yuan, R.; Zhang, Z.; Fu, J.; Liu, Q.; Zhang, G.; Wang, Z.; Qi, Y.; Xu, Y.; and Chu, W. 2024. OpenCoder: The Open Cookbook for Top-Tier Code Large Language Models. ArXiv:2411.04905 [cs].

Hwang, W.; Yim, J.; Park, S.; and Seo, M. 2019. A Comprehensive Exploration on WikiSQL with Table-Aware Word Contextualization. *CoRR*, abs/1902.01069.

Kaplan, J.; McCandlish, S.; Henighan, T.; Brown, T. B.; Chess, B.; Child, R.; Gray, S.; Radford, A.; Wu, J.; and Amodei, D. 2020. Scaling Laws for Neural Language Models. arXiv:2001.08361.

Lee, D.; Park, C.; Kim, J.; and Park, H. 2024. MCS-SQL: Leveraging Multiple Prompts and Multiple-Choice Selection For Text-to-SQL Generation. arXiv:2405.07467.

Lei, F.; Chen, J.; Ye, Y.; Cao, R.; Shin, D.; Su, H.; Suo, Z.; Gao, H.; Hu, W.; Yin, P.; Zhong, V.; Xiong, C.; Sun, R.; Liu, Q.; Wang, S.; and Yu, T. 2024. Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. arXiv:2411.07763.

Li, H.; Zhang, J.; Liu, H.; Fan, J.; Zhang, X.; Zhu, J.; Wei, R.; Pan, H.; Li, C.; and Chen, H. 2024. CodeS: Towards Building Open-source Language Models for Text-to-SQL. ArXiv:2402.16347 [cs].

Li, J.; Hui, B.; Qu, G.; Yang, J.; Li, B.; Li, B.; Wang, B.; Qin, B.; Cao, R.; Geng, R.; Huo, N.; Zhou, X.; Ma, C.; Li, G.; Chang, K. C. C.; Huang, F.; Cheng, R.; and Li, Y. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. ArXiv:2305.03111 [cs].

Maamari, K.; Abubaker, F.; Jaroslawicz, D.; and Mhedhbi, A. 2024. The Death of Schema Linking? Text-to-SQL in the Age of Well-Reasoned Language Models. ArXiv:2408.07702 [cs].

Madaan, A.; Tandon, N.; Gupta, P.; Hallinan, S.; Gao, L.; Wiegreffe, S.; Alon, U.; Dziri, N.; Prabhumoye, S.; Yang, Y.; Gupta, S.; Majumder, B. P.; Hermann, K.; Welleck, S.; Yazdanbakhsh, A.; and Clark, P. 2023. Self-Refine: Iterative Refinement with Self-Feedback. arXiv:2303.17651.

Mirzadeh, I.; Alizadeh, K.; Shahrokhi, H.; Tuzel, O.; Bengio, S.; and Farajtabar, M. 2024. GSM-Symbolic: Understanding the Limitations of Mathematical Reasoning in Large Language Models. arXiv:2410.05229.

Mitra, A.; Corro, L. D.; Mahajan, S.; Codas, A.; Simoes, C.; Agarwal, S.; Chen, X.; Razdaibiedina, A.; Jones, E.; Aggarwal, K.; Palangi, H.; Zheng, G.; Rosset, C.; Khanpour, H.; and Awadallah, A. 2023. Orca 2: Teaching Small Language Models How to Reason. arXiv:2311.11045.

Muennighoff, N.; Yang, Z.; Shi, W.; Li, X. L.; Fei-Fei, L.; Hajishirzi, H.; Zettlemoyer, L.; Liang, P.; Candès, E.; and Hashimoto, T. 2025. s1: Simple test-time scaling. arXiv:2501.19393.

Nan, L.; Zhao, Y.; Zou, W.; Ri, N.; Tae, J.; Zhang, E.; Cohan, A.; and Radev, D. 2023. Enhancing Few-shot Text-to-SQL Capabilities of Large Language Models: A Study on Prompt Design Strategies. arXiv:2305.12586.

Pourreza, M.; Li, H.; Sun, R.; Chung, Y.; Talaei, S.; Kakkar, G. T.; Gan, Y.; Saberi, A.; Ozcan, F.; and Arik, S. O. 2024. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. arXiv:2410.01943.

Pourreza, M.; and Rafiei, D. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. ArXiv:2304.11015 [cs].

Pourreza, M.; and Rafiei, D. 2024. DTS-SQL: Decomposed Text-to-SQL with Small Large Language Models.

Qwen; :; Yang, A.; Yang, B.; Zhang, B.; Hui, B.; Zheng, B.; Yu, B.; Li, C.; Liu, D.; Huang, F.; Wei, H.; Lin, H.; Yang, J.; Tu, J.; Zhang, J.; Yang, J.; Yang, J.; Zhou, J.; Lin, J.; Dang, K.; Lu, K.; Bao, K.; Yang, K.; Yu, L.; Li, M.; Xue, M.; Zhang, P.; Zhu, Q.; Men, R.; Lin, R.; Li, T.; Tang, T.;

Xia, T.; Ren, X.; Ren, X.; Fan, Y.; Su, Y.; Zhang, Y.; Wan, Y.; Liu, Y.; Cui, Z.; Zhang, Z.; and Qiu, Z. 2025. Qwen2.5 Technical Report. arXiv:2412.15115.

Shi, L.; Tang, Z.; Zhang, N.; Zhang, X.; and Yang, Z. 2024. A Survey on Employing Large Language Models for Text-to-SQL Tasks. ArXiv:2407.15186 [cs].

Sprague, Z.; Yin, F.; Rodriguez, J. D.; Jiang, D.; Wadhwa, M.; Singhal, P.; Zhao, X.; Ye, X.; Mahowald, K.; and Durrett, G. 2024. To CoT or not to CoT? Chain-of-thought helps mainly on math and symbolic reasoning. arXiv:2409.12183.

Sun, R.; Arik, S.; Muzio, A.; Miculicich, L.; Gundabathula, S.; Yin, P.; Dai, H.; Nakhost, H.; Sinha, R.; Wang, Z.; and Pfister, T. 2024. SQL-PaLM: Improved Large Language Model Adaptation for Text-to-SQL (extended). arXiv:2306.00739.

Talaei, S.; Pourreza, M.; Chang, Y.-C.; Mirhoseini, A.; and Saberi, A. 2024. CHESS: Contextual Harnessing for Efficient SQL Synthesis. ArXiv:2405.16755 [cs].

Team, D.-A. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948.

Team, G. 2024a. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv:2403.05530.

Team, O. 2024b. GPT-4 Technical Report. arXiv:2303.08774.

Wang, B.; Shin, R.; Liu, X.; Polozov, O.; and Richardson, M. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 7567–7578. Online: Association for Computational Linguistics.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Ichter, B.; Xia, F.; Chi, E.; Le, Q.; and Zhou, D. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903.

Xie, Y.; Jin, X.; Xie, T.; Lin, M.; Chen, L.; Yu, C.; Cheng, L.; Zhuo, C.; Hu, B.; and Li, Z. 2024. Decomposition for Enhancing Attention: Improving LLM-based Text-to-SQL through Workflow Paradigm. arXiv:2402.10671.

Ye, T.; Xu, Z.; Li, Y.; and Allen-Zhu, Z. 2024a. Physics of Language Models: Part 2.1, Grade-School Math and the Hidden Reasoning Process. arXiv:2407.20311.

Ye, T.; Xu, Z.; Li, Y.; and Allen-Zhu, Z. 2024b. Physics of Language Models: Part 2.2, How to Learn From Mistakes on Grade-School Math Problems. arXiv:2408.16293.

Yu, T.; Zhang, R.; Yang, K.; Yasunaga, M.; Wang, D.; Li, Z.; Ma, J.; Li, I.; Yao, Q.; Roman, S.; Zhang, Z.; and Radev, D. 2019. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. arXiv:1809.08887.

Zhang, C.; Mao, Y.; Fan, Y.; Mi, Y.; Gao, Y.; Chen, L.; Lou, D.; and Lin, J. 2024a. FinSQL: Model-Agnostic LLMs-based Text-to-SQL Framework for Financial Analysis. arXiv:2401.10506.

Zhang, T.; Chen, C.; Liao, C.; Wang, J.; Zhao, X.; Yu, H.; Wang, J.; Li, J.; and Shi, W. 2024b. SQLfuse: Enhancing Text-to-SQL Performance through Comprehensive LLM Synergy. arXiv:2407.14568.