Developing and Benchmarking Verification Algorithms to Improve Text-to-SQL Generation

Tarfah Alrashed
Google
tarfah@google.com

Madhup Sukoon
Google
madhups@google.com

David R. Karger
Google
dkarger@google.com

Natasha Noy
Google
noy@google.com

ABSTRACT

Large Language Models (LLMs) have achieved impressive performance in translating natural language queries into executable SQL. However, these systems remain prone to *deceptive failures*: generating syntactically valid SQL that executes but fails to capture the user's intent. In this work, we argue that further progress can be made by focusing specifically on *verification*. In this paper, we formalize Text-to-SQL verification as a standalone task: assessing whether a candidate SQL query semantically satisfies a natural language request, without access to ground-truth labels. We propose and compare two modular verification strategies: *Round-Trip Critique*, which reverse-translates SQL into natural language to detect semantic drift, and *Synthetic Execution Consistency*, which uses unit-test-like synthetic inputs to ground verification in execution results. Our evaluation shows that these methods provide a robust signal for identifying incorrect queries, successfully flagging 64% of errors from a state-of-the-art generator and outperforming standard error detectors. We demonstrate two critical applications: (1) auditing foundational benchmarks (Spider, BIRD and KaggleDBQA), revealing that over two-thirds of "generator failures" actually stem from flawed benchmark labels, and (2) enabling Selective Generation, where a system uses verification signals to abstain from answering when confidence is low. Our results show that this paradigm significantly improves the quality of deployed data interfaces by transforming silent failures into explicit abstentions that alert the user to the failure and give them a chance to correct it.

PVLDB Reference Format:
Tarfah Alrashed, Madhup Sukoon, David R. Karger, and Natasha Noy. Developing and Benchmarking Verification Algorithms to Improve Text-to-SQL Generation. PVLDB, 19(11): 3732 - 3744, 2026.
doi:10.14778/3836663.3836721

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://figshare.com/s/c88a751bc29d9751816b.

1 INTRODUCTION

Researchers have made substantial progress in leveraging Large Language Models (LLMs) to translate natural language queries (NLQs) into executable SQL. This Text-to-SQL capability is foundational to providing intuitive, code-free access to structured data, making it an increasingly vital component of business intelligence, knowledge exploration, and automated data analysis, especially for users who do not know SQL [5, 12, 13, 23].

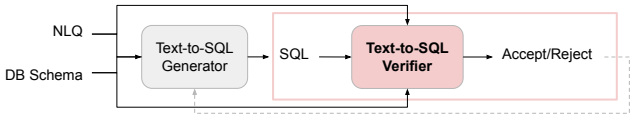

Figure 1: Text-to-SQL maps natural language question to SQL; our focus is the verifier that accepts or rejects queries.

In this work, we show how NL to SQL *generation* can be enriched by a study of NL to SQL *verification*. While generation takes an NLQ and asks for a matching SQL, verification takes both an NLQ and candidate SQL translation and determines whether they match (see Figure 1). Generators are implicitly "verifying" their generated SQL in the sense that they want it to be correct, but pulling out that verification step as a separate "module" lets us develop new techniques that can be used to improve those generators. Crucially, this modular verifier utilizes a separate, "fresh" LLM instance entirely isolated from the generator. By stripping away the original generation context, the verifier acts as an independent judge, breaking the confirmation bias that often causes a single model to overlook its own reasoning flaws. A verifier can also be used to detect and remove errors in the benchmarks used to assess generators and in the synthetic data used to train them.

In this paper, we formalize LLM-based Text-to-SQL verification and propose techniques to address it. Moving beyond the standard "critique" model [2, 5, 16, 27, 28, 36]—which prompts the LLM to verify and correct its own generated SQL—we introduce two verification strategies. First, adapted from round-trip correctness in code generation [1], we propose *round-trip critique*, which translates the candidate SQL back to natural language and asks an LLM to assess whether the resulting NLQ preserves the original intent. Second, we introduce *synthetic execution*, which adapts offline test-database generation and semantic evaluation principles [11, 31, 42] into an online, inference-time verification mechanism. Like unit-testing, this approach dynamically synthesizes a small schema-consistent database and compares NLQ and SQL results on it to assess whether the generated SQL executes and yields the correct answer.

As a first application, we show that our verification algorithms detect numerous errors in the ground truth "gold" SQL provided in popular Text-to-SQL benchmarks such as BIRD [18], Spider [40] and KaggleDBQA [15].

As a second application, we demonstrate that we can use a verifier to improve generators, by directly verifying the generated output. This direct verification, however, immediately raises a question: if the verifier rejects the generated SQL, what should the

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836721

generator return to the user? This question reveals a limitation in the currently dominant design and assessment of generators: the assumption that generators must *always* return an answer. This assumption ignores the potentially significant cost of returning an incorrect answer, which may lead a user to incorrect decisions. Weighing this cost is particularly important in NL to SQL systems, because failures are *deceptive*: the data returned by an incorrect query is often plausible enough to appear correct, leading the user to accept and act on that incorrect information. We can reduce this cost by making a generator *selective*, able to *abstain* from providing an answer if its confidence is low [16, 35]. This explicit abstention alerts the user to a failure—whether due to an ambiguous natural language request that requires reformulation, or simply a limitation in the model's ability to accurately translate a well-formed query—preventing them from acting on incorrect data.

This paper makes the following contributions:

- We propose and evaluate two complementary, model-based verification approaches—*Round-Trip Critique* and *Synthetic Execution Consistency*— to assess whether generated SQL aligns with the user's natural language intent, without requiring gold SQL labels or execution against a live database.
- We extend existing Text-to-SQL benchmarks (Spider, BIRD, and KaggleDBQA) by introducing new annotations to support research on verification methods.
- We show that (combinations of) our verification approaches achieve high accuracy, and that using them as a check improves the reliability of SQL generators.
- We show how our verification algorithms can improve the performance of *selective* generators that abstain from answering certain requests.
- We release our newly constructed verification-benchmark dataset alongside open-source implementations of all verification methods, including prompting templates and evaluation scripts.

2 RELATED WORK

Our work sits at the intersection of natural language interfaces, formal verification, and LLM self-evaluation. We enable lightweight verification of generated SQL using only the schema, without requiring gold queries or production data access.

Software Verification and the Cost of Failure. Traditional programming relies on compilers and unit tests to mitigate errors [3, 6]. Text-to-SQL systems often lack these guardrails, leaving them vulnerable to "silent failures"—plausible and syntactically valid but semantically incorrect queries that can cause catastrophic outcomes in high-stakes domains [24, 33]. LLM-generated explanations exacerbate this risk by fostering "automated reliance" on flawed outputs [14, 34]. Just as compilers and testing protect programmers, SQL verifiers can act as gatekeepers against undetected misinformation [32].

Evaluation of Text-to-SQL Systems. Benchmarks like Spider [40], BIRD [18], and KaggleDBQA [15] typically rely on Execution Accuracy (EA). However, EA is a noisy proxy for semantic correctness that distorts system evaluation in two distinct ways. First, EA can artificially *inflate* a system's reported accuracy: a generated query may be semantically incorrect but coincidentally yield the same result as the reference query on a sparse database [5, 20]. Second, EA can artificially *deflate* a system's score when the benchmark's "gold" query itself is wrong [25, 26]. If a gold query fails to match the natural language intent and produces an incorrect result, a benchmarked system that generates the genuinely correct query will be unfairly penalized under EA.

While formally proving semantic correctness is computationally intractable for many SQL fragments, our *Synthetic Execution* verifier is able to use LLM reasoning to demonstrate semantic *incorrectness*. Indeed, one of our contributions is demonstrating that hundreds of the "gold" queries in popular benchmarks are fundamentally flawed, leading to precisely these types of evaluation distortions.

Verification and Self-Evaluation of LLM Outputs. Verifying LLM outputs have emerged as a promising direction beyond static correctness metrics. In code generation, researchers have explored direct critiques [4, 29, 30] to detect bugs or inconsistencies. Other directions include Round-Trip Correctness [1], where a model's output is reverse-translated and compared with the original input, and dynamic verification such as test generation [21], while this offers stronger guarantees, they often require unit tests or formal specifications that are unavailable in ad-hoc Text-to-SQL tasks. To address this, recent work in Text-to-SQL evaluation generates synthetic, rule-compliant databases to rigorously test model accuracy and uncover hidden logic errors [11, 31, 42]. Building upon these foundations, our work formalizes Text-to-SQL verification as a standalone task that operates at inference time.

Verification in Text-to-SQL Pipelines. Systems use execution-guided decoding [38], extended context [5], and multi-stage "generate –execute–refine" loops [7, 27] to improve correctness. State-of-the-art models incorporate internal selectors and validity checks to choose among candidates [27, 39], but generally commit to returning a "best choice" even if all are incorrect. Here we argue for considering the verification problem on its own, yielding improved algorithms that can be incorporated into any of these generate-and-verify loops as well as enabled principled selective generation.

Abstention and its Cost/Benefit. Wang et al. Wang et al. [35] argue that systems should abstain from answering certain natural language queries, specifically those that are *ambiguous* or *unanswerable* due to missing data. They propose models to both detect and generate such queries to improve system reliability. Lee et al. [16] extends this by introducing TrustSQL, a metric evaluating a generator's ability to abstain or respond correctly. While they make use of a standard critique step to verify query correctness, their main technical contribution is a specialized modeling for ab-initio detection of queries that are *ambiguous* or *fundamentally unanswerable* from the available database schema. In contrast, we focus on post-hoc improved verification algorithms and show how they improve the quality of selective generation.

3 VERIFICATION METHODS

We formalize a verifier V as an inference-time function that assesses the semantic alignment between a natural language query q and a candidate SQL statement s for a given schema d:

$$V(q, s, d) \mapsto v$$

Notably, to be useful in real world deployments, verifiers as we consider them must operate without access to gold-standard SQL references. The output v follows one of two formats:

- **Binary Verifier:** $v \in \{0, 1\}$, representing a categorical rejection or acceptance.
- **Scoring Verifier:** $v \in \{1, \ldots, 5\}$, a discrete Likert scale from no alignment (1) to perfect equivalence (5).

Scoring verifiers are binarized by accepting queries where $v \geq \tau$ for a threshold $\tau \in [1, 5]$.

We now describe each method in detail.

3.1 SQL Critique (SC) - Direct Critique Baseline

Intuition. An intuitive baseline verifier directly prompts an LLM to evaluate the semantic and logical correspondence between the user's question q, the generated SQL query s, and the database schema d. As the most straightforward approach, this direct critique baseline is widely adopted in existing text-to-SQL pipelines [5, 9, 16, 27]. Evaluating this baseline allows us to measure the marginal value added by our proposed structured, multi-step verification methods. For our implementation, we adopt the error detection module (ERROR$_P$) from TrustSQL [16].

Formulation. We define a critique model $C_{\text{error\_p}}$ that maps the (q, s, d) tuple to a binary verification score $v_{\text{error\_p}} \in \{0, 1\}$. Following the TrustSQL template (illustrated in Figure 2), the verifier enforces a strict judgment and rejects the query if $v_{\text{error\_p}} = 0$ (indicating an incorrect response):

$$v_{\text{error\_p}} = C_{\text{error\_p}}(q, s, d)$$

Figure 2: An illustration of SQL Critique pipeline.

3.2 Round-Trip Critique (RT)

Intuition. If a SQL query s faithfully represents the original natural language query q, translating s back into natural language should yield a query q' semantically equivalent to q. A separate critique model compares q and q' to detect subtle logical shifts (e.g., incorrect negations or flipped operators) that embedding-based similarities often miss. Figure 3 illustrates this flow.

Formulation. Let a reverse translation model R generate a natural language question q' from generated SQL s and schema d:

$$q' = R(s, d)$$

A critique model C evaluates semantic alignment, producing a verification score v_{rt}. The verifier rejects the SQL query if $v_{rt} < \tau_{rt}$, where τ_{rt} is a predefined threshold:

$$v_{rt} = C(q, q'), \quad v_{rt} \in [1, 5]$$

To prevent this critique model from being biased by knowledge of the original NLQ, we isolate the back-translation model R from the original query q. If the generated SQL misrepresents the user's intent, an independent reverse translation is unlikely to perfectly "undo" the error, thereby exposing the semantic drift.

We experimented with continuous, binary, and Likert scales, finding that a discrete 5-point Likert scale provides the best overall balance for our primary framework. Prior studies confirm discrete scales maximize human-LLM alignment [19], whereas continuous scales introduce numeric token noise [10]. A discrete scale also allows us to anchor judgments using few-shot exemplars with clear qualitative boundaries. Notably, while the Likert setup yields the best general verification calibration across our core modules, strict binary scoring proves slightly more optimal in specific contexts, as detailed in our second baseline (Section 3.1).

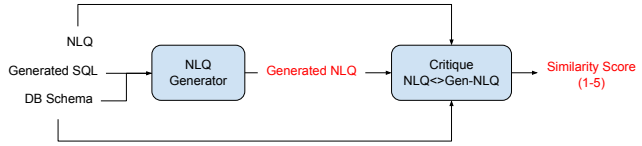

Figure 3: An illustration of the RT pipeline.

3.3 Synthetic Execution Consistency (SE)

Intuition. This method evaluates s by executing it on small, LLM-generated test databases to see if the query result matches the LLM's expected answer. To maximize edge-case coverage, we prompt the LLM to generate multiple test pairs using one of two approaches:

- *Answer-first*: The model invents plausible answers to q, then synthesizes a minimal database instance that makes those answers valid.
- *Data-first*: The model generates plausible database states, then deduces the correct answers from them.

Figure 4 illustrates this pipeline, which leverages a core asymmetry: generating a small data instance yielding a clear answer to an NLQ is significantly easier for an LLM than writing syntactically and logically flawless SQL. We generate synthetic data rather than using the real database for three practical reasons: (1) Context Windows: real production databases are often too massive to fit into an LLM's context window; and (2) Relevant Data: randomly sampling real rows will often yield no rows relevant to the query, making the answer trivial and unrevealing; and (3) Targeted Edge Cases: synthetic generation lets us explicitly inject crucial edge cases—such as zeros, nulls, duplicate values, or empty tables—to stress-test the SQL logic.

Formulation. We use a generator D_{gen} and answer model A to construct a test pair $(D_{\text{synth}}, a_{\text{pred}})$ via one of two modes: (1) Answer-first: $a_{\text{pred}} = A(q, d)$, then $D_{\text{synth}} = D_{\text{gen}}(q, a_{\text{pred}}, d)$; or (2) Data-first: $D_{\text{synth}} = D_{\text{gen}}(q, d)$, then $a_{\text{pred}} = \text{A}(q, D_{\text{synth}})$.

The candidate SQL s is executed to obtain $a_{\text{exec}} = \text{Exec}(s, D_{\text{synth}})$. The final binary verification decision v_{se} rejects the query if $v_{\text{se}} = 0$, defined via the indicator function:

$$v_{\text{se}} = \mathbb{I}(a_{\text{pred}} = a_{\text{exec}}), \quad v_{\text{se}} \in \{0, 1\}$$

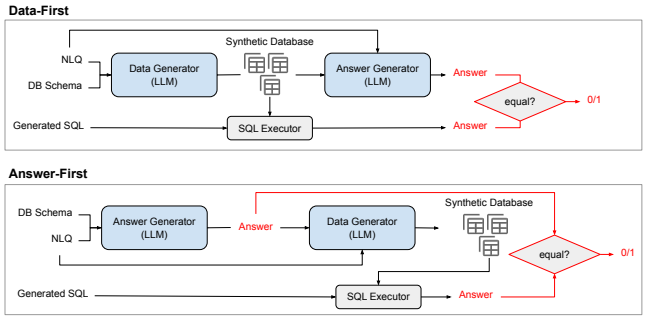

Figure 4: An illustration of the SE pipeline.

4 A VERIFICATION BENCHMARK

We have argued above, and will demonstrate further below, the research in verification algorithms is worthwhile. As with generation, such research can be strengthened by benchmarks that allow us to assess and compare different verification algorithms.

In this section, we describe the verification benchmark that we have created and released alongside this work, and demonstrate its use in evaluating our verification algorithms. Because we derive our verification benchmark from existing Text-to-SQL generation benchmarks (Figure 1), we begin by discussing their structure and inherent challenges. We then describe how verification benchmarks differ: in addition to examples of *correct* Text-to-SQL translations typically found in generator benchmarks, which a verifier should accept, a verification benchmark also needs examples of *incorrect* Text-to-SQL translations that it should reject.

4.1 Generation Benchmarks

Current Text-to-SQL benchmarks consist of evaluation pairs: (i) a natural language query and database schema, and (ii) a reference *gold* SQL query or execution result to determine correctness.

Inputs. At a minimum, each instance provides a natural language query. To generate SQL, systems also require a database schema in order to ground column and table references. Schema metadata alone often proves insufficient; systems may require semantic mappings (e.g., `status=2` for "completed sale") or sample values to resolve ambiguities [5, 22, 37].

Evaluation Criteria. To assess a generator's *candidate* SQL, benchmarks provide a *gold* reference and a rule for determining correctness. Existing benchmarks typically employ one of three approaches, ranging in their degree of semantic faithfulness and computational tractability:

- Semantic Equivalence: The gold standard, requiring matching results across all valid databases. It is generally undecidable and computationally intractable.
- Exact Match: Requires exact syntax after normalization. Extremely conservative, missing semantically identical queries with different structures.
- Execution Accuracy (EA): Compares two query results on a single benchmark database. Highly liberal, as semantically different queries can coincidentally yield identical results on specific data.

While metrics like Exact Match are now regarded as too rigid for modern LLM evaluation [17, 18, 40], formal semantic equivalence remains intractable. Consequently EA is the most common evaluation metric in current benchmarks. We therefore adopt EA for core labeling as the most pragmatic choice to ensure comparability with prior literature [15, 18, 40]. Unlike semantic equivalence metrics, EA requires an evaluation database, which all our benchmarks provide. Importantly, our framework uses EA as a baseline precisely to expose its flaws; our Synthetic Execution method and Section 6 analysis are designed to probe these vulnerabilities. Crucially, these benchmark criteria are *not* verification algorithms, as they rely on ground-truth references (gold SQL or results) that are unavailable to a verifier in a production environment.

We construct our verification benchmark from three widely used generation benchmarks:

- **Spider** [40]: 10,000+ questions across 200 databases from 100+ domains, emphasizing cross-domain generalization and complex SQL constructs. We use the Spider 1.0 test set.
- **BIRD** [18]: 12,000+ real-world business examples across 35 industries, emphasizing multi-turn reasoning and domain knowledge. We evaluate on the dev set.
- **KaggleDBQA** [15]: 544 practical analytical queries over eight Kaggle competition databases, with schema descriptions and example values. We use the test set.

Benchmark Flaws. Recent studies reveal significant noise in standard Text-to-SQL benchmarks, including faulty NLQ-to-SQL ground truths, "EA-correct" queries (semantically incorrect but yielding correct results on the benchmark database), and ambiguous questions [5, 41]. Crucially, unlike specialized benchmarks from TrustSQL [16] and DTE [35] that intentionally introduce flaws, these standard benchmarks fail to classify such problematic instances as unanswerable. Our verification algorithms offer a systematic mechanism to quantify, identify, and mitigate these inherent flaws.

4.2 Our Verification Benchmark

Standard Text-to-SQL benchmarks, like BIRD, pair each natural language question with a database schema and a single "gold" SQL query. While ideal for evaluating generation, this setup is insufficient for verification systems, which must judge both *correct* and *incorrect* SQL (Figure 1). The gold SQLs in generator benchmarks can be used to measure *false rejects* (flagging a correct query as wrong), but include no labeled *incorrect* SQLs that can be used to measure *false accepts* where an incorrect query is wrongly accepted.

While generating random incorrect SQL is trivial, such queries are too easily rejected. To rigorously stress-test verification, we need "near misses": plausible but subtly flawed SQL. Because mistakes from state-of-the-art generators naturally mimic human-like errors, we collect outputs from an advanced Text-to-SQL model and treat its mistakes as verification test cases.

Establishing a ground-truth for verification presents a unique challenge: it requires an oracle to decide which generated SQL is correct—the very problem our system aims to solve. However, while deployed systems lack access to ground-truth references, we can construct an evaluation benchmark by leveraging the evaluation rules of existing Text-to-SQL datasets. We transform traditional

generation benchmarks into verification tasks by following a systematic labeling pipeline:

- Select a natural language query q and its database schema d from the source benchmark.
- Produce a candidate SQL s for query q using a state-of-the-art generator.
- Determine the correctness label c by applying the benchmark's execution-based evaluation rule against the gold-standard reference.
- Record the quadruplet (q, s, d, c) as a verification instance, where a verifier is expected to output c given (q, s, d).

Using this framework, we construct our benchmark from the development sets of Spider, BIRD, and KaggleDBQA, generating candidate queries via a state-of-the-art model [5] and deriving labels c via Execution Accuracy (EA) [40].

We apply a uniform enrichment process to the database schema d across all datasets [5]. Because Text-to-SQL generators perform poorly without rich schema metadata due to degraded schema linking [22, 37], we augment d with column descriptions, example values, and database statistics (e.g., unique value counts). For Spider, which lacks native metadata, we generate these attributes synthetically. To optimize the context window, we use keyword heuristics to prune metadata for high-cardinality fields like `id` or `date`, though the columns themselves remain available in d for query generation and verification.

To normalize execution-based verification and mitigate ordering artifacts [15], we specially handle results with an `ORDER BY` clause in the gold SQL. When present, we apply only an inner sort over projected columns to account for schema variability while preserving vertical row order. Without `ORDER BY`, we perform both inner and outer sorting, treating results as unordered multisets. This protocol extends the order-invariant evaluation of Spider and BIRD—which typically rely only on global row sorting—by ensuring that variations in column sequence (common in LLM outputs) do not trigger false negatives if the underlying data is identical.

While automated and efficient, our approach inherits label noise from the source benchmarks, primarily stemming from ground-truth errors, spurious execution matches, and ambiguous queries. Consequently, verifiers may be penalized for correctly flagging semantic mismatches that the benchmark's execution rules overlook; we analyze this phenomenon in Section 6. Future human validation could mitigate this issue, which is highly feasible given that verifying SQL is significantly less labor-intensive than authoring it from scratch.

One may ask why a verification benchmark is needed if we can just verify candidate queries against gold benchmark SQLs. But this misunderstands the role of verification algorithms and benchmarks. EA is a plausible metric for comparing a candidate generated SQL to ground-truth data, but a real-world verification algorithm must operate "in the wild" where no gold SQL exists. Furthermore, local, non-standardized verification sets make cross-field results incomparable. By providing a fixed, public benchmark, we ensure verification algorithms are tested against a consistent set of hard cases without the computational overhead of running custom generators. Over time, as verification systems saturate this benchmark, the community can easily update it by swapping in newer generators.

Table 1 shows our benchmark distribution. As expected, incorrect queries scale with difficulty, peaking at 52.0% in BIRD's challenging split, offering a robust spectrum of near-miss error states.

Table 1: Benchmark distribution. Breakdown of correct and incorrect queries by source dataset and difficulty level.

Benchmark	Difficulty	Correct (%)	Incorrect (%)
BIRD	Simple	63.78	36.22
	Moderate	51.94	48.06
	Challenging	48.00	52.00
	Total	58.74	41.26
Spider 1.0	Easy	88.51	11.49
	Medium	83.78	16.22
	Hard	80.78	19.22
	Extra Hard	79.55	20.45
	Total	83.47	16.53
KaggleDBQA	Total	57.00	43.00

5 EXPERIMENTS

This section describes the performance of our proposed verification methods against the baseline direct critique method.

5.1 Experimental Setup

This section outlines our experimental setup, detailing the underlying LLM, evaluation datasets, and the metrics used to assess verification effectiveness.

5.1.1 Model. All experiments used *gemini-2.5-pro* via the Google Cloud Vertex AI API (global region). To ensure generative diversity, generation used a temperature of 1.0 and a randomized seed. Structured output mode was enforced to guarantee schema-conforming responses. The model relied exclusively on prompt context and parametric knowledge, using default Top-P (0.95) and maximum output token limits (65,535) with safety settings disabled. Inference jobs were parallelized via Cloud Run in *us-central1* (32 GiB memory, 8 vCPUs). We kept the underlying backbone LLM strictly identical across all evaluated methods (SC, RT, and SE) to ensure a controlled architectural evaluation.

5.1.2 Metrics. To quantify verifier reliability, we report two metrics: the *False Acceptance Rate (FAR)*, which measures the proportion of incorrect SQL queries wrongly accepted, and the *False Rejection Rate (FRR)*, which measures the proportion of correct queries wrongly rejected.

We use Execution Accuracy (EA) for core labeling to maintain direct comparability with prior literature [15, 18, 40], as it remains the standard when formal semantic equivalence is intractable. Thus, we label a false reject when the verifier rejects a gold SQL, or a generated SQL for the same NLQ that matches it under EA, while we label a false accept when the verifier accepts a generated SQL that disagrees with the gold SQL under EA.

Some of the "false rejects" of gold SQLs and "false accepts" of disagreeing generated SQLs turn out to be *correct* because the gold SQLs are *incorrect*. This highlights EA's known limitations (Section 4). We discuss implications of these benchmark flaws, and ways to cope with them, in Section 6.

5.2 Experimental Evaluation

5.2.1 Performance Summary of Individual Verifiers. We evaluate our SQL verification techniques by first analyzing individual verifier performance, focusing on the trade-off between FAR and FRR. Table 2 summarizes the performance of the three primary methods at their best operational threshold. For RT, we experimentally settled on the score threshold $\tau = 4$, as the other thresholds produced far worse results.

Table 2: Summary of Verifier Performance for False Acceptance Rate (FAR) and False Rejection Rate (FRR)

Method	FAR	FRR
SQL Critique (SC)	0.775	0.062
Round-Trip Critique (RT)	0.658	0.158
Synthetic Execution (SE)	0.595	0.169

All three of the methods are at optimal threshold, and they are ranked from the most permissive (SC with highest FAR and lowest FRR) to most conservative (SE with lowest FAR and highest FRR). As explored further in Section 6, this elevated FRR for SE largely stems from failures in the verification test itself (e.g., hallucinated synthetic data) and the exposure of latent errors within the benchmarks' gold SQL labels. In particular, we use sampling to show that with 99% confidence, the FRR of SE on *correct* SQLs is only 8.6%—the rest of the FRR measured here is due to its rejection of incorrect gold SQLs.

5.2.2 Individual Method Variations. We evaluated variations in prompting strategies and internal pipelines for RT and SE (summarized in Table 3). Our SC (TrustSQL's $ERROR_P$) was left in its original baseline configuration

For all 4 configurations, moving from zero-shot to few-shot prompting (ie, providing some examples of valid verifications) makes the verifiers more permissive, increasing FAR and decreasing FRR, but for most the change is minor.

SE's Answer-to-Data mode ($A \rightarrow D$) provides the lowest FAR (0.595 under zero-shot), whereas Data-to-Answer ($D \rightarrow A$) slightly favors FRR.

Table 3: Performance variations across individual verifiers at their operational thresholds ($\tau = 4$).

Method	Prompting Strategy	FAR	FRR
RT	Zero-shot	0.658	0.158
	Few-shot	0.715	0.127
SE ($A \rightarrow D$)	Zero-shot	0.595	0.169
	Few-shot	0.597	0.154
SE ($D \rightarrow A$)	Zero-shot	0.618	0.158
	Few-shot	0.632	0.131

5.2.3 Detailed Benchmark Breakdown. Table 4 breaks down the performance of the individual verifiers by benchmark and difficulty level. Performance fluctuates across datasets and query difficulties with no standout trend.

5.2.4 Performance Trade-offs via Verifier Combinations. We can improve the FAR–FRR tradeoff by combining verifiers. One natural class of conservative strategies uses a conjunction of verifiers (`AND`—all must accept) while a natural class of permissive strategies uses a disjunction (`OR`—any one can accept). To comprehensively evaluate the impact of combining verifiers, we analyzed the FAR and FRR metrics across all 8 possible `AND` and `OR` combinations of our verifiers (SC, RT, and SE). The optimal configurations are detailed in Table 5. We plot FAR and FRR for all combinations in Figure 5. At the top left is the most conservative strategy combining all verifiers with `AND` (FAR = 0.364, FRR = 0.326), while at the bottom right is the most permissive strategy combining all verifiers with `OR` (FAR = 0.914, FRR = 0.012).

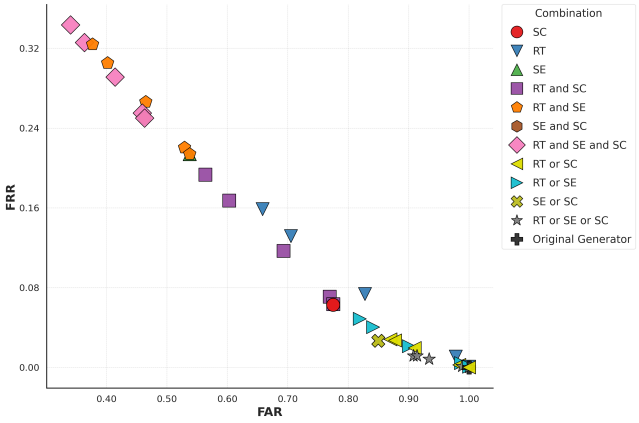

Figure 5: FAR vs. FRR across verifier combinations and thresholds. From top-left (conservative) to bottom-right (liberal), performance shifts are shown via transparent color.

5.2.5 Qualitative Analysis of Error Categories. To better understand the structural patterns driving these verification trade-offs, we manually analyzed a stratified sample of 40 queries across the four verification outcome categories: True Accept, False Accept, False Reject, and True Reject. For this analysis, candidate queries were evaluated using the joint RT AND SE conservative consensus pipeline, where an output is labeled as Accepted if and only if both verifiers accept it. We sampled 10 queries per category (4 from BIRD, 4 from Spider, and 2 from KaggleDBQA) to identify distinct structural and semantic signatures in each group:

- True Accept (Correct SQL, Accepted): These are simple queries where the question directly matches the database schema. Common examples include basic lookups with exact filters (like finding a post's score using its ID) or standard logic (like combining lists with INTERSECT) where the generated SQL matches the gold SQL perfectly.
- False Accept (Incorrect SQL, Accepted): These are broken queries that the verifier accidentally let pass. The main

Table 4: Detailed Verifier Performance Across Benchmarks and Difficulty

Benchmark	Difficulty	SQL Critique		Round-Trip Critique		Synthetic Execution	
		FAR	FRR	FAR	FRR	FAR	FRR
BIRD	Simple	0.812	0.095	0.693	0.215	0.644	0.247
	Moderate	0.664	0.162	0.583	0.278	0.484	0.354
	Challenging	0.600	0.200	0.440	0.329	0.345	0.317
	Total	0.747	0.125	0.636	0.245	0.567	0.286
Spider 1.0	Easy	0.889	0.024	0.556	0.101	0.608	0.103
	Medium	0.849	0.032	0.777	0.124	0.637	0.150
	Hard	0.764	0.040	0.652	0.096	0.698	0.104
	Extra Hard	0.849	0.042	0.836	0.095	0.743	0.074
	Total	0.839	0.034	0.712	0.108	0.661	0.117
KaggleDBQA	Total	0.860	0.039	0.632	0.351	0.509	0.247

Table 5: Performance Metrics (FAR and FRR) across Algorithm Combinations (at threshold 4)

Combination	FAR	FRR
RT and SC	0.603	0.167
SE and SC	0.463	0.250
RT and SE	0.402	0.305
RT and SE and SC	0.364	0.326
RT or SC	0.878	0.027
SE or SC	0.850	0.027
RT or SE	0.841	0.040
RT or SE or SC	0.914	0.012

problems are leaving out requested columns (like checking lap times without actually showing them in the output) or making filters too broad (like changing an AND to an OR, which pulls the wrong data). This often happens when the query misses necessary table joins or fails to use the right columns in the database.

- False Reject (Correct SQL, Rejected): These are correct queries that get wrongly rejected because they do not match the benchmark's SQL. This happens when the query solves the problem using a different approach (e.g., isolating a maximum value using a `MAX(CAST(column AS REAL))` aggregate instead of the benchmark's sorted `ORDER BY column DESC LIMIT 1` structure) or when it adds extra cleanup steps ((e.g., appending `NULLS LAST` or non-empty checks like `column <> ''`)) that the question did not ask for.
- True Reject (Incorrect SQL, Rejected): These are totally wrong queries that the verifier successfully rejected. Common failures include forgetting to group or add up data (like `SUM` and `GROUP BY`) or pulling information from completely wrong columns and tables.

Our qualitative breakdown uncovers three core failure modes:

- The verifier penalizes correct SQL code because it uses a different logic or math approach to get the answer (e.g., sorting and limiting rows instead of using a `MAX()` function).
- The verifier approves queries that have correct filters and joins, but output the wrong columns—either by leaving out requested columns or adding extra ones.
- The verifier approves queries where the conditions were loosened (e.g., changing strict conditions into a multi-way `OR`), which returns the wrong rows despite having a similar code structure.

5.2.6 Error Independence Analysis. We investigated whether the errors in our verifiers are correlated. Consider two verifiers with FAR's (ie, probabilities of false acceptance) p and q, respectively. The probability that a conjunctive (AND) verifier false-accepts is the probability that both base verifiers false-accept. If the verifiers err independently, then this probability is simply pq. On the other hand, a disjunctive (OR) verifier false-accepts if either verifier does, which under independence would happen with probability $1-(1-p)(1-q)$. For FRR the formulas are reversed: for base FRRs p' and q', the disjunctive verifier rejects if both base verifiers do—with probability $p'q'$ given independence—while the conjunctive verifier rejects if either does—with probability $1-(1-p')(1-q')$.

- At our operational threshold ($\tau = 4$), the independent expected FRR for RT AND SE is 0.3. Our empirical implementation yields a similar FRR of 0.305, indicating that under strict consensus, false rejections behave almost independently to filter out non-overlapping sets of valid queries.
- For RT OR SE, the independent error model yields a theoretical joint FRR of $p'q' = 0.158 \times 0.169 = 0.0267$. In practice, our empirical OR combination drives the actual joint FRR down even lower to nearly zero (mostly < 0.01). This slight deviation from pure independence suggests a minor complementary effect, where the structural weaknesses of the verifiers do not fully overlap, allowing valid queries to almost always be recognized by at least one pipeline.

This low error covariance indicates the verifiers are driven by distinct, non-overlapping systemic factors, capturing orthogonal signals from the queries rather than uniform noise.

5.3 Runtime Performance and Resource Utilization

In addition to verification accuracy, we evaluate the computational efficiency and cost-effectiveness of each method. Table 6 summarizes the average latency (in seconds), the number of LLM API calls required per verification, and the average token consumption for both zero-shot and few-shot prompting strategies. Combining these strategies compounds pipeline overhead to 4 API calls and over 90 seconds, revealing a steep trade-off between logical complexity and deployment costs.

Table 6: Performance and resource comparison. Latency denotes end-to-end processing time (s); token counts aggregate all internal pipeline stages.

Method	Calls	Avg. Latency	Tokens (Zero-Few Shot)
SC	1	8.06	1,288
RT	2	41.39	9,839 - 22,751
SE	2	51.22	16,949 - 24,897

6 ERRORS IN THE BENCHMARKS

Our initial analysis assumed that the gold SQLs of the underlying generator benchmarks were correct, implying that a verifier that rejected a gold SQL must be incorrect. However, as we dove deeper, we discovered that our verifiers were often *correct* in rejecting the gold SQLs because those benchmark SQLs were wrong. In this section, we describe types of incorrect gold SQLs and present two targeted sampling experiments that estimate how often these anomalies occur in standard text-to-SQL benchmarks we used.

6.1 A Taxonomy of Benchmark Issues

We identify three distinct types of benchmark problems that lead to verifier disagreements: (1) *data/instance incorrectness*, where the gold SQL is simply wrong given the benchmark database; (2) *semantic incorrectness*, where the gold SQL happens to work for the benchmark database instance but fails to capture the true semantics of the NLQ; and (3) *ambiguity*, where multiple valid interpretations of the NLQ yield conflicting SQL formulations, making it impossible to specify a single "correct" answer.

Data/Instance Incorrectness: The benchmark SQL is demonstrably wrong. For example, in KaggleDBQA (*"Sum of the acres burned in NC state in 2006?"*), it used `WHERE State = 'NY'` instead of `WHERE State = 'NC'`. Because this fails against its own data, any generated query returning the correct result is incorrectly labeled as an error under EA.

Semantic Incorrectness: The benchmark SQL produces a correct answer on the current instance by happenstance but fails to generalize to the schema. For example, for BIRD (*"How many cities are there in Orange County, Virginia?"*), the benchmark uses `SELECT COUNT(city)`, while our system generates `SELECT COUNT( DISTINCT city)`. Both yield identical results for "Orange County" because its entries happen to be unique. However, this query breaks on "Loudoun County" (36 rows, 16 unique cities), where the benchmark over-counts duplicate rows. One might object that this query is correct under the execution accuracy criterion, but this would mean that a clearly "cheating" query such as `SELECT 16` would have to be considered correct. Execution accuracy is a *concession to computational limitations* rather than the true measure of correctness and should not be a straitjacket for asserting correctness.

Ambiguous Natural Language Queries (NLQs): The NLQ admits multiple valid interpretations, penalizing models that fail to guess the annotator's specific subjective intent. For example, in *"Which is the most popular state... in terms of commodities distribution?"*, "most popular" could plausibly mean the highest frequency of records (`COUNT`) or largest volume of goods (`SUM`), which yield conflicting SQL queries and answers.

6.2 Quantifying Benchmark Issues

We used random sampling to estimate a *lower bound* the number of gold SQL errors in the popular generator benchmarks we used. In particular, we focused on gold SQLs that the SE verifier *rejected* and used sampling to estimate what fraction of those rejections were actually justified. There may be other erroneous gold SQLs that SE accepted, but we have no easy way to locate them.

Recall that to create our verification benchmark we took each (NLQ, gold SQL) pair from each generator benchmark and added it to our verification benchmark with a "correct" label. We then generated a second SQL from the NLQ and added it with the NLQ as a second pair to the verification benchmark, labeled "correct" if the generated SQL matched the gold SQL under EA and labeled "incorrect" if it did not match. It is those labels, hidden, that a verification algorithm is required to predict.

We partitioned the instances into two "scenarios" based on whether the verifier also rejected the second SQL that was generated for our benchmark from the same NLQ:

Scenario 1: For a given NLQ, SE rejects the gold SQL but accepts the generated SQL ($N = 357$). The fact that SE is rejecting the gold SQL but accepting a different SQL gives strong evidence that the gold SQL is incorrect—while a broken verifier might reject everything, it would be peculiar for the verifier to prefer a different wrong answer to the right one. So we expected that most of these gold SQLs would be incorrect (and this is indeed the case).

Scenario 2: SE rejects both generated and gold SQL for an NLQ ($N = 709$). . Here the evidence is more mixed; the rejection could be due to an incorrect SQL or could be do to failures of the verifier.

The cross product of 2 scenarios with the 3 underlying generator benchmarks defines 6 distinct *populations* of interest. We sampled 15 items at random from each of the 6 populations and manually determined how many of the sampled gold SQLs were incorrect. From these numbers we applied the standard binomial estimator to read off estimated (mean) number of gold errors in the population as well as 99% confidence intervals.

After looking at the 6 populations separately, we summed the estimates and CI lower/upper bounds to get estimates for each benchmark as well as for the each scenario, and for the entire population. Note that these subtotal confidences are conservatively computed from the 6 populations confidences and do not reflect additional experiments. There are more precise estimations for

Table 7: Estimated number of incorrect gold SQLs by benchmark and Scenario, with 99% confidence intervals.

Benchmark	Scenario	Population size	Sampled	Sample Errors	Estimated errors	99% CI
Bird	Scenario 1	172	15	14	160.5	(117, 171)
	Scenario 2	404	15	7	188.5	(85, 296)
	Subtotal	576	30	21	349.1	(202, 467)
Spider	Scenario 1	163	15	15	163.0	(127, 163)
	Scenario 2	271	15	7	126.5	(57, 201)
	Subtotal	434	30	22	289.5	(184, 364)
Kaggle	Scenario 1	23	15	14	21.5	(16, 22)
	Scenario 2	34	15	6	13.6	(7, 22)
	Subtotal	57	30	20	35.1	(23, 44)
Grand total		1067	90	63	673.6	(359, 910)
Scenario 1 subtotal		358	45	43	345.0	(260, 356)
Scenario 2 subtotal		709	45	20	328.6	(149, 519)

stratified sampling that can be used to narrow the confidence intervals (since errors in the 6 population samples are independent). But these conservative estimates suffice to give orders of magnitude. Our results are summarized in Table 7.

Practical Utility. These flagged subsets offer a cheap cleaning mechanism. Purging the 358 instances from the first scenario removes an estimated $\geq$ 354 errors (3815 − 357 = 3541 remaining queries). Purging Scenario 2 items removes an additional estimated $\geq$ 328 errors, leaving a higher-quality benchmark of 2,832 items.

6.3 Detailed Analysis of Samples

In this section we discuss the detailed characteristics of our 90 samples from the 6 different populations.

Scenario 1: Verifier Rejects Benchmark, Accepts Generated SQL. In this scenario, the SE Verifier actively corrects a benchmark flaw. Our audit reveals that across all three benchmarks, disagreements are overwhelmingly caused by systemic benchmark deficiencies rather than verifier flaws:

- BIRD (n = 15): We identified 14 benchmark-side issues. Among these, 10 cases exhibited severe logical errors in the gold SQL. For example, responding to *"What's the county that contains most recorded burns in 2000 according to the database?"*, the benchmark SQL group-by filtered incorrectly, entirely omitting the `WHERE year = 2000` clause. Additionally, 2 cases suffered from ambiguous NLQs and 2 instances appended unrequested extra columns. Only 1 case was a true boundary condition lacking schema constraints, where conflicting boolean representations of normal vs. abnormal laboratory ranges were used.
- Spider (n = 15): Every single disagreement (15/15) stemmed from benchmark deficiencies, split between 13 explicitly incorrect gold SQL queries (e.g., missing essential joins or columns) and 2 ambiguous NLQs.
- KaggleDBQA (n = 15): Benchmark deficiencies caused 14/15 errors—12 incorrect gold queries and 2 ambiguous questions. Only 1 error was a generator issue (including an extra column for clarity).

Scenario 2: Verifier Rejects Both Benchmark and Generated SQL. When both queries are rejected, the error profile shifts, revealing a mix of benchmark ambiguity and strict verifier constraints. Note that what we discuss here is the rejected *gold* SQLs, although the corresponding generated SQLs were rejected as well.

- BIRD (n = 15): There were 7 benchmark issues (5 wrong gold SQLs, 2 ambiguous NLQs) and 8 verifier errors. The verifier errors on the gold SQLs primarily occurred because the synthetic data generator introduced structural formatting variations (e.g., returning an entity ID instead of a name), triggering a strict rejection.
- Spider (n = 15): Documented 7 benchmark issues (5 ambiguous NLQs, 2 wrong gold SQLs) and 7 verifier edge cases. Notably, 2 of the verifier rejections stemmed from cases in the generated synthetic databases (e.g., evaluating *"Calculate the average height of all neutral superheroes"* against a synthetic schema that represented missing heights as −1, causing a breakdown in natural execution averages).
- KaggleDBQA (n = 15): Features 6 benchmark issues and 9 verifier over-rejections. Crucially, 7 of these 9 verifier errors were overly conservative rather than incorrect; the generator appended extra informational columns in the answer that were not explicitly requested in the prompt, which strictly failed the Execution Accuracy (EA) assertion despite being semantically rich.

The overall distribution of identified errors across these 90 combined samples highlights semantically incorrect benchmark SQL as the leading issue (46.67%), followed by verifier errors (27.78%) and ambiguous questions (21.11%).

Some of these observations point to ways to refine our verifiers in the future; for example, SE should clearly be prompted to provide *only* the data requested in the NLQ and nothing additional.

6.4 Re-Estimating True Verifier Performance

While we used EA to maintain consistency with the benchmarking literature, our manual audits reveal that this evaluation mechanism severely mischaracterizes verifier reliability. From our sample estimation of the frequency of incorrect gold SQLs, we can better bound the *actual FRR* and *actual FAR* of the SE verifier.

Our sampling experiment showed that many gold SQLs labeled as false rejects for our SE verifier were in fact *true* (correct) rejects because the gold SQL was incorrect. Using the sample to estimate the number of such occurrences, our mean estimate of the actual FRR drops from 0.169 to an estimated ~ 0.042, with a 99% confidence interval of $[0.018, 0.086]$. This indicates that the SE verifier is a safe gatekeeper that rarely discards genuinely correct queries, even though the flawed benchmark data makes its error rate look worse than it actually is.

Similarly, some of our benchmark's queries labeled as false accepts (because of an EA mismatch with the gold query) were in fact true accepts because they were correct and the gold SQL was incorrect. Our sampling estimation of the frequency of such instances lets us assert a decreased actual FAR from 0.595 to ~ 0.412, with a 99% confidence interval of $[0.325, 0.498]$.

The generator leaderboards, such as the BIRD Leaderboard[1], rank models based on these flawed benchmarks may well suffer from similar errors where generators are are rewarded for incorrect answers and penalized for correct ones. A similar experiment sampling a small number of generator submissions labeled "correct" and "incorrect" in the benchmark could help quantify the error and provide more accurate rankings.

7 SELECTIVE GENERATORS

Generative model evaluations traditionally use a "forced-response" mechanism, scoring models solely on correct answers ($n_{correct}$) for every query (Q). As noted in prior literature [16], this approach creates a perverse incentive: like a student facing no penalty for wrong answers, the model is motivated to guess rather than remain silent when uncertainty is high.

In real-world deployments, incorrect database queries are often deceptively distinct from explicit failures because they execute successfully but return plausible, yet incorrect, data (e.g., aggregating over the wrong rows). Users may confidently apply this flawed information with severe consequences [16, 35]. Given these risks, it can be better for a deployment-ready Text-to-SQL system to be selective, leveraging an internal or modular verification signal to *abstain* from providing an answer when its confidence is low.

An explicit abstention alerts the user to a generation hazard. Crucially, a verifier rejection does not inherently mean the natural language query was flawed or unanswerable; it frequently signals that the generator simply reached its capability limits or produced a faulty translation. By abstaining, the system safely signals a failure to the user, allowing them to reformulate or resubmit their request—a far less costly outcome than acting on deceptive misinformation.

A verifier provides a natural way to transform a non-selective generator into a selective one: after generation, test the result with the verifier and abstain if it is wrong. A high-quality verifier will avoid sending a user costly incorrect responses while canceling relatively few of the correct ones.

In this section, we evaluate the impact of using different verifiers on the performance of selective generators.

7.1 Relationship to Prior Abstention Models.

The paradigm of selective prediction and explicit abstention in Text-to-SQL was pioneered by TrustSQL [16], which evaluates reliability using a penalty-based framework we describe below, and was developed for pipelines targeting ambiguous or infeasible questions, similar to the DTE model [35]. TrustSQL's pipeline-based method approaches abstention via two distinct mechanisms: (1) an a-priori classifier trained to detect and abstain from structurally unanswerable or ambiguous queries *before* attempting to generate SQL, and (2) a post-hoc verifier module (which they call an error detector) $ERROR_P$ that evaluates the candidate SQL after generation.

Because the specialized, fine-tuned intent classifiers from the unanswerable-question detector are not publicly available, we could not incorporate them into our study. In any case, our work focuses on post-hoc correctness verification—assessing whether a model successfully translated a request, regardless of whether the initial query was clear or ambiguous. Any verifier can serve as a drop-in replacement for $ERROR_P$, and Section 5.2 showed that RT and SE perform significantly better than $ERROR_P$ (which we used as the baseline critique verifier), so we expect that either would significantly improve the performance of the TrustSQL generator on its metrics. We expect that integrating an a-priori unanswerability detector stage would uniformly benefit both their baseline and our structural verifiers by filtering out shared linguistic noise.

TrustSQL also trained their pipeline *on the benchmarks they were testing*. They did this in an appropriate way, separating each benchmark into a training set and test set, and reporting results only on the test set. However, training on a large portion of the benchmark likely gave their models substantial insight into what types of questions are answerable, and what correct SQL looks like. In contrast, the systems we consider here do not rely on such specialized training.

7.2 Metrics for Selective Prediction

Unlike traditional systems that are forced to answer every query, selective generators yield three distinct outcomes: correct responses, incorrect responses, and abstentions. As fractions summing to 1, these represent two degrees of freedom that require two numbers to fully describe system performance. Many pairs of numbers suffice. Following classical selective prediction literature [8], we focus on *coverage* (ϕ) and *selective accuracy* (A_{sel}):

- **Coverage (ϕ):** The fraction of questions the system chooses to answer: $\phi = \frac{n_{generated}}{Q}$
- **Selective Accuracy (A_{sel}):** The accuracy calculated strictly on the subset of questions where the system provides an answer: $A_{\text{sel}} = \frac{n_{correct}}{n_{generated}}$

There are other ways to represent the two degrees of freedom, and different communities have settled on different ones. For example, selective accuracy is mathematically equivalent to *precision* evaluated over the generated subset (where $n_{generated} = TP + FP$)

[1] https://bird-bench.github.io/

if we think of generation as "acceptance" by a classifier. Similarly, *selective risk* ($R_{\text{sel}} = \frac{n_{incorrect}}{n_{generated}} = 1 - A_{\text{sel}}$) is equivalent to the False Discovery Rate (FDR). We focus on Coverage and Precision because they are the standard for selective prediction tasks that mirror our task here. Other popular metrics such as precision are better suited to classification tasks, which ours is not. Also, we believe these two measures make natural sense to potential users of these systems.

Coverage and accuracy can be combined in context-sensitive ways to produce a single ranking score to help users choose among alternatives. For example, to linearize the trade-off between coverage and precision into a single scalar value for system ranking, we can adopt the penalty-weighted metric from TrustSQL [16], defined here as **Utility@X**:

$$U_X = \frac{n_{correct} - (X \cdot n_{incorrect})}{Q}$$

where X represents the penalty multiplier for an error relative to a success, and an explicit abstention yields 0. Both $n_{correct}$ and $n_{accepted}$ can easily be computed from ϕ, A, and Q and plugged in to yield U_X.

A single ranking metric metric like `Utility@X` has limitations, however. It inherently requires a deployer to pre-determine an arbitrary penalty parameter X, which varies heavily depending on whether the deployment environment is low-stakes ($X = 1$) or high-stakes ($X = 10$). Because the exact tradeoff is context dependent, we recommend that developers and evaluators of selective generation systems compute and report the two factors separately so that different implementers or users can make their own best choices based on their context. Instead of rankings at an arbitrarily selected X, scatter plots such as our Figure 6 can still concisely convey the comparison between different systems and indicate the Pareto frontier. Moving away from a single ranking can also encourage exploration of different designs for low-stakes and high-stakes systems without a fight for the proper value of "X".

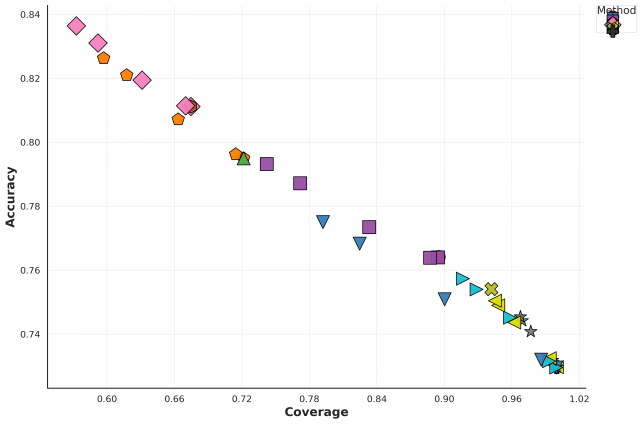

Figure 6: Coverage vs. selective accuracy across verifier combinations and thresholds. From top-left (high-reliability) to bottom-right (high-coverage).

7.3 Experimental Results

Verifiers transition "total" generators into selective ones: an answer is returned only if the verifier accepts the generator's output. We evaluate combinations of a state-of-the-art total generator [5] paired with our verifiers.

As Figure 6 shows, verification shifts performance upward and to the left, trading coverage for higher accuracy. Table 8 presents these configurations optimized across our operational thresholds. While the baseline generator achieves 100% coverage, it suffers from a 27% error rate (Precision = 0.730). An intersectional consensus composite of all three verifiers (RT and SC and SE) successfully drives this risk down, yielding a selective precision of 0.831 at 0.592 coverage. While this conservative posture slightly lowers `Utility@1` (0.392 vs. 0.459), it severely mitigates error penalties in high-stakes settings, changing `Utility@2` from 0.189 to 0.292.

Table 8: Performance metrics for individual verifiers and the selected composite progression (at threshold 4). Configurations are optimized for balanced performance.

Method	RT_{th}	ϕ	A_{sel}	U@1	U@2
Original Generator	-	1.000	0.730	0.459	0.189
SC	-	0.895	0.764	0.473	0.262
RT	4	0.825	0.768	0.442	0.251
SE	-	0.722	0.795	0.426	0.278
RT and SC	4	0.772	0.787	0.443	0.279
SE and SC	-	0.675	0.811	0.420	0.292
RT and SE	4	0.618	0.821	0.396	0.286
RT and SE and SC	4	0.592	0.831	0.392	0.292
RT or SC	4	0.948	0.749	0.472	0.234
SE or SC	-	0.942	0.754	0.479	0.247
RT or SE	4	0.929	0.754	0.472	0.243
RT or SE or SC	4	0.969	0.744	0.473	0.225

8 DISCUSSION

The Challenge of Ambiguity in Benchmarks. Defining correctness in Text-to-SQL remains difficult when user intent lacks a single, clean database mapping. While benchmarks like TrustSQL [16] and DTE [35] integrate these edge cases, they treat ambiguity through the broader lens of unanswerability, framing it as a binary trigger for absolute abstention. We argue that ambiguous queries deserve distinct treatment from inherently unanswerable ones (e.g., missing schema data). While missing-data queries are dead ends, ambiguous queries are conditionally answerable if a system can map the user's underlying intent. Robust interfaces require evaluation standards that move beyond static silence. Although TrustSQL acknowledges active clarification conceptually, they relegate it to future work [16]. Next-generation benchmarks should instead reward comprehensive coverage—returning valid results for all plausible interpretations—or targeted active clarification. Crucially, when systems flag ambiguity, our verifier-generated diagnostic signals (e.g., conflicting execution results in SE) can supply the structural context needed for explanation modules to guide user reformulation, turning a vague query into a precise execution.

Beyond Execution Accuracy. While we use EA for comparability with the Spider and BIRD literature, our framework is not constrained by its flaws. Rather, the primary value of our verification methods—particularly Synthetic Execution—is demonstrated precisely when they diverge from EA. By identifying logically flawed queries that coincidentally produce correct results on benchmark databases (Section 6), our verifiers actively expose EA's blind spots. We use EA merely as a bootstrap mechanism to transition the evaluation paradigm toward robust Semantic Equivalence. One practical improvement going forward would be to provide each benchmark NLQ with *multiple* benchmark databases and define a correct SQL to be one that yields the correct answer on *all* the databases.

Limitations. Our benchmark pairs correct Text-to-SQL translations with incorrect ones generated by current models. While the correct examples remain robust for testing false reject rates, the incorrect samples may not reflect the increasingly subtle errors of future generators, requiring periodic updates with more sophisticated "hard" instances.

Furthermore, accurately assessing correctness remains difficult when result ordering is ambiguous [15]. We currently assume order is significant only if the gold SQL contains an `ORDER BY` clause. However, this is an imperfect proxy for user intent; a top-k query might use `ORDER BY` in its implementation without strictly requiring an ordered final output, risking penalizing valid but unordered queries. To mitigate this, an ideal benchmark would include a hidden metadata flag indicating semantic order importance. We provide our best estimate of this information as an order-matters column in our published benchmark to support future evaluations.

While our SE verifier successfully identifies logical flaws using query-aware synthetic data, it does not yet guarantee exhaustive coverage of adversarial edge cases (e.g., boundary values, nulls). Because SE operates at inference time, systematically populating a comprehensive database to stress-test all schema constraints is computationally prohibitive. Future work will explore adversarial prompting strategies and formal SQL equivalence methods to generate highly challenging synthetic environments without incurring prohibitive latency.

Finally, while our ensemble methods rely on strict Boolean logic (AND/OR) to preserve a training-free pipeline, future work could explore training meta-classifiers or learning continuous mixing ratios (β) to optimally weight the signals from different verifiers.

9 CONCLUSION

Lightweight, model-based verification offers a powerful defense against incorrect Text-to-SQL generation. By combining Round-Trip Critique and Synthetic Execution Consistency, our framework flagged 64% of errors from a state-of-the-art model, validating the generate-then-verify paradigm as an effective guardrail for LLM trustworthiness. To support future research, we release a new verification-focused benchmark—extending Spider, BIRD, and KaggleDBQA with explicit evaluation annotations—offering a clear path toward more reliable natural language data interfaces.

REFERENCES

[1] Miltiadis Allamanis, Sheena Panthaplackel, and Pengcheng Yin. 2024. Unsupervised evaluation of code LLMs with round-trip correctness. In *Proceedings of the 41st International Conference on Machine Learning (ICML'24)*. JMLR.org, Vienna, Austria, Article 44, 17 pages.

[2] Arian Askari, Christian Poelitz, and Xinye Tang. 2025. MAGIC: generating self-correction guideline for in-context text-to-SQL. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'25/IAAI'25/EAAI'25)*. AAAI Press, Article 2613, 9 pages. https://doi.org/10.1609/aaai.v39i22.34511

[3] Paul E Black and Paul E Black. 2019. *Formal Methods for Statistical Software.* US Department of Commerce, National Institute of Standards and Technology.

[4] Ziru Chen, Shijie Chen, Michael White, Raymond Mooney, Ali Payani, Jayanth Srinivasa, Yu Su, and Huan Sun. 2023. Text-to-SQL Error Correction with Language Models of Code. In *Proceedings of the 61th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, Toronto, Canada. https://arxiv.org/abs/2305.13073

[5] Yeounoh Chung, Gaurav T. Kakkar, Yu Gan, Brenton Milne, and Fatma Özcan. 2025. Is Long Context All You Need? Leveraging LLM's Extended Context for NL2SQL. *Proc. VLDB Endow.* 18, 8 (April 2025), 2735–2747. https://doi.org/10.14778/3742728.3742761

[6] Edmund M Clarke, E Allen Emerson, and Joseph Sifakis. 2009. Model checking: algorithmic verification and debugging. *Commun. ACM* 52, 11 (2009), 74–84.

[7] Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, lu Chen, Jinshu Lin, and Dongfang Lou. 2023. C3: Zero-shot Text-to-SQL with ChatGPT. arXiv:2307.07306 [cs.CL] https://arxiv.org/abs/2307.07306

[8] Ran El-Yaniv and Yair Wiener. 2010. On the Foundations of Noise-free Selective Classification. *J. Mach. Learn. Res.* 11 (Aug. 2010), 1605–1641.

[9] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. arXiv:2308.15363 [cs.DB] https://arxiv.org/abs/2308.15363

[10] Luke Guerdan, Solon Barocas, Ken Holstein, Hanna Wallach, Steven Wu, and Alexandra Chouldechova. 2026. Validating LLM-as-a-Judge Systems under Rating Indeterminacy. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems.* https://openreview.net/forum?id=ZwDMrArTBg

[11] Mohammadamin Habibollah and Davood Rafiei. 2026. SynSQL: Synthesizing Relational Databases for Robust Evaluation of Text-to-SQL Systems. arXiv:2604.27261 [cs.DB] https://arxiv.org/abs/2604.27261

[12] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2025. Next-Generation Database Interfaces: A Survey of LLM-based Text-to-SQL. arXiv:2406.08426 [cs.CL] https://arxiv.org/abs/2406.08426

[13] George Katsogiannis-Meimarakis and Georgia Koutrika. 2023. A survey on deep learning approaches for text-to-SQL. *The VLDB Journal* 32, 4 (2023), 905–936. https://doi.org/10.1007/s00778-022-00776-8

[14] Sunnie SY Kim, Jennifer Wortman Vaughan, Q Vera Liao, Tania Lombrozo, and Olga Russakovsky. 2025. Fostering appropriate reliance on large language models: The role of explanations, sources, and inconsistencies. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems.* 1–19.

[15] 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)*. Association for Computational Linguistics, Online, 2261–2273. https://aclanthology.org/2021.acl-long.176

[16] Gyubok Lee, Woosog Chay, Seonhee Cho, and Edward Choi. 2024. TrustSQL: Benchmarking Text-to-SQL Reliability with Penalty-Based Scoring. arXiv:2403.15879 [cs.AI] https://arxiv.org/abs/2403.15879

[17] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763* (2024).

[18] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C.C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM already serve as a database interface? a big bench for large-scale database grounded text-to-SQLs. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NeurIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1835, 28 pages.

[19] Weiyue Li, Minda Zhao, Weixuan Dong, Jiahui Cai, Yuze Wei, Michael Pocress, Yi Li, Wanyan Yuan, Xiaoyue Wang, Ruoyu Hou, Kaiyuan Lou, Wenqi Zeng, Yutong Yang, Yilun Du, and Mengyu Wang. 2026. Grading Scale Impact on LLM-as-a-Judge: Human-LLM Alignment Is Highest on 0-5 Grading Scale. arXiv:2601.03444 [cs.CL] https://arxiv.org/abs/2601.03444

[20] Xinyu Liu, Shuyu Shen, Boyan Li, Nan Tang, and Yuyu Luo. [n.d.]. NL2SQL-BUGs: A benchmark for detecting semantic errors in NL2SQL translation, 2025. *URL https://arxiv.org/abs/2503.11984* ([n. d.]).

[21] Zihan Ma, Taolin Zhang, Maosong Cao, Junnan Liu, Wenwei Zhang, Minnan Luo, Songyang Zhang, and Kai Chen. 2025. Rethinking Verification for LLM

Code Generation: From Generation to Testing. arXiv:2507.06920 [cs.CL] https://arxiv.org/abs/2507.06920

[22] Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. 2024. The Death of Schema Linking? Text-to-SQL in the Age of Well-Reasoned Language Models. arXiv:2408.07702 [cs.CL] https://arxiv.org/abs/2408.07702

[23] Ali Mohammadjafari, Anthony S. Maida, and Raju Gottumukkala. 2025. From Natural Language to SQL: Review of LLM-based Text-to-SQL Systems. arXiv:2410.01066 [cs.CL] https://arxiv.org/abs/2410.01066

[24] Arpit Narechania, Adam Fourney, Bongshin Lee, and Gonzalo Ramos. 2021. DIY: Assessing the Correctness of Natural Language to SQL Systems. In *Proceedings of the 26th International Conference on Intelligent User Interfaces*. 597–607.

[25] Zheng Ning, Yuan Tian, Zheng Zhang, Tianyi Zhang, and Toby Jia-Jun Li. 2024. Insights into natural language database query errors: From attention misalignment to user handling strategies. *ACM Transactions on Interactive Intelligent Systems* 14, 4 (2024), 1–32.

[26] Zheng Ning, Zheng Zhang, Tianyi Sun, Yuan Tian, Tianyi Zhang, and Toby Jia-Jun Li. 2023. An empirical study of model errors and user error discovery and repair strategies in natural language database queries. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*. 633–649.

[27] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2024. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. *arXiv preprint arXiv:2410.01943* (2024).

[28] Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 36339–36348. https://proceedings.neurips.cc/paper_files/paper/2023/file/72223cc66f63ca1aa59edaec1b3670e6-Paper-Conference.pdf

[29] Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NeurIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 2997, 13 pages.

[30] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NeurIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 377, 19 pages.

[31] Sunanda Somwase, Parismita Das, and S. Sudarshan. 2025. Test Data Generation for Complex SQL Queries. *Proc. ACM Manag. Data* 3, 6, Article 367 (Dec. 2025), 26 pages. https://doi.org/10.1145/3769832

[32] Yanqi Song, Ruiheng Liu, Shu Chen, Qianhao Ren, Yu Zhang, and Yongqi Yu. 2024. SecureSQL: Evaluating Data Leakage of Large Language Models as Natural Language Interfaces to Databases. In *Findings of the Association for Computational Linguistics: EMNLP 2024*. 5975–5990.

[33] Yuan Tian, Jonathan K Kummerfeld, Toby Jia-Jun Li, and Tianyi Zhang. 2024. SQLucid: Grounding Natural Language Database Queries with Interactive Explanations. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. 1–20.

[34] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI conference on human factors in computing systems extended abstracts*. 1–7.

[35] Bing Wang, Yan Gao, Zhoujun Li, and Jian-Guang Lou. 2023. Know What I don't Know: Handling Ambiguous and Unknown Questions for Text-to-SQL. In *Findings of the Association for Computational Linguistics: ACL 2023*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 5701–5714. https://doi.org/10.18653/v1/2023.findings-acl.352

[36] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. In *Proceedings of the 31st International Conference on Computational Linguistics*, Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert (Eds.). Association for Computational Linguistics, Abu Dhabi, UAE, 540–557. https://aclanthology.org/2025.coling-main.36/

[37] Bailin Wang, Richard Shin, Xiaodong Liu, Alex Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *ACL 2020*. https://www.microsoft.com/en-us/research/publication/rat-sql-relation-aware-schema-encoding-and-linking-for-text-to-sql-parsers/

[38] Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. 2018. Robust Text-to-SQL Generation with Execution-Guided Decoding. arXiv:1807.03100 [cs.CL] https://arxiv.org/abs/1807.03100

[39] Pengfei Wang, Baolin Sun, Xuemei Dong, Yaxun Dai, Hongwei Yuan, Mengdie Chu, Yingqi Gao, Xiang Qi, Peng Zhang, and Ying Yan. 2025. Agentar-Scale-SQL: Advancing Text-to-SQL through Orchestrated Test-Time Scaling. *arXiv preprint arXiv:2509.24403* (2025).

[40] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, Brussels, Belgium, 3911–3921. https://doi.org/10.18653/v1/D18-1425

[41] Janghyeon Yun and Sang-goo Lee. 2025. SEED: Enhancing Text-to-SQL Performance and Practical Usability Through Automatic Evidence Generation. *arXiv e-prints* (June 2025). https://doi.org/10.48550/arXiv.2506.07423 arXiv:2506.07423 [cs.CL]

[42] Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic Evaluation for Text-to-SQL with Distilled Test Suites. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 396–411. https://doi.org/10.18653/v1/2020.emnlp-main.29