ADVSQL: Automated Discovery and Exploitation of Vulnerabilities in Text-to-SQL Models

Anonymous ACL submission

Abstract

Large language models (LLMs) have achieved significant progress on Text-to-SQL tasks. However, they still exhibit inherent vulnerabilities, producing incorrect or unsafe SQL queries when exposed to minor input or schema perturbations. The lack of automated, scalable methods to identify and exploit these weaknesses limits the development of truly robust models. To address these challenges, we propose ADVSQL, an automated framework designed to systematically discover and exploit vulnerabilities in Text-to-SQL models. Our method follows an iterative pipeline with three core modules: robustness sample generation and exploration, vulnerability extraction and consolidation, and vulnerability refinement and repository update. ADVSQL generates perturbations from both query and schema perspectives, distills recurring failure patterns into reusable exploits, and further refines these patterns to uncover subtler vulnerabilities. Experiments on BIRD and Spider datasets demonstrate the effectiveness of ADVSQL for Text-to-SQL tasks. By leveraging data generated by ADVSQL from identified vulnerabilities, models can be trained or evaluated on diverse and challenging scenarios, thereby improving their robustness.

1 Introduction

Text-to-SQL is a critical bridge between natural language processing and database systems, enabling users to retrieve structured information seamlessly through natural language interfaces. On standard benchmarks such as Spider (Yu et al., 2018) and BIRD (Li et al., 2023b), recent large language models (LLMs) have demonstrated high execution accuracy (Hong et al., 2024; Shi et al., 2025; Huang et al., 2025; Liu et al., 2025b; Xie et al., 2025). However, these benchmark results often conceal a fundamental fragility when models are deployed in real-world applications. In practice, models must accommodate diverse user inputs and complex database schemas, yet even minor perturbations or semantically equivalent variations can lead to incorrect SQL queries (Gan et al., 2021a; Deng et al., 2021; Sahitaj et al., 2025). Such fragility poses a serious risk for large-scale data access, as small structural errors in SQL can lead to query failures or misleading results.

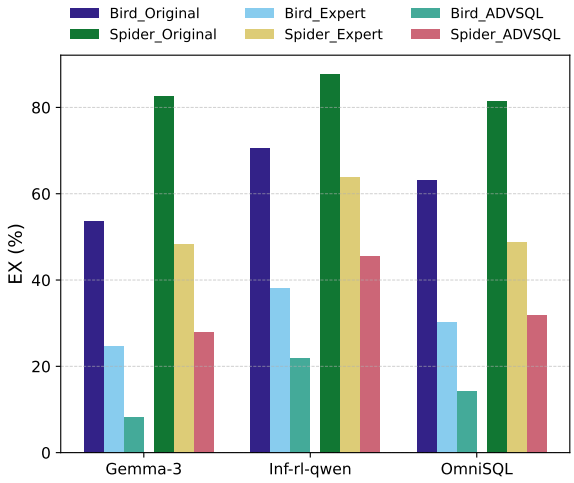

Figure 1: Accuracy comparison of various LLMs on BIRD and Spider datasets, showing their performance before and after applying our ADVSQL framework.

Prior studies reveal that Text-to-SQL models suffer significant performance degradation under perturbations (Huang et al., 2025). Dr.Spider (Chang et al., 2023) reports a performance drop of at least 14% when perturbations are applied to queries, databases, or SQL. ADVETA (Gao et al., 2022) highlights model vulnerabilities in handling schema variations. However, existing robustness evaluations, from those focused on surface-level query rewrites (Gan et al., 2021a; Patil et al., 2023; Bhaskar et al., 2023) to more comprehensive benchmarks like Dr.Spider (Chang et al., 2023), predominantly rely on static perturbation sets crafted through manual rules. This reliance on pre-defined rules, rather than dynamic, context-aware genera-

tion, may limit the ability to systematically uncover complex failure modes. As a result, such evaluations could provide only a partial assessment of robustness and offer limited insights into the underlying causes of model failures.

To this end, we propose ADVSQL, an automated framework that systematically discovers and exploits vulnerabilities in Text-to-SQL models. Unlike traditional methods based on static human annotations or one-off adversarial examples, ADVSQL operates in a closed loop to cost-effectively uncover latent failure modes in LLMs. It generates semantically equivalent perturbations across three angles: queries, relevant schema elements, and irrelevant schema elements, capturing diverse error sources from input formulation to schema understanding and noise sensitivity. Through iterative perturbation and failure induction, ADVSQL uncovers more than 4,000 issues along with the mechanisms that trigger each failure and distills them into seven vulnerability patterns, including errors related to syntax, question interpretation, table relationships, column semantics, constraints, aggregation, and other semantic aspects. Experiments on BIRD and Spider datasets demonstrate the effectiveness of our framework. As shown in Figure 1, three LLMs exhibit 47.65% and 48.91% lower average execution accuracy on BIRD and Spider, respectively, exceeding the impact of expert-designed perturbations and revealing a broader and more informative spectrum of model vulnerabilities. Moreover, by leveraging data generated by ADVSQL from identified vulnerabilities, models can be trained or evaluated on diverse and challenging scenarios, thereby improving their robustness.

Our contributions are summarized as follows:

- We introduce an automated framework for the systematic discovery and exploitation of vulnerabilities in Text-to-SQL models. It identifies diverse, fine-grained failure patterns and generates robustness-oriented datasets from a small set of seed examples.
- By leveraging data generated by ADVSQL from identified vulnerabilities, models can be trained or evaluated on more challenging inputs, improving robustness efficiently.
- The vulnerability repository produced by ADVSQL is transferable across models, capturing model-agnostic failure patterns, enabling cross-model evaluation.

2 Related Work

2.1 LLM-based Text-to-SQL Generation

With the rapid advancement of LLMs, LLM-based approaches to Text-to-SQL have demonstrated more enhanced transferability and reasoning capabilities than traditional rule-based systems (Mahmud et al., 2015; Lyu et al., 2020), neural models (Choi et al., 2021; Xu et al., 2017), and pretrained models (Li et al., 2023a; Yin et al., 2020), propelling the field into a new stage (Hong et al., 2024; Shi et al., 2025). Current SQL generation methods with LLMs mainly follow two paradigms: prompt engineering and fine-tuning. Prompt engineering methods are typically based on closed-source LLMs and improve SQL generation quality through techniques such as task decomposition (Pourreza and Rafiei, 2023; Xie et al., 2025), chain-of-thought (CoT) reasoning (Xu et al., 2024; Liu et al., 2025a), and multi-agent collaboration (Xia et al., 2024; Deng et al.). Fine-tuning methods, in contrast, rely on open-source LLMs to reduce inference costs and enhance privacy, controllability, and stability (Li et al., 2025; Qu et al., 2025; Guo et al., 2025; Li et al., 2024; Sheng et al., 2025; Cheng et al., 2025). Most prior work focuses on improving overall performance or addressing isolated subtasks. Few studies treat robustness as a primary concern, and even fewer provide systematic methods to investigate the causes of SQL generation failures in realistic settings, highlighting the need for frameworks that go beyond accuracy to analyze model behavior more comprehensively.

2.2 Research on Robustness in Text-to-SQL

The Text-to-SQL field has established a multi-dimensional evaluation framework. Foundational datasets such as WikiSQL (Zhong et al., 2017), Spider (Yu et al., 2018), KaggleDBQA (Lee et al., 2021), and BIRD (Li et al., 2023b) establish the foundation for evaluating model capability. Robustness studies build on these resources through derived test sets, following three primary paradigms: (1) Semantic-preserving perturbations (Deng et al., 2021; Gan et al., 2021b,a; Sahitaj et al., 2025). (2) Rule-based manual perturbations across queries, schemas, and SQL logic (Chang et al., 2023), or reorganizing test cases to evaluate structural complexity and lexical generalization (Patil et al., 2023). (3) Systematic evaluations that target ambiguity and structural variation (Bhaskar et al., 2023; Saparina and Lapata, 2024; Pi et al., 2022; Zhang et al.,

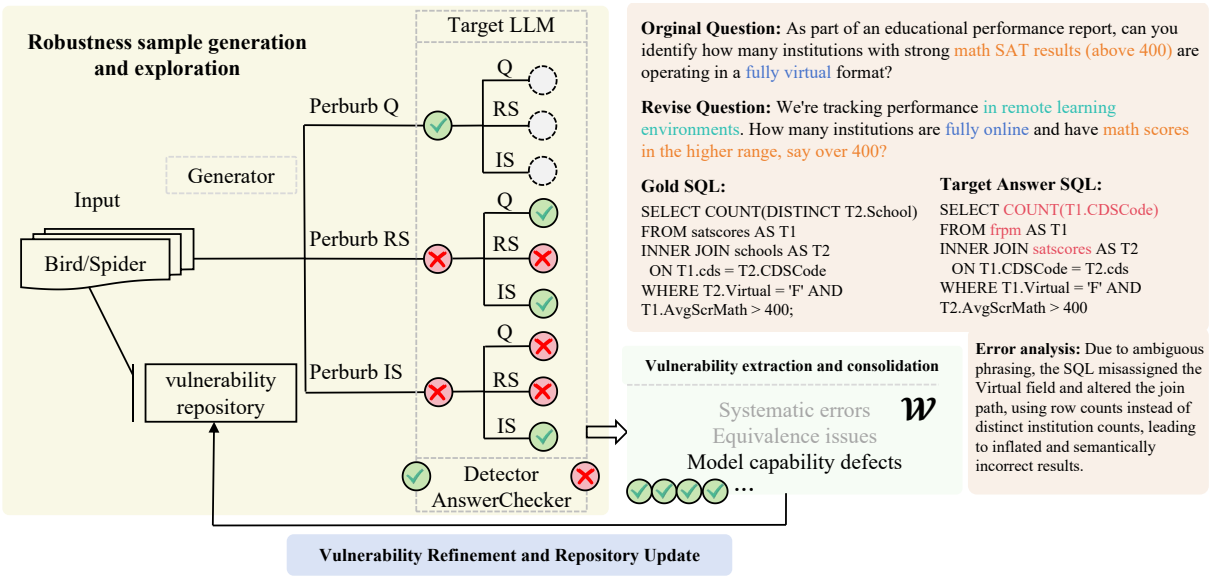

Figure 2: Overview of ADVSQL, which consists of three collaborative modules: robustness sample generation and exploration, vulnerability extraction and consolidation, and strategy retrieval and iteration, together with perturbations along three angles, `Perturb Q` as queries, `Perturb RS` as relevant schema elements, and `Perturb IS` as irrelevant schema elements

2024). Existing studies on adversarial robustness remain largely confined to semantic-level or rule-based perturbations, lacking deeper diagnostic insights into model behavior. To address the scalability limitations of manual methods, recent work has developed automated robustness evaluation approaches (Waghela et al., 2024; Paulus et al., 2024; Geisler et al., 2025; Fang et al., 2025; Liu et al., 2024; Zhang, 2025). These methods hold the promise of systematically exploring vast perturbation spaces to uncover unexpected failure cases. However, they are typically designed for general-domain robustness assessment, focusing on measuring a model's resilience rather than diagnosing its core weaknesses. To bridge this gap, we introduce a framework that automatically perturbs both natural language queries and database schemas, not merely to test robustness, but to systematically expose and analyze the latent weaknesses within Text-to-SQL models.

3 Method

This section details ADVSQL, our framework for automated vulnerability discovery and refinement. Our method is structured as an iterative loop comprising three core modules: robustness sample generation and exploration (Section 3.1), vulnerability extraction and consolidation (Section 3.2), and strategy retrieval and iteration (Section 3.3). The overall architecture is illustrated in Figure 2. Before the main iterative loop, ADVSQL performs a warm-up phase to initialize the vulnerability repository, as detailed in Section 3.4. The full algorithmic procedure for one iteration of the main automated cycle is detailed in Appendix B. The subsequent sections elaborate on each component.

3.1 Robustness sample generation and exploration

This module generates candidate robustness samples through controllable perturbations and selects high-value failures that reveal latent vulnerabilities while preserving semantic validity. The generation is guided by the refined dynamic vulnerability repository $\mathcal{W}^*_{t-1}$, initialized during the warm-up phase (as detailed in Section 3.4).

At iteration t, we sample a case (Q, DB) for which the target model produces a correct SQL query. A perturbation is generated iteratively, guided by the modification intention i_{t-1} inherited from the previous iteration (or from the warm-up phase when $t = 1$). The intention is encoded as $v_{\text{query}} = E(i_{t-1})$. We then retrieve the top-k relevant vulnerabilities from $\mathcal{W}^*_{t-1}$ by cosine similarity:

$$\Gamma_t = \text{Top-k}\Big(\{\text{sim}(v_{\text{query}}, v_w) \mid w \in \mathcal{W}^*_{t-1}\}\Big), \quad (1)$$

where $v_w = E(w.\texttt{error_summary})$. Here, Γ_t serves as guidance for perturbation generation: while similarity is computed on the abstract

`error_summary`, each entry additionally provides the `error_detail` for concrete modifications.

The Generator LLM then uses the original case (Q, DB) and the retrieved guidance Γ_t to jointly generate a perturbed case $(\tilde{Q}, \tilde{DB})$ and a new modification intent i_t, which will guide the next iteration. Perturbations follow high-level angles $\mathbb{A}$ applied to queries or schema elements, encompassing single or composite operations to ensure diverse and meaningful perturbations. Detailed prompts are provided in Appendix F. Unified contextual input is provided via M-schema (Gao et al., 2024), including table names, column names, relationships, descriptions, and example values, to mitigate failures caused by missing context.

Sample quality control is handled by a two-stage validation process using a **Detector** and an **AnswerChecker** to isolate failures caused by strictly answer-preserving perturbations.

First, the Detector, implemented using *Qwen-32B*, certifies that a perturbed input $(\tilde{Q}, \tilde{DB})$ does not alter the execution result of the original gold query, $SQL^{\star}$. A perturbation is deemed valid only if it is semantically equivalent and preserves the ground truth answer. The high reliability of the Detector in this task is confirmed through strong agreement with human annotations (Section A).

For perturbations validated by the Detector, the AnswerChecker then executes the model-predicted SQL $\tilde{SQL}$ on the perturbed database $\tilde{DB}$:

$$\tilde{R} = \text{Exec}(\tilde{SQL}, \tilde{DB}), \tag{2}$$

and compares it against the original, cached gold result:

$$R_{\text{gold}} = \text{Exec}(SQL^{\star}, DB). \tag{3}$$

A perturbed sample is marked as an **A-level** failure only when the perturbation is certified as answer-preserving by the Detector, but the model's output fails to match the original gold result:

$$\begin{aligned} \text{Detector}((Q, DB), (\tilde{Q}, \tilde{DB})) &= \text{Valid}, \\ \text{AnswerChecker}(\tilde{R}, R_{\text{gold}}) &= \text{Incorrect}. \end{aligned} \tag{4}$$

These A-level samples serve as the primary input to the vulnerability extraction module.

3.2 Vulnerability extraction and consolidation

This module converts the set of A-level samples, A, into structured and interpretable vulnerability entries that are stored in the vulnerability repository, $\mathcal{W}$. Only those samples where the perturbation successfully exposes a model failure are retained in the repository, ensuring that $\mathcal{W}$ focuses exclusively on confirmed vulnerabilities.

For each failure sample, a Summarizer processes the original state (Q, DB), the perturbed state $(\tilde{Q}, \tilde{DB})$, the model's erroneous output $\tilde{R}$, and the modification intention i_t that was generated alongside the perturbation. It classifies the failure, identifies its underlying cause, and encodes this information into a structured vulnerability entry w. Each entry includes three components: error category, which provides a high-level classification of the failure type; error summary, which offers an abstract linguistic description of the issue in natural language; and error detail, which records concrete, low-level information such as schema modifications, query perturbations, and contextual cues that led to the model failure.

Formally, this process can be represented as a mapping function:

$$w = f_{\text{sum}}\big((Q, DB), (\tilde{Q}, \tilde{DB}), \tilde{R}, i_t\big) \tag{5}$$

The resulting entry is then stored for future retrieval and reuse: $\mathcal{W} \leftarrow \mathcal{W} \cup \{w\}$.

The Summarizer uses heuristic rules to automatically classify errors and generalize their underlying causes: (1) Systematic errors arise from insufficient context or incomplete input, preventing a valid model response. (2) Equivalence issues occur when model outputs are semantically equivalent to the reference answer but incorrectly judged due to superficial differences. (3) Model capability defects arise from deficiencies in reasoning or understanding, leading to outputs that deviate from the expected answer. For the purpose of vulnerability extraction and subsequent optimization, we focus exclusively on the third category, *model capability defects*, which are further subdivided into more fine-grained subtypes (see Section 5.4). Only failures categorized as model capability defects are used to expand and refine the vulnerability repository.

During the warm-up phase, this process initializes the repository $\mathcal{W}$, and in the main iterative cycle, it continuously updates and expands new entries. The repository is subsequently cleaned, deduplicated, and consolidated into $\mathcal{W}^*$ (Section 3.3), ensuring that only distinct, representative, and actionable vulnerabilities are retained for efficient retrieval and subsequent reuse.

3.3 Vulnerability Refinement and Repository Update

This module is responsible for maintaining and evolving the vulnerability repository $\mathcal{W}^*$ by consolidating newly discovered failures at the end of each iteration.

At the conclusion of the warm-up phase and each main iteration t, the module consolidates all newly discovered vulnerabilities from the extraction phase, $\mathcal{W}_{\text{new},t}$, with the existing repository $\mathcal{W}_{t-1}$:

$$\mathcal{W}_t = \mathcal{W}_{t-1} \cup \mathcal{W}_{\text{new},t}. \tag{6}$$

To ensure the repository remains concise and non-redundant, a refinement process is applied. Each new entry $w_{\text{new}} \in \mathcal{W}_{\text{new},t}$ is encoded into a normalized embedding vector $v_{w_{\text{new}}} = E(w_{\text{new}}.\texttt{error_summary})$. We then identify the most similar entry in the existing repository $\mathcal{W}_{t-1}$ by searching for its nearest neighbor using the L2 distance. If this minimum distance falls below a predefined threshold, $\theta = 0.1$, the new failure case is considered a duplicate and is merged with the most similar existing entry. This merge operation typically involves aggregating its `error_detail` as a new supporting example, enriching the vulnerability pattern. If no existing entry is sufficiently similar, w_{new} is added as a new, distinct vulnerability record.

This refinement process can be formally represented as:

$$\mathcal{W}_t^* = \text{Refine}(\mathcal{W}_t, \theta), \tag{7}$$

which produces a deduplicated and consolidated repository $\mathcal{W}_t^*$. This process creates a collection of unique, interpretable, and actionable vulnerability patterns, improving the efficiency of retrieval in subsequent iterations.

Through this iterative refinement, the initial, static set of expert-defined strategies $\mathcal{P}$ gradually evolves into a dynamic, data-driven repository of vulnerabilities. The updated repository $\mathcal{W}_t^*$ serves as the knowledge base for the sample generation module in the next iteration $(t+1)$, guiding the discovery of new and diverse robustness cases.

3.4 Warm-up Phase: Initializing the Vulnerability Repository

The main iterative cycle of ADVSQL relies on a dynamic vulnerability repository, $\mathcal{W}$, for targeted perturbation. To bootstrap this process, ADVSQL begins with a warm-up phase designed to establish an initial version of this repository.

The process begins with an initial set of correct samples, D_0. For each sample, the generation module (Section 3.1) is guided by a set of expert-defined, high-level perturbation strategies, denoted as $\mathcal{P}$, to create candidate perturbations. We formulated this initial set by summarizing the perturbation patterns from seminal robustness studies such as Dr.Spider, SpiderSyn, and ADVETA (Chang et al., 2023; Gan et al., 2021a; Pi et al., 2022). For each successful failure case identified during this phase, the vulnerability extraction module (Section 3.2) analyzes it and generates a structured vulnerability entry. These initial entries populate the repository $\mathcal{W}$, while their corresponding modification intentions provide the initial set of previous intentions (i_{t-1}) for the first main iteration. Following the warm-up, this initial repository undergoes the refinement process detailed in Section 3.3 to produce the first deduplicated repository, $\mathcal{W}^*$. This ensures the main cycle begins with a concise and non-redundant knowledge base, enabling the subsequent fully automated and self-evolving cycles of vulnerability discovery and refinement.

4 Experiment

4.1 Experiment Setup

Datasets In this section, we conduct experiments on *BIRD* (Li et al., 2023b) and *Spider* (Yu et al., 2018) datasets. To investigate robustness, we fine-tune an open-source LLM on a strategically curated subset of 1,000 training samples from *BIRD*, selected to ensure balanced coverage of database schemas and SQL query complexities.

Model Selection For our main experimental evaluations, we select three representative models that span distinct paradigms: a general-purpose model, Gemma-3 (Team et al., 2025) (*Gemma-3-12B-it*); a domain-adapted model, OmniSQL (Li et al., 2025) (*OmniSQL-32B*); and a reinforcement-learned model, Inf-rl-qwen (Geisler et al., 2025) (*inf-rl-qwen-coder-32B-2746*). For fine-tuning, we use Qwen2.5-Coder(*Qwen2.5-Coder-7B-Instruct*) as the base model.

Implementation Details Both the rewrite module, responsible for generating targeted perturbations, and the automated Detector, designed to validate perturbation correctness, are implemented using **Qwen3-32B**. To rigorously validate the De-

Model	Setting	BIRD			Spider		
		EX	Failure Rate	Avg. Queries	EX	Failure Rate	Avg. Queries
Gemma-3	Original	53.65	-	-	81.53	-	-
	Expert	24.64	54.07	7.73	48.36	52.16	10.95
	ADVSQL	8.34	84.45	5.08	12.38	66.32	7.85
Inf-rl-qwen	Original	70.53	-	-	87.72	-	-
	Expert	38.20	45.84	9.81	63.93	27.12	19.97
	ADVSQL	21.90	68.95	7.27	56.19	48.07	12.93
OmniSQL	Original	63.23	-	-	81.53	-	-
	Expert	30.31	52.06	7.09	48.84	40.10	10.60
	ADVSQL	14.21	77.53	5.47	31.82	60.97	8.15

Table 1: Performance on BIRD and Spider under Original, Expert (expert-designed few-shot perturbations), and Ours (ADVSQL) settings. Failure Rate and Avg. Queries are not applicable for Original.

tector's reliability, we randomly sampled 100 perturbation cases from the Gemma-3 experiments and subjected them to annotation by three independent human evaluators, with final labels determined by majority vote. The observed agreement between Qwen3-32B's classifications and the human consensus reached 89.3%, thereby supporting the Detector's efficacy. A detailed annotation protocol and full statistical analysis are presented in Appendix A. For fine-tuning experiments on *Qwen2.5-Coder-7B-Instruct*, we employ LoRA-based parameter-efficient tuning for both SFT and DPO. All fine-tuning and evaluation code is based on the `llama-factory`[1] framework. The key hyperparameters are provided in Appendix Table 5.

Metrics We evaluate model performance using three complementary metrics:

- **Execution Accuracy (EX)**: Measures the proportion of generated SQL queries whose execution results exactly match the ground truth.
- **Failure Rate**: Captures robustness against targeted perturbations generated by our framework. It is defined as the proportion of validly perturbed samples for which the model produces incorrect execution results, directly reflecting the discovery rate of A-level failure cases.
- **Average Queries (Avg. Queries)**: This metric evaluates the efficiency of the perturbation process. It is calculated as the ratio of the total number of queries made to the number of successfully perturbed questions.

These metrics together provide a comprehensive evaluation of model performance across correctness, robustness, and efficiency.

[1] https://github.com/hiyouga/LLaMA-Factory

4.2 Main Results

Table 1 reports the performance of each target model on the BIRD and Spider datasets across three distinct evaluation settings. While all models show competent baseline performance in the *Original* setting, their accuracy declines markedly under perturbation. In our proposed setting, the EX decreases substantially, while the Failure Rate rises sharply, often exceeding 60%. This result highlights the high sensitivity of current Text-to-SQL models to targeted perturbations. Across all models, Qwen-Coder-32B variants (Inf-rl-qwen and OmniSQL) consistently demonstrate greater robustness than the general-purpose Gemma-3. Furthermore, while the *Expert* setting proves effective, our automated method achieves a more substantial drop in EX and a higher Failure Rate. Crucially, our method also demonstrates higher efficiency, requiring significantly fewer queries to induce failures, as shown by the Avg. Queries metric.

To further understand why our method induces stronger perturbation effects, we examine the dimensions that triggered model failures in the *Expert* and *Ours* settings (see Appendix D for details). The expert-guided setting reveals vulnerabilities primarily in query text perturbations (63.5%), with substantially fewer errors triggered by schema-level modifications. In contrast, our automated framework uncovers a more diverse and balanced distribution of vulnerabilities across multiple dimensions. This demonstrates that our method mitigates the biases of expert-driven design and systematically explores a broader spectrum of model vulner-

Method	Strategy	EX	Failure Rate	Avg. Queries
ADVSQL	Q + IS + RS	8.34	84.45	5.08
w/o vulnerability Lib	Q + IS + RS	24.64	54.00	7.73
w Only Q	Only Q	14.08	73.75	5.54
w Only IS	Only IS	27.84	48.12	10.84
w Only RS	Only RS	26.21	51.15	8.90

Table 2: Ablation study results on the BIRD dataset using Gemma-3

abilities at both semantic and structural levels.

4.3 Ablation Studies

To assess the contribution of each component in our framework, we conduct ablation studies on the BIRD dataset using Gemma-3 as the target model, with results reported in Table 2. We first examine the role of the dynamic vulnerability repository. Without this repository, performance degradation is relatively limited, whereas ADVSQL leverages it to reduce EX to 8.34% and increase the Failure Rate to 84.45%. This demonstrates that the repository is critical for systematically identifying and exploiting model vulnerabilities. Furthermore, this mechanism enhances efficiency, reducing the average number of queries required to induce a failure from 7.73 to 5.08.

We then decompose the composite perturbation strategy to analyze the individual impact of its three angles: queries (Q), relevant schema elements (RS), and irrelevant schema elements (IS). Our results confirm the superiority of the combined strategy (Q+IS+RS). While perturbations along any single angle cause a performance decline, their combined effect proves more potent, leveraging multiple perturbation angles to uncover deeper structural vulnerabilities. Collectively, these ablations affirm that the synergy between our dynamic vulnerability repository and the multi-angle perturbation strategy is essential for achieving effectiveness in vulnerability discovery.

5 Analysis

5.1 Transfer Study of the Vulnerability Repository

To assess cross-model transferability of our vulnerability repository, we conducted a single-iteration strategy-transfer experiment on the BIRD dataset, applying strategies distilled from each of the three models to every other model. The results, summarized in Figure 3, show a consistent cross-model effectiveness pattern. For instance, strategies originating from OmniSQL yield a 70.96% Failure Rate when applied to Gemma-3, which is slightly below

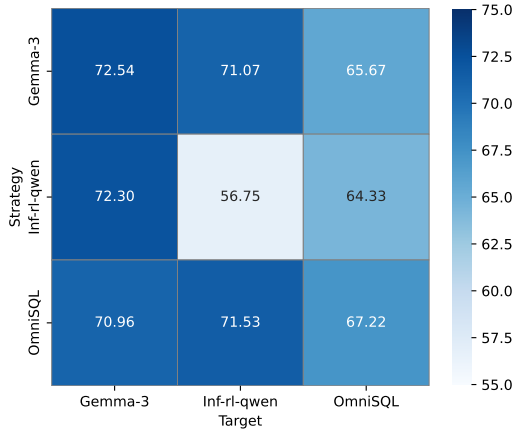

Figure 3: Cross-Model Strategy Transfer Performance

Gemma-3's own baseline Failure Rate of 72.54%. Notably, the transfer effect can be asymmetric, exposing vulnerabilities that are not easily discovered via self-testing. Inf-rl-qwen, which has the lowest baseline Failure Rate of 56.75%, experiences a substantial increase in Failure Rate to 71.07% and 71.53% when tested against strategies from Gemma-3 and OmniSQL, respectively. This finding suggests that strategies derived from other models can uncover a broader and more challenging set of vulnerabilities. This strong cross-model performance indicates that our framework captures fundamental, model-agnostic failure patterns rather than superficial, model-specific quirks. It thereby demonstrates the potential of our method to build a general and reusable repository of vulnerabilities applicable to a wide range of models.

5.2 Robustness Enhancement via Fine-Tuning Strategies

To investigate the efficacy of different fine-tuning strategies for improving robustness in Text-to-SQL tasks, we sampled 1,000 instances from the BIRD training set and used our framework to synthesize a robustness-focused dataset to fine-tune the *Qwen2.5-Coder-7B-Instruct* model using both SFT and DPO. For SFT, we used the perturbed samples and their corresponding correct SQL queries as the fine-tuning data. For DPO, the perturbed sample served as the prompt, with the correct SQL as the positive example and the model's incorrect output as the negative example. As shown in Table 3, SFT yields consistent gains across both the original distribution and robustness-oriented metrics. Specifically, Org. EX improves substantially from 42.63% to 52.87%, while robustness EX more than doubles (from 4.37% to 9.58%). By contrast, DPO

Strategy	Org. EX	EX	Failure Rate	Avg. Queries
Vanilla	42.63	4.37	89.76	3.61
SFT	52.87	9.58	81.87	4.59
DPO	46.41	2.93	93.68	3.24

Table 3: Effect of SFT and DPO on robustness in Text-to-SQL tasks. "Org. EX" denotes the Execution Accuracy on the original BIRD dataset.

Model	EX	Failure Rate	Avg. Queries
Gemma-3	14.73	72.54	5.68
GPT-4o	22.36	65.80	5.35

Table 4: Scalability Evaluation on GPT-4o.

achieves only a modest increase in Org. EX (from 42.63% to 46.41%) while degrading other robustness metrics. We attribute this behavior to DPO's sensitivity to noisy or imperfect target responses in the synthesized dataset, which can induce misrankings and hinder learning. These results indicate that synthesized robustness examples can improve performance on the original BIRD task; however, straightforward SFT is currently more reliable than DPO under our synthesis pipeline.

5.3 Scalability Evaluation on GPT-4o

To further assess the scalability of our approach, we evaluated its effectiveness on the larger and more capable GPT-4o model using the same error strategies distilled from Gemma-3 on the BIRD dataset. Due to the substantial cost of running GPT-4o, we performed a single-iteration transfer experiment only. As presented in Table 4, despite the limited experimental setting, the transferred strategies produce a substantial 65.80% Failure Rate on GPT-4o, indicating that our framework generalizes effectively to larger and more advanced LLMs, highlighting the framework's robustness and generalizability across model scales.

5.4 Error Analysis

Figure 4 shows the distribution of model errors across seven predefined categories (definitions in Appendix E). *Question Interpretation* is the most frequent error, accounting for 30–34% across models, indicating that capturing user intent remains a primary bottleneck. *Column Semantics* contributes roughly 25% of errors, reflecting shared difficulties in precise column-level mapping. Gemma-3 exhibits a notably higher Failure Rate in *Table Relationship* (16.1%, compared to 11–12% for other models), indicating that it struggles more with reasoning over complex relational structures and modeling cross-table dependencies.

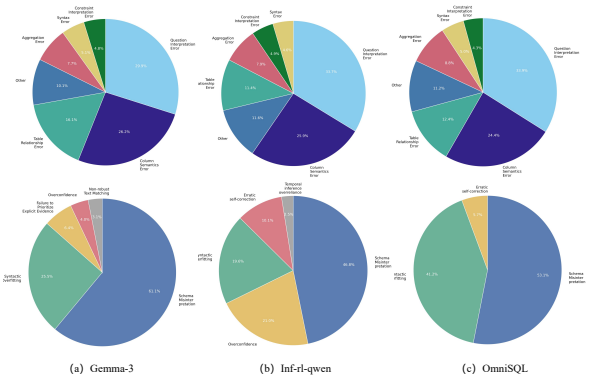

Figure 4: Error distribution across models. The top panel shows the distribution of predefined error categories for Gemma-3, Inf-rl, and OmniSQL, while the bottom panel provides a fine-grained breakdown of the errors originally grouped under "Others".

Beyond the established categories, Figure 4 further presents a detailed analysis of errors typically grouped under the *OTHER* category. By combining expert manual inspection with systematic reclassification, we uncover recurring and nuanced cognitive failure patterns. These include: schema misinterpretation, syntactic overfitting, insufficient prioritization of explicit evidence, overconfidence, non-robust text matching, erratic self-correction, and overreliance on temporal inference. This trigger-based partitioning moves beyond surface-level heuristics, substantially enriching standard error categories and, more importantly, providing fine-grained, actionable insights for targeted model improvement. Comprehensive examples and the full breakdown of this refined analysis are given in Appendix E.

6 Conclusion

In this paper, we propose ADVSQL, an automated framework that systematically discovers and exploits vulnerabilities in Text-to-SQL models. Our core approach utilizes a closed-loop mechanism that iteratively perturbs queries and database schemas, automatically distilling the resulting failures into reusable strategies to uncover progressively deeper vulnerabilities. We conduct extensive experiments on the BIRD and Spider benchmarks, demonstrating that ADVSQL effectively reveals critical failure modes in advanced Text-to-SQL models and generates targeted data that substantially enhances their robustness.

Limitations

In this work, we propose ADVSQL, a novel framework for the systematic discovery and exploitation of vulnerabilities in Text-to-SQL models. However, there are some limitations to our work, which we discuss below. ADVSQL's iterative, closed-loop methodology necessitates substantially greater computational resources compared to static benchmarks, involving numerous inference runs of both the target and auxiliary LLMs. However, we argue this additional cost is warranted. The dynamic and adaptive nature of our framework offers superior diagnostic value by revealing the subtle, model-specific failure patterns essential for effective model hardening, which static evaluation methods cannot uncover.

References

Adithya Bhaskar, Tushar Tomar, Ashutosh Sathe, and Sunita Sarawagi. 2023. Benchmarking and improving text-to-sql generation under ambiguity. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7053–7074.

Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, and 1 others. 2023. Dr. spider: A diagnostic evaluation benchmark towards text-to-sql robustness. *arXiv preprint arXiv:2301.08881*.

Song Cheng, Qiannan Cheng, Linbo Jin, Lei Yi, and Guannan Zhang. 2025. Sqlord: A robust enterprise text-to-sql solution via reverse data generation and workflow decomposition. In *Companion Proceedings of the ACM on Web Conference 2025*, pages 919–923.

DongHyun Choi, Myeong Cheol Shin, EungGyun Kim, and Dong Ryeol Shin. 2021. Ryansql: Recursively applying sketch-based slot fillings for complex text-to-sql in cross-domain databases. *Computational Linguistics*, 47(2):309–332.

Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. Reforce: A text-to-sql agent with self-refinement, format restriction, and column exploration. In *ICLR 2025 Workshop: VerifAI: AI Verification in the Wild*.

Xiang Deng, Ahmed Hassan, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2021. Structure-grounded pretraining for text-to-sql. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1337–1350.

Sen Fang, Weiyuan Ding, and Bowen Xu. 2025. Evaloop: Assessing llm robustness in programming from a self-consistency perspective. *arXiv preprint arXiv:2505.12185*.

Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R Woodward, Jinxia Xie, and Pengsheng Huang. 2021a. Towards robustness of text-to-sql models against synonym substitution. In *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)*, pages 2505–2515.

Yujian Gan, Xinyun Chen, and Matthew Purver. 2021b. Exploring underexplored limitations of cross-domain text-to-sql generalization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8926–8931.

Chang Gao, Bowen Li, Wenxuan Zhang, Wai Lam, Binhua Li, Fei Huang, Luo Si, and Yongbin Li. 2022. Towards generalizable and robust text-to-sql parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 2113–2125.

Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, and 1 others. 2024. A preview of xiyan-sql: A multi-generator ensemble framework for text-to-sql. *arXiv preprint arXiv:2411.08599*.

Simon Geisler, Tom Wollschläger, MHI Abdalla, Vincent Cohen-Addad, Johannes Gasteiger, and Stephan Günnemann. 2025. Reinforce adversarial attacks on large language models: An adaptive, distributional, and semantic objective. *arXiv preprint arXiv:2502.17254*.

Yu Guo, Dong Jin, Shenghao Ye, Shuangwu Chen, Jianyang Jianyang, and Xiaobin Tan. 2025. SQL-Forge: Synthesizing reliable and diverse data to enhance text-to-SQL reasoning in LLMs. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 8441–8452, Vienna, Austria. Association for Computational Linguistics.

Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2024. Next-generation database interfaces: A survey of llm-based text-to-sql. *arXiv preprint arXiv:2406.08426*.

Yiming Huang, Jiyu Guo, Wenxin Mao, Cuiyun Gao, Peiyi Han, Chuanyi Liu, and Qing Ling. 2025. Exploring the landscape of text-to-sql with large language models: Progresses, challenges and opportunities. *arXiv preprint arXiv:2505.23838*.

Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. 2021. Kaggledbqa: Realistic evaluation of text-to-sql parsers. In *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)*, pages 2261–2273.

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, and 1 others. 2025. Omnisql: Synthesizing high-quality text-to-sql data at scale. *arXiv preprint arXiv:2503.02240*.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023a. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 13067–13075.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. Codes: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data*, 2(3):1–28.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2023b. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357.

Hanbing Liu, Haoyang Li, Xiaokang Zhang, Ruotong Chen, Haiyong Xu, Tian Tian, Qi Qi, and Jing Zhang. 2025a. Uncovering the impact of chain-of-thought reasoning for direct preference optimization: Lessons from text-to-sql. *arXiv preprint arXiv:2502.11656*.

Xiaogeng Liu, Peiran Li, Edward Suh, Yevgeniy Vorobeychik, Zhuoqing Mao, Somesh Jha, Patrick McDaniel, Huan Sun, Bo Li, and Chaowei Xiao. 2024. Autodan-turbo: A lifelong agent for strategy self-exploration to jailbreak llms. *arXiv preprint arXiv:2410.05295*.

Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and 1 others. 2025b. Xiyan-sql: A novel multi-generator framework for text-to-sql. *arXiv preprint arXiv:2507.04701*.

Qin Lyu, Kaushik Chakrabarti, Shobhit Hathi, Souvik Kundu, Jianwen Zhang, and Zheng Chen. 2020. Hybrid ranking network for text-to-sql. *arXiv preprint arXiv:2008.04759*.

Tanzim Mahmud, KM Azharul Hasan, Mahtab Ahmed, and Thwoi Hla Ching Chak. 2015. A rule based approach for nlp based query processing. In *2015 2nd international conference on electrical information and communication technologies (EICT)*, pages 78–82. IEEE.

Rajaswa Patil, Manasi Patwardhan, Shirish Karande, Lovekesh Vig, and Gautam Shroff. 2023. Exploring dimensions of generalizability and few-shot transfer for text-to-sql semantic parsing. In *Transfer Learning for Natural Language Processing Workshop*, pages 103–114. PMLR.

Anselm Paulus, Arman Zharmagambetov, Chuan Guo, Brandon Amos, and Yuandong Tian. 2024. Advprompter: Fast adaptive adversarial prompting for llms. *arXiv preprint arXiv:2404.16873*.

Xinyu Pi, Bing Wang, Yan Gao, Jiaqi Guo, Zhoujun Li, and Jian-Guang Lou. 2022. Towards robustness of text-to-sql models against natural and realistic adversarial table perturbation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2007–2022.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems*, 36:36339–36348.

Ge Qu, Jinyang Li, Bowen Qin, Xiaolong Li, Nan Huo, Chenhao Ma, and Reynold Cheng. 2025. SHARE: An SLM-based hierarchical action CorREction assistant for text-to-SQL. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11268–11292, Vienna, Austria. Association for Computational Linguistics.

Ariana Sahitaj, Markus Nilles, Ralf Schenkel, and Vera Schmitt. 2025. Utilising large language models for adversarial attacks in text-to-sql: A perpetrator and victim approach. In *Datenbanksysteme für Business, Technologie und Web (BTW 2025)*, pages 919–931. Gesellschaft für Informatik, Bonn.

Irina Saparina and Mirella Lapata. 2024. Ambrosia: A benchmark for parsing ambiguous questions into database queries. *Advances in Neural Information Processing Systems*, 37:90600–90628.

Lei Sheng, Shuai-Shuai Xu, and Wei Xie. 2025. Basesql: A powerful open source text-to-sql baseline approach. *arXiv preprint arXiv:2502.10739*.

Liang Shi, Zhengju Tang, Nan Zhang, Xiaotong Zhang, and Zhi Yang. 2025. A survey on employing large language models for text-to-sql tasks. *ACM Comput. Surv.* Just Accepted.

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, and 1 others. 2025. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*.

Hetvi Waghela, Jaydip Sen, Sneha Rakshit, and Subhasis Dasgupta. 2024. Adversarial text generation with dynamic contextual perturbation. In *2024 IEEE Calcutta Conference (CALCON)*, pages 1–6. IEEE.

Hanchen Xia, Feng Jiang, Naihao Deng, Cunxiang Wang, Guojiang Zhao, Rada Mihalcea, and Yue Zhang. 2024. $r\hat{}3$:" this is my sql, are you with me?" a consensus-based multi-agent system for text-to-sql tasks. *arXiv preprint arXiv:2402.14851*.

Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment. *Preprint*, arXiv:2502.14913.

Bo Xu, Shufei Li, Yifei Wu, Shouang Wei, Ming Du, Hongya Wang, and Hui Song. 2024. Chain-of-program prompting with open-source large language models for text-to-sql. In *2024 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. Tabert: Pretraining for joint understanding of textual and tabular data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8413–8426.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, and 1 others. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018*, pages 3911–3921. Association for Computational Linguistics.

Bin Zhang, Yuxiao Ye, Guoqing Du, Xiaoru Hu, Zhishuai Li, Sun Yang, Chi Harold Liu, Rui Zhao, Ziyue Li, and Hangyu Mao. 2024. Benchmarking the text-to-sql capability of large language models: A comprehensive evaluation. *arXiv preprint arXiv:2403.02951*.

Yangshijie Zhang. 2025. Emoti-attack: Zero-perturbation adversarial attacks on nlp systems via emoji sequences. *arXiv preprint arXiv:2502.17392*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

A Human evaluation of the Detector

To verify the reliability of the automated detector (implemented with Qwen3-32B), we conducted a targeted human evaluation. We recruited three graduate-level annotators with expertise in Text-to-SQL tasks to ensure the quality of the assessment. From the Gemma-3 experiments, we randomly sampled 100 perturbed cases for evaluation. Each annotator independently judged whether each perturbed query was valid (i.e., the perturbation preserves semantic equivalence and the correct answer should remain unchanged) or invalid (i.e., the perturbation alters the meaning or introduces inconsistencies).

Each annotator was provided with (i) the original natural-language query and its gold answer, (ii) the perturbed query, and (iii) the corresponding M-schema context. They followed a concise guideline: (1) assess semantic equivalence between the original and perturbed queries, (2) verify whether the perturbation is realistic and contextually appropriate, and (3) label the sample as either Valid or Invalid. The final ground-truth label for each case was determined by majority vote across the three annotators to reduce subjectivity and ensure consistency.

We then compared the detector's predictions with these human consensus labels. The detector achieved an average accuracy of 89.3%, indicating a high level of agreement with expert human judgments in identifying valid perturbations.

B Detailed Algorithm for One Iteration

The complete Algorithm **??** summarizes one iteration of the main automated cycle.

C Training configurations and key hyperparameters

This section lists the key hyperparameters for the parameter-efficient fine-tuning experiments on *Qwen2.5-Coder-7B-Instruct*. We run two tuning protocols: supervised fine-tuning (SFT) and direct preference optimization (DPO). Table 5 summarizes the most influential training choices for SFT and DPO.

Parameter	SFT / DPO
Per-device train batch size	1
Gradient accumulation steps	8
Effective learning rate	1.0×10^{-4}
Number of training epochs	3
LR scheduler	cosine (warmup ratio = 0.1)
Precision	bf16
DPO-specific params	SFT: — DPO: pref_beta = 0.1, pref_loss = sigmoid
Max dataset samples	8000
Cutoff length (tokens)	10240
Preprocessing workers	16
Dataloader workers	4
Optimization details	lr_scheduler = cosine, warmup = 10%
Speed-up flags	enable_liger_kernel = True

Table 5: Key training hyperparameters (SFT vs DPO).

Algorithm 1 One Iteration of the Automated Vulnerability Discovery and Refinement Cycle (Iteration t)

Input: A dataset of correct samples $D = \{(Q, DB, SQL^\star)\}$; Refined vulnerability repository from the previous iteration $\mathcal{W}^*_{t-1}$; Set of modification intentions from the previous iteration I_{t-1}; A set of perturbation angles $\mathbb{A}$; All necessary LLM modules (Generator, Target, Detector, AnswerChecker, Summarizer, Refine); Distance threshold for refinement θ.
Initialize: An empty set for newly discovered vulnerabilities in this iteration: $\mathcal{W}_{\text{new},t} \leftarrow \emptyset$.
for each case $(Q, DB, SQL^\star)$ sampled from dataset D **do**
  Select a guiding modification intention i_{t-1} from I_{t-1}.
  Retrieve top-k relevant vulnerabilities as guidance from the repository: $\Gamma_t \leftarrow \text{Retrieve}(\mathcal{W}^*_{t-1}, i_{t-1})$.
  for each perturbation angle $a \in \mathbb{A}$ **do**
    Generate a perturbed sample and a new modification intention i_t:

$$\{(\tilde{Q}, \tilde{DB}), i_t\} \leftarrow \text{Generator}(Q, DB, \Gamma_t, a)$$

    Obtain the target model's predicted SQL query: $\tilde{SQL} \leftarrow \text{Target}(\tilde{Q}, \tilde{DB})$.
    ▷ Perform two-stage validation to identify A-level failures.
    Step 1: Certify if the perturbation is answer-preserving (semantically equivalent): is_valid $\leftarrow$ Detector$((Q, DB), (\tilde{Q}, \tilde{DB}))$
    Step 2: Check the correctness of the model's prediction via execution: is_correct $\leftarrow$ AnswerChecker$(\tilde{SQL}, \tilde{DB}, SQL^\star, DB)$
    if is_valid $=$ Valid $\wedge$ is_correct $=$ Incorrect **then**
      Extract a structured vulnerability entry: $w \leftarrow \text{Summarizer}((Q, DB), (\tilde{Q}, \tilde{DB}), \tilde{SQL}, \boldsymbol{i_t})$
      Add the newly found vulnerability to the temporary set: $\mathcal{W}_{\text{new},t} \leftarrow \mathcal{W}_{\text{new},t} \cup \{w\}$.
    end if
  end for
end for
▷ At the end of the iteration, consolidate and refine the repository.
Consolidate: Merge newly discovered vulnerabilities with the existing repository: $\mathcal{W}_t \leftarrow \mathcal{W}^*_{t-1} \cup \mathcal{W}_{\text{new},t}$.
Refine: Deduplicate and consolidate the merged repository: $\mathcal{W}^*_t \leftarrow \text{Refine}(\mathcal{W}_t, \theta)$.
Output: The updated and refined vulnerability repository $\mathcal{W}^*_t$ (to be used in the next iteration $t + 1$).

D Detailed Analysis of Failure-Triggering Dimensions

To supplement the main results, Figure 5 provides a detailed comparative analysis of the failure-triggering dimensions for the *Expert* setting versus the first two iterations of our automated framework.

E Error Analysis

The table 6 presents the seven predefined error categories. Furthermore, we offer a detailed analysis of the errors under the *OTHER* category. Through manual inspection and reclassification, we summarize seven main types of errors: schema misinterpretation, syntactic overfitting, insufficient prioritization of explicit evidence, overconfidence, non-robust text matching, erratic self-correction, and overreliance on temporal inference. The table 7 summarizes the error-triggering mechanisms we discovered, along with representative examples for each category.

- **Schema misinterpretation** The model infers table or column semantics from surface cues or training priors rather than the actual schema. This leads to wrong joins, nonexistent column references, or incorrect grouping dimensions. The SQL may still be syntactically valid but semantically flawed.
- **Syntactic overfitting** The model overuses frequent SQL templates or complex constructs such as nested subqueries, unnecessary `DISTINCT`, and redundant aggregations. As a result, it often produces SQL that is overly complex and fragile, even when simpler expressions would be correct.
- **Failure to prioritize explicit evidence** The model overlooks literal constraints such as numbers, IDs, or dates when queries also contain am-

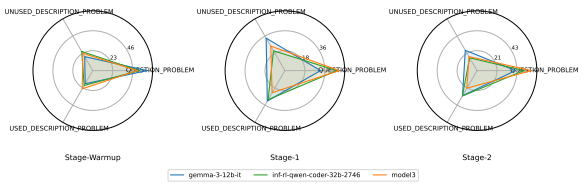

Figure 5: Comparison of the distribution of vulnerabilities identified by expert experience and our methods across failure-triggering dimensions. QUESTION_PROBLEM, USED_DESCRIPTION_PROBLEM, and UNUSED_DESCRIPTION_PROBLEM, which denote perturbation patterns applied to queries, relevant schema elements, and irrelevant schema elements, respectively.

Error Type	Explanation
Syntax Error	Incorrect SQL syntax, formatting issues, or function usage errors
Question Interpretation Error	Misunderstanding of the user's question intent
Table Relationship Error	Incorrect inference of relationships between tables
Constraint Interpretation Error	Misinterpretation of user-defined or schema-level constraints
Column Semantics Error	Incorrect understanding of column meaning or values
Aggregation Error	Incorrect aggregation or grouping of data
Other	Other types of errors not covered above

Table 6: Description of error distributions

biguous natural-language phrasing. It favors inferred or contextual interpretations over explicit evidence, which causes user-specified conditions to be ignored.

- **Overconfidence** The model outputs SQL with high certainty even when the evidence is weak. It does not perform self-checks or validation and fails to signal uncertainty or abstain. As a result, logically incorrect queries are often asserted as correct.
- **Non-robust text matching** The model relies on strict string matching, such as case-sensitive equality or exact tokens. It does not apply robust strategies like case normalization, substring search, or fuzzy matching. This leads to missed records when handling noisy text data.
- **Erratic self-correction** The model reacts excessively during self-reflection, often producing multiple conflicting SQL variants in response to minor input changes. These revisions are not grounded in schema structure or consistent reasoning. As a result, outputs become unstable and may worsen compared to the initial answer.
- **Temporal inference overreliance** The model depends on prior temporal heuristics such as month lengths or typical date ranges. It does not always verify against explicit input evidence. This produces logically plausible but factually inconsistent temporal reasoning.

F Prompt Templates for Perturbation and Validation

We provide the prompt templates used in our framework. Figures 6, Figures 7, and Figures 8 illustrate the prompt templates for perturbations applied to natural language queries, schema-relevant elements, and schema-irrelevant elements, respectively.

Error Type	Example
Schema misinterpretation	**Task**: Query "the names of teachers who teach Mathematics and the number of students they have." **Model output**: `SELECT t.name, COUNT(s.student_id) FROM student_info s JOIN teacher_records t ON s.student_id = t.teacher_id WHERE t.course = 'Mathematics' GROUP BY t.name;` **Explanation**: The model infers a join from surface-similar ID fields (student_id = teacher_id) instead of following the true foreign-key chain (e.g., student → course → teacher), producing a semantically incorrect join.
Syntactic overfitting	**Task**: Query "customers with order amounts greater than 1000 in Q1 2023." **Model output**: `SELECT DISTINCT customer_id FROM ( SELECT o.customer_id, o.amount FROM orders o WHERE o.amount > 1000 ) sub WHERE sub.customer_id IS NOT NULL;` **Explanation**: The model uses unnecessary nesting and `DISTINCT` where a simple `SELECT ... WHERE amount > 1000` would suffice, demonstrating over-reliance on complex SQL templates instead of a simpler expression.
Failure to prioritize explicit evidence	**Task**: Query "students with scores above 85." **Model output**: `SELECT name FROM students WHERE grade >= 70;` or `SELECT name FROM students WHERE grade = 'Excellent';` **Explanation**: The model ignores the literal threshold '85' and substitutes default thresholds or categorical labels, failing to prioritize the explicit numeric evidence provided by the user.
Overconfidence	**Task**: "Count students enrolled in course X." **Model output**: `SELECT COUNT(*) FROM students;` — "The result is 200." **Explanation**: The produced SQL omits the necessary course filter or joins, but the model nonetheless asserts a concrete result. This demonstrates confident assertions without validation or uncertainty signaling.
Non-robust text matching	**Task**: Match course name variants like 'Mathematics', 'mathematics', 'Math'. **Model output**: `SELECT * FROM courses WHERE course_name = 'Mathematics';` **Explanation**: Strict equality matching is brittle to case differences and abbreviations; robust strategies (e.g., `LOWER()`, `ILIKE`, or substring matching) are needed to handle noisy or variant text.
Erratic self-correction	**Behavior example**: the model produces conflicting revisions after self-reflection. **Initial**: `SELECT t.name, COUNT(s.id) FROM students s JOIN teachers t ON s.teacher_id = t.id GROUP BY t.name;` **Revised**: `SELECT teacher_name, COUNT(student_no) FROM teacher_records tr JOIN student_info si ON tr.teacher_id = si.teacher_no GROUP BY teacher_name;` **Explanation**: The model's correction alters column names, aliases, and join conditions inconsistently, producing multiple contradictory SQL variants that are not grounded in the schema, reflecting unstable self-correction.
Temporal inference over-reliance	**Task**: User specifies "until 2023-08-30." **Model output**: `SELECT * FROM orders WHERE order_date <= '2023-08-31' AND amount > 0;` **Explanation**: The model applies a heuristic (+1 day) instead of using the explicit boundary. Correct behavior should respect the literal date (e.g., `order_date <= '2023-08-30'` for a date field) or choose a semantically appropriate half-open interval for timestamps, based on the column type.

Table 7: Representative Error Examples in LLM SQL Predictions.

The prompt templates for perturbations applied to natural language queries

Task:
Please **rewrite the question and the evidence** in a way that **preserves the original intent and does not change the final correct answer** (i.e., the gold SQL query must still be valid and applicable). Your rewritten version should:
- Better reflect **real-world use cases or natural user phrasing**.
- Increase **linguistic and structural diversity** ...
- Introduce **more contextual framing, ambiguity, or reasoning cues** ...
- Remain **faithful to the original semantics and answer**.

Guidelines:
- Do **not** change the final answer (i.e., gold SQL must still be valid).
- Focus on **rewriting**, not expanding or omitting information.
- You must **describe your modification strategy** clearly in the ‘improvement‘ field.

Output Format:
Please return your output strictly in the following JSON format:
“‘json
{{
“improvement": “Describe what changes you made and why ...",
“question": “your rewritten question",
“evidence": “your rewritten evidence"
}}
Example 1-3: ...

Now it is your turn:
Below is the database schema:
{schema}
Question: {question}
Evidence: {evidence}
Now it is your turn:
Below is the database schema:
{schema}
Question: {question}
Evidence: {evidence}
The gold SQL query is: {gold_sql}

Figure 6: The prompt templates for perturbations applied to natural language queries.

The prompt templates for perturbations applied to schema-irrelevant elements

Task:
Your task is to **modify the provided database-related information** while ensuring that the **core semantics and the correct answer remain unchanged**.
Your goal is to introduce **maximum lexical, structural, and semantic diversity** in order to **evaluate the model’s robustness** against variations in metadata.
You are allowed to revise the following fields for each column:
- **column_description**: Modify the column description using paraphrasing, synonyms, different sentence structures ...
- **value_description**: Change the way value types, ranges, or categories are described ...
- **example_value**: Add more diverse, realistic, or representative sample values ...

Important Guidelines:
- **Do not** change the table names or column names.
- **Do not** alter the underlying meaning or change the final answer.
- You must output your result in the **strict JSON format** shown below.
- You must change the metadata for **at least 3 columns** and **at most 5 columns**.

Output Format:

“table_name": [
{{
“column": “column_name",
“improvement": “Describe your strategy for modifying this column’s metadata.",
“column_description": “Modified description of the column.",
“value_description": “Modified description of the values.",
“example_value": [“example1", “example2"]
}}

Example 1-3: ...

Below is a partial view of the database schema. It includes only the tables and columns that are directly relevant to answering the given question; all unrelated schema elements have been omitted.
{json_str}
Question: {question}
Evidence: {evidence}
The gold SQL query is: {gold_sql}
Here are some successful attack strategies you can refer to: {error_strategies}
#The column_description and value_description fields should each be no longer than 100 tokens.
Now it’s your turn. Please return your answer strictly in the above JSON format.

Figure 7: The prompt templates for perturbations applied to schema-irrelevant elements.

The prompt templates for perturbations applied to schema-relevant elements

You are an expert in prompt-based robustness evaluation for database-related tasks.
Task:
Please **modify the database-related information** while ensuring that the **core semantics and the correct answer remain unchanged**. Your goal is to introduce **greater lexical, structural, and semantic diversity** to challenge the robustness of the model.
You may revise the following elements for each column:
- **column_description**: Modify the description using paraphrasing, synonyms, different sentence structures, or multilingual expressions.
- **value_description**: Alter the value type description or re-express the value mappings, ranges, or classifications.
- **example_value**: Enrich the example values with varied, realistic, and semantically appropriate entries.

Important:
- Do **not** change the table name or column name.
- Your output must be in **strict JSON format** as shown below.
- You must change the metadata for **at least 3 columns** and **at most 5 columns**.

Here are some output Examples:
Example 1-3: ...
Now it is your turn to summarize
Below is a partial view of the database schema. It includes only the tables and columns that are relevant to answering the given question; unrelated schema elements have been omitted.
{json_str}
Question: {question}
Evidence: {evidence}
The gold SQL query is: {gold_sql}

Return your modifications using the following structure:

{{
“table_name": [
{{
“column": “column_name",
“improvement": “Describe your strategy for modifying this column’s metadata.",
“column_description": “Modified description of the column.",
“value_description": “Modified description of the values.",
“example_value": [“example1", “example2"]
}}
]
}}

Here are some successful attack strategies you can refer to.{error_strategies}
The column_description and value_description fields should each be no longer than 100 tokens.
Now it’s your turn. Please return your answer strictly in the above JSON format.

Figure 8: The prompt templates for perturbations applied to schema-relevant elements.