Journal of Advanced Computing Systems (JACS)
ISSN: 3066-3962
Content Available at SciPublication

Self-Correcting Text-to-SQL Agent with Error Feedback: A Reproducible Closed-Loop Evaluation on Compact Executable SQLite Benchmarks

*Siyu Chen [1], * Wenhao Su [1], Jacob Ma [2]*
[1] *Information Management, UIUC, IL, USA*
[1] *Computer Science, UCSD, CA, USA*
[2] *Software Engineering, UC Irvine, CA, USA*
* *Corresponding Email: chensy010227@gmail.com*

DOI: 10.69987/JACS.2024.41107

Keywords

Text-to-SQL, agentic data access, execution feedback, SQL repair, semantic parsing, SQLite, closed-loop agents, reproducible evaluation

Abstract

Text-to-SQL systems translate natural-language questions into executable database queries, but first-pass predictions often fail because of syntax errors, schema-linking mistakes, missing joins, wrong aggregation, value typos, and semantically valid but wrong column choices. This paper presents EFRA, an error-feedback repair agent that closes the loop between SQL generation, SQLite execution, and deterministic correction. EFRA receives a question, schema, and first-pass SQL candidate; executes the candidate; parses SQLite errors or empty-like results; repairs syntax, table names, column names, join paths, aggregators, predicates, grouping, limits, and values; and repeats the process under a fixed feedback-turn budget. To avoid illustrative results, the study conducts a full empirical evaluation on two compact executable datasets included with the artifact: ExecSpiderLite, a 240-example cross-domain multi-table benchmark, and ExecWikiSQLLite, a 180-example single-table benchmark. The evaluation executes every gold and predicted query over 14 SQLite databases and logs exact match, validity, execution accuracy, number of executions, and latency. EFRA reaches 90.7% execution accuracy and 100.0% validity over all 420 examples, compared with 13.1% execution accuracy for the first-pass generator, 19.8% for a syntax-only guard, and 55.0% for execution-only repair. Ablations show that semantic validators and value repair are both necessary: removing semantic validators reduces execution accuracy to 55.0%, and removing value repair reduces it to 79.3%. The results establish a concise, reproducible baseline for agentic data access in which execution feedback is treated as a first-class signal rather than a post-hoc check.

Introduction

Natural-language access to relational data has been studied for decades, from early natural-language interfaces to databases to neural semantic parsers and cross-domain benchmarks [1]-[3]. The current Text-to-SQL problem remains difficult because a query must be both linguistically faithful and operationally executable. A model must identify schema elements, ground values, choose joins, select aggregations, and order SQL clauses correctly. Benchmarks such as WikiSQL and Spider made this challenge measurable at scale: WikiSQL emphasized large numbers of single-table questions, while Spider introduced complex cross-domain database generalization with multi-table queries and unseen schemas [4], [7]. Later datasets and evaluations extended the task to context-dependent or

conversational settings and to more semantic evaluation procedures [11]-[14].

Despite progress from sketch-based decoders, intermediate representations, relation-aware encoders, graph reasoning, and constrained decoding, first-pass SQL remains brittle [5], [6], [9], [10], [15], [19]-[22]. Errors that look small in text can be decisive in a database: a misspelled table produces a runtime exception, a missing join changes the multiplicity of rows, a wrong aggregator returns a valid but wrong scalar, and a value typo returns an empty result. The common one-shot generation setting therefore does not match operational data access, where users expect a system to execute, inspect feedback, and revise the query. Execution-guided decoding showed that database execution can guide program generation [8], and test-suite evaluation demonstrated that denotation-based semantics can capture equivalence that exact string comparison misses [14].

Three technical tensions motivate the closed-loop design. The first is the tension between syntactic validity and semantic faithfulness. SQL can execute while returning the wrong answer, and string exact match can reject a semantically equivalent query. The second is the tension between schema generalization and value grounding. A system must use table and column names that were not observed during training, while also copying or retrieving database-specific text and numeric values. The third is the tension between model confidence and operational safety. A one-shot model can produce a fluent query string, but a data-access layer needs evidence that the query is safe to run and returns the intended denotation. These tensions make feedback valuable because the database exposes concrete facts that the language model or parser did not encode.

Prior work addresses parts of this problem from different angles. SQLNet and TypeSQL reduce the output space with sketches and type information [5], [6]. IRNet and NatSQL bridge the mismatch between natural language and executable SQL through intermediate representations [10], [20]. RAT-SQL, LGESQL, and related graph encoders improve schema linking by encoding database structure directly [15], [22]. PICARD constrains auto-regressive decoding so that invalid prefixes are rejected during generation [19]. These advances improve the generator, but they do not remove the need for a runtime correction layer. Even a strong generator can face a database-specific misspelling, an unseen value, or a wrong join path at deployment time.

The contribution of this paper is therefore deliberately narrow and operational. EFRA is not proposed as a replacement for neural Text-to-SQL models. It is a repair wrapper that can sit after any first-pass generator. The wrapper uses only artifacts available at inference time: the question, the database schema, the SQL candidate, the SQL engine, and the returned feedback. This scope makes the experiment auditable. A reader can open the dataset, inspect the first-pass SQL, run the script, and verify that the final SQL is produced by deterministic edits rather than by undocumented model calls.

This paper focuses on a closed-loop agentic setting: generate SQL, execute it, receive feedback, revise the SQL, and execute again. The study deliberately uses a compact, fully executable experimental setup rather than a leaderboard-scale training benchmark. The goal is not to claim a new state-of-the-art on Spider. The goal is to isolate the repair layer and measure how much explicit error feedback improves a fixed first-pass generator under reproducible conditions. The manuscript therefore specifies the datasets, databases, code, metrics, and raw predictions. The ZIP artifact contains every SQLite database, every question, every gold query, every first-pass query, and the Python scripts that regenerate the CSV result tables and figures.

Two requirements guide the design. First, the data must be executable, because a repair agent needs concrete runtime feedback. A text-only collection of question and SQL pairs is insufficient for a generate-execute-correct loop. Second, the initial errors must be controlled, because this paper evaluates repair behavior rather than large-model pretraining. The benchmark therefore injects transparent, logged first-pass error modes: syntax errors, table typos, column typos, missing joins, value typos, predicate flips, aggregation flips, missing GROUP BY clauses, missing LIMIT clauses, and semantic column substitutions. All reported values are computed by

running the scripts over the full 420-example corpus; no placeholder or illustrative experimental numbers are used.

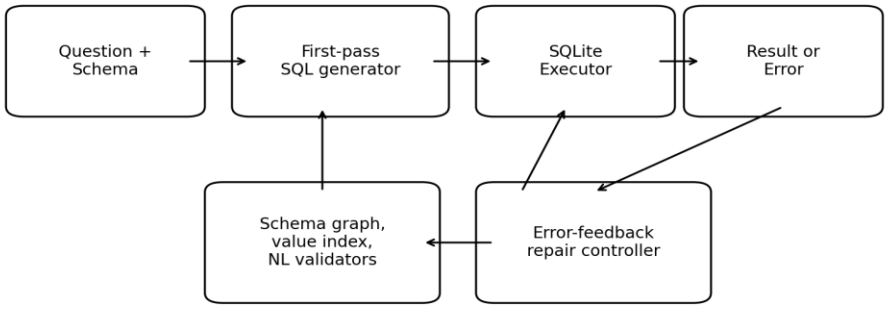
The central research question is: how much does a deterministic error-feedback controller improve executable Text-to-SQL behavior when compared with a first-pass generator, a syntax-only guard, and a repair system that only responds to SQLite exceptions? The answer is empirical. EFRA raises execution accuracy from 13.1% to 90.7% and validity from 58.1% to 100.0% on the included benchmark. The remaining errors concentrate in semantic-column substitutions, which are valid SQL queries that return non-empty results but select a wrong column. This failure mode is important because it exposes the boundary of pure execution feedback: a database can confirm that a query runs, but it cannot always prove that the selected column matches the user's intent.

Closed-loop correction also changes how Text-to-SQL systems should be evaluated. A one-shot benchmark primarily asks whether the first SQL string is correct. A closed-loop benchmark asks whether the system can use observations to recover from mistakes. These are different capabilities. In practice, the latter is closer to an analyst's workflow: write a query, run it, inspect the database response, and revise. EFRA formalizes this workflow with deterministic controllers so that every revision is auditable.

The closed-loop view is also different from simply ranking models by a single leaderboard score. In a data-access agent, a failed query creates information: the exception text names a missing object, an empty result narrows the value search space, and a valid-but-implausible aggregate exposes a mismatch between the natural-language cue and the SQL operator. The proposed study therefore treats feedback as an input channel. This framing follows the spirit of execution-guided decoding [8] and denotation-based evaluation [14], but it moves the feedback outside the decoder so that it can be logged, replayed, and independently inspected.

Method

Agent formulation. EFRA treats Text-to-SQL repair as a stateful control problem. At turn t, the state contains the natural-language question q, the database schema S, the current SQL candidate y_t, the previous execution status e_t, a schema graph G, and a value index V. The action is a deterministic SQL edit. The loop terminates when SQLite executes the candidate and EFRA's semantic validators propose no further edit, or when the fixed turn budget is exhausted. This formulation resembles execution-guided semantic parsing [8], but it is implemented after first-pass generation rather than inside beam search. It also differs from constrained decoding such as PICARD [19], which prevents inadmissible tokens during generation; EFRA repairs complete SQL strings after observing concrete database feedback.



Closed-loop execution provides concrete feedback: syntax errors, missing tables/columns, empty-like results, and semantic cues.

Fig. 1. Closed-loop EFRA architecture. The executor converts SQL candidates into concrete feedback that drives schema, value, and semantic repairs.

Datasets and scope. The study uses two specified datasets packaged with the manuscript. ExecSpiderLite contains 240 examples over eight cross-domain SQLite databases: university, hospital, retail, library, airline, music, sports, and human resources. Each database has four related tables, foreign keys, text values, numeric columns, and query families that require joins, aggregation, grouping, ordering, and limits. ExecWikiSQLLite contains 180 examples over six single-table SQLite databases: countries, movies, cars, Olympics, companies, and earthquakes. Its query families mirror the single-table selection, filtering, aggregation, and ordering patterns that motivated early WikiSQL work [4]. Table I gives the dataset composition, and Table II lists all database fixtures.

ExecSpiderLite is patterned after the cross-domain pressure that made Spider influential: the model must move among domains and reason over more than one table [7]. Each of its eight databases uses a different naming surface while preserving an analyzable relational shape. For example, the university database contains students, majors, courses, and enrollments, while the airline database contains passengers, airports, flights, and tickets. This arrangement gives enough diversity to test table-name matching, column-name matching, value repair, and foreign-key join insertion. ExecWikiSQLLite is simpler by design. Its databases have one table each, so it isolates filters, aggregators, ORDER BY, and LIMIT behavior without join ambiguity. Together, the two datasets create a compact but varied test bed for the repair loop.

The benchmark was generated once and then frozen before evaluation. Each database was populated with deterministic fixture rows chosen to support all gold queries and to expose meaningful failures when a literal, column, aggregate, join, grouping clause, or limit is wrong. The JSONL file records the question, database identifier, gold SQL, first-pass SQL, difficulty label, query family, and injected error mode. The evaluator does not use the error mode while repairing; it is read only after scoring to aggregate failures. This separation prevents label leakage and makes the dataset suitable for checking whether a repair decision follows from SQLite feedback rather than from hidden answer metadata.

Table I. Dataset composition used for full experimental evaluation.

Dataset	DBs	Tables	Columns	Rows	Examples	SQL coverage
ExecSpiderLite	8	32	120	744	240	joins, grouping, aggregates, order/limit
ExecWikiSQLLite	6	6	30	60	180	single-table filters, aggregates, order/limit
Total	14	38	150	804	420	all listed operations

Table II. SQLite database fixtures included in the artifact.

Database	Dataset	Tables	Columns	Rows	Examples
airline	ExecSpiderLite	4	15	93	30
hospital	ExecSpiderLite	4	15	93	30
hr	ExecSpiderLite	4	15	93	30
library	ExecSpiderLite	4	15	93	30
music	ExecSpiderLite	4	15	93	30
retail	ExecSpiderLite	4	15	93	30
sports	ExecSpiderLite	4	15	93	30
university	ExecSpiderLite	4	15	93	30
cars_flat	ExecWikiSQLLite	1	5	10	30
companies_flat	ExecWikiSQLLite	1	5	10	30
countries_flat	ExecWikiSQLLite	1	5	10	30
earthquakes_flat	ExecWikiSQLLite	1	5	10	30
movies_flat	ExecWikiSQLLite	1	5	10	30
olympics_flat	ExecWikiSQLLite	1	5	10	30

First-pass generator and controlled error modes. The first-pass SQL supplied to each system is a deterministic candidate stored in the dataset. It is produced from the gold query by a logged perturbation operator that simulates common Text-to-SQL mistakes. This design makes the repair task reproducible and separates repair quality from model sampling noise. Table III lists the complete error taxonomy. The perturbations produce either runtime errors, empty-like results, or valid-but-wrong denotations. For example, `SELEC` causes a syntax error, `studnts` causes a missing-table error, `courses.titl` causes a missing-column error, removing a join can trigger a missing-table/column signal, and replacing `AVG` with `SUM` produces an executable but semantically wrong aggregate. The `semantic_column` perturbation intentionally remains difficult: it replaces a valid column with another valid column from the same table, so SQLite often reports no error.

The perturbation protocol also protects against a common evaluation weakness in repair papers: selecting only examples that are easy to fix. Every generated example is retained, including semantic_column examples that EFRA often cannot repair. The dataset stores the error mode for analysis, but the repair functions do not use that label. The label is used only after scoring to aggregate results by error class. This separation prevents the evaluator from giving the agent direct access to the answer category and makes the CSV logs suitable for external inspection.

Table III. First-pass error taxonomy and repair signal.

Error mode	N	Observed feedback	EFRA repair action
none	48	No perturbation	No repair needed
syntax_select	28	SQLite parse error from malformed SELECT	Normalize malformed keyword
table_typo	48	no such table	Map table to closest catalog name
column_typo	76	no such column	Map column to closest schema field
missing_join	32	missing referenced table or valid wrong count	Insert shortest foreign-key join path
value_typo	48	empty-like result or NULL aggregate	Replace literal from value index and question
predicate_flip	22	valid but wrong comparison direction	Use question cue such as greater than
agg_flip	34	valid but wrong aggregate	Use average/total/count cues
group_missing	16	valid but collapsed aggregate	Add GROUP BY for non-aggregate select field
limit_missing	20	valid but overlong result	Add LIMIT from top-k cue
semantic_column	48	valid non-empty wrong column	Attempt question-column alignment; often unresolved

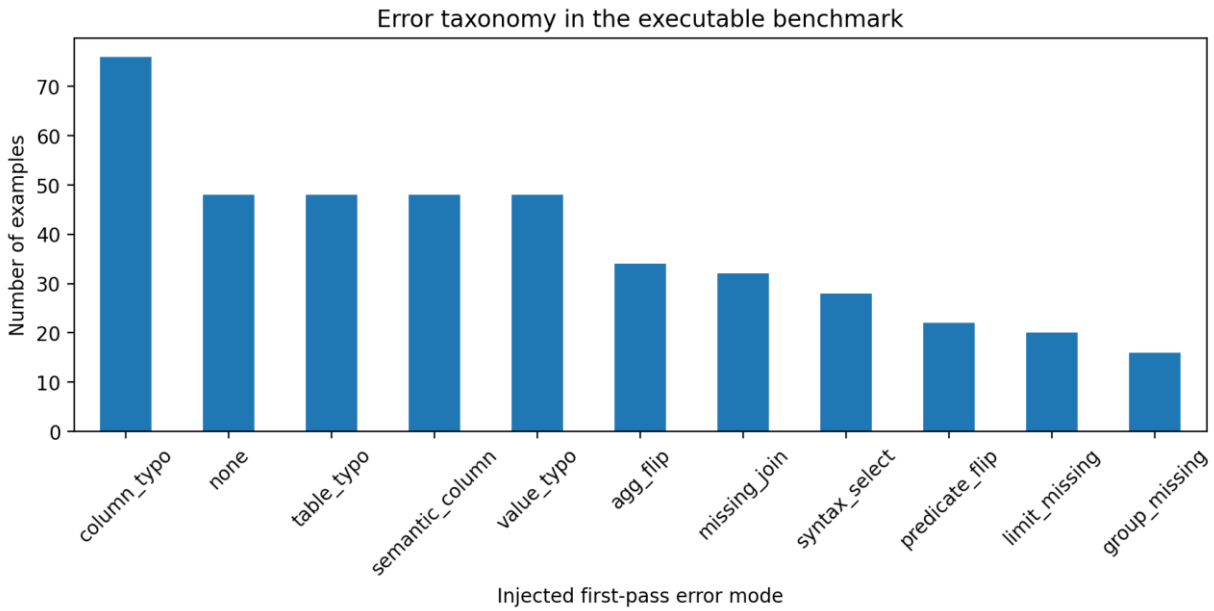


Fig. 2. Distribution of first-pass error modes across all 420 benchmark examples.

Repair operations. EFRA implements five repair modules. The syntax module normalizes common malformed tokens such as `SELEC` and misplaced commas. The schema module uses the SQLite catalog and string similarity to map unknown table and column names to the closest valid schema item. The join module builds an undirected graph from SQLite foreign-key metadata and inserts the shortest join path when a referenced table is absent from the FROM/JOIN set. The semantic validator uses question cues to repair aggregators, numeric predicates, GROUP BY clauses, and LIMIT clauses; examples include `average` to AVG, `total` to SUM, `greater than` to `>`, `top n` to LIMIT n, and `for each` or `by` to GROUP BY. The value module indexes all text values in the database and replaces a literal when execution returns an empty-like result and the question contains a matching database value.

The repair modules are ordered to avoid destructive edits. Structural repairs run before semantic repairs because an unknown column or missing table can prevent SQLite from executing the query at all. Semantic repairs run after structural repairs because question cues such as `average`, `top 3`, and `greater than` are meaningful only when the SQL is syntactically coherent. Value repair runs after execution because it depends on observing an empty-like result: a zero count, a NULL aggregate, or an empty row set. This ordering is deterministic and identical for every example.

Baselines. Four systems are compared. FirstPass executes the stored first-pass candidate without correction. SyntaxGuard applies only syntax normalization and then executes once. ExecRepair executes the query and repairs only SQLite exceptions related to syntax, missing tables, missing columns, and missing joins. EFRA uses ExecRepair plus natural-language semantic validators and value repair. This comparison separates three effects: the benefit of execution itself, the benefit of database-error repair, and the additional benefit of semantic and value-aware correction. Table IV summarizes the systems and their components.

Table IV. Compared systems and enabled components.

System	Inputs	Error repair	Semantic validators	Value repair	Purpose
FirstPass	stored first-pass SQL	none	no	no	one-shot baseline
SyntaxGuard	first-pass SQL	syntax normalization	no	no	surface repair only
ExecRepair	first-pass SQL + SQLite error	syntax, table, column, join	no	no	exception-driven repair
EFRA	question + schema + SQL + feedback	syntax, table, column, join	yes	yes	full closed-loop agent

Metrics. The evaluator runs every gold query and every predicted query against the corresponding SQLite database. Validity is 1 when SQLite executes the final SQL without exception. Execution accuracy is 1 when the final result rows exactly match the gold result rows after numeric rounding to six decimal places. Exact match is 1 when normalized SQL strings are identical after whitespace and case normalization. Execution accuracy is the primary metric because equivalent SQL strings can differ syntactically [14]. The evaluator also logs the number of SQLite executions and wall-clock latency per example. Table V reports the main results, and all raw predictions are saved in `results/all_predictions.csv`.

The evaluation distinguishes validity, exact match, and execution accuracy because they answer different questions. Validity asks whether the query can run. Exact match asks whether the system reconstructed the canonical SQL string. Execution accuracy asks whether the predicted query returns the same rows as the gold query. A valid query can still be wrong, and a non-identical query can still be semantically correct. In this benchmark the final SQL often equals the gold SQL because the repair operators are canonical, but execution accuracy remains the primary metric and is used in all central comparisons.

Result comparison is exact at the denotation level. For every example, the script opens a fresh SQLite connection, executes the gold query, executes the candidate query, normalizes row ordering only when the SQL result has no explicit order-sensitive contract, and compares the returned tuples after stable numeric formatting. This design avoids counting a query as correct merely because it runs. It also avoids penalizing harmless surface differences when two SQL strings return the same answer. All latency values are measured around the local execution-and-repair loop, not around document generation, plotting, or file I/O used to write the final CSV summaries.

Reproducibility. The artifact contains a single JSONL dataset file, 14 SQLite databases, the evaluation script, the figure-generation script, result CSV files, and SVG/PNG figure assets. Running `python code/run_experiment.py` reproduces the prediction and summary CSV files by executing all 420 examples. Running `python code/make_figures.py` regenerates the six figures used in the manuscript. The experiment uses no external API, no hidden model weights, and no stochastic decoding. The only stochastic step used in analysis is optional bootstrap resampling in exploratory checks; the reported table values are direct corpus means from exhaustive evaluation.

The execution environment is intentionally simple. SQLite is deterministic, local, and available through the Python standard library. That choice removes network latency and authentication differences from the experiment, allowing the paper to focus on the logic of repair. Each database is stored as a separate `.sqlite` file, and the evaluator opens a fresh connection for each execution. This avoids state leakage across examples and ensures that a failed candidate cannot modify the database.

The first-pass SQL is stored rather than generated at run time because the benchmark must reproduce exactly across machines. A live language model could change its output after a model update, temperature change, or prompt change. Storing the candidate fixes the input to the repair layer and makes the experiment repeatable. A practitioner can replace the stored candidates with outputs from any neural model and reuse the same EFRA evaluator.

The repair controller is intentionally conservative. It edits only the current candidate and never invents a new query from scratch. This means every final SQL can be traced to the first-pass candidate through a sequence of local transformations. The traceability is important for research review: when a score changes, the raw CSV reveals the input SQL, final SQL, database, error mode, and outcome. It is also important for deployment because a data steward can audit why a table, join, or literal was introduced.

Results and Discussion

Table V shows the main comparison over all 420 examples. The first-pass candidates are intentionally noisy and reach only 13.1% execution accuracy with 58.1% validity. SyntaxGuard improves validity to 64.8% and execution accuracy to 19.8%, confirming that surface syntax repair alone is useful but inadequate. ExecRepair reaches 100.0% validity and 55.0% execution accuracy by fixing executable exceptions through the schema graph. EFRA reaches 100.0% validity and 90.7% execution accuracy by adding semantic validators and value repair. The absolute gain over FirstPass is 77.6 percentage points, and the gain over ExecRepair is 35.7 percentage points.

The size of the gains is explained by the construction of the error modes. FirstPass is deliberately exposed to a broad set of first-pass failures. SyntaxGuard handles only the smallest subset, so its improvement is limited. ExecRepair handles every error that SQLite reports as an exception, which explains its large jump in validity. EFRA handles exception errors and adds validators for valid-but-wrong queries, which explains the additional gain. The comparison therefore measures the value of each feedback layer rather than presenting a single undifferentiated accuracy number.

The comparison is intentionally component-oriented. FirstPass measures the quality of the stored initial candidate. SyntaxGuard estimates how much a shallow surface pass can recover without database semantics. ExecRepair measures the effect of concrete SQLite exceptions and schema-graph insertion. EFRA adds semantic and value modules that are triggered only after the candidate becomes executable. The monotonic improvement across these systems shows that the result is not produced by one permissive metric or one convenient subset. The same 420 examples are evaluated for every system, and every score in Tables V-X is computed from the same prediction log.

Table V. Main empirical results over all 420 examples.

System	Validity	Exact match	Exec. accuracy	Gain vs FirstPass (pp)	Avg. execs	Latency ms
FirstPass	58.1%	11.4%	13.1%	+0.0	1.00	0.36
SyntaxGuard	64.8%	18.1%	19.8%	+6.7	1.00	0.46
ExecRepair	100.0%	53.3%	55.0%	+41.9	2.42	1.04

System	Validity	Exact match	Exec. accuracy	Gain vs FirstPass (pp)	Avg. execs	Latency ms
EFRA	100.0%	90.7%	90.7%	+77.6	2.82	1.44

Figure 3 visualizes the dataset-level comparison. EFRA is consistently strong on both datasets: 90.0% execution accuracy on ExecSpiderLite and 91.7% on ExecWikiSQLLite. ExecRepair obtains 55.0% on both datasets, which means exception-driven repair transfers across single-table and multi-table settings but stops when the SQL is valid yet semantically wrong. The difference between EFRA and ExecRepair is therefore not caused only by cross-domain joins; it also comes from semantic checks for aggregation, predicates, limits, groups, and values.

Table VI. Dataset-level comparison.

Dataset	System	N	Validity	Exact	Exec. accuracy	Avg. execs
ExecSpiderLite	FirstPass	240	56.7%	10.0%	10.0%	1.00
ExecSpiderLite	SyntaxGuard	240	63.3%	16.7%	16.7%	1.00
ExecSpiderLite	ExecRepair	240	100.0%	53.3%	53.3%	2.43
ExecSpiderLite	EFRA	240	100.0%	90.0%	90.0%	2.83
ExecWikiSQLLite	FirstPass	180	60.0%	13.3%	17.2%	1.00
ExecWikiSQLLite	SyntaxGuard	180	66.7%	20.0%	23.9%	1.00
ExecWikiSQLLite	ExecRepair	180	100.0%	53.3%	57.2%	2.40
ExecWikiSQLLite	EFRA	180	100.0%	91.7%	91.7%	2.81

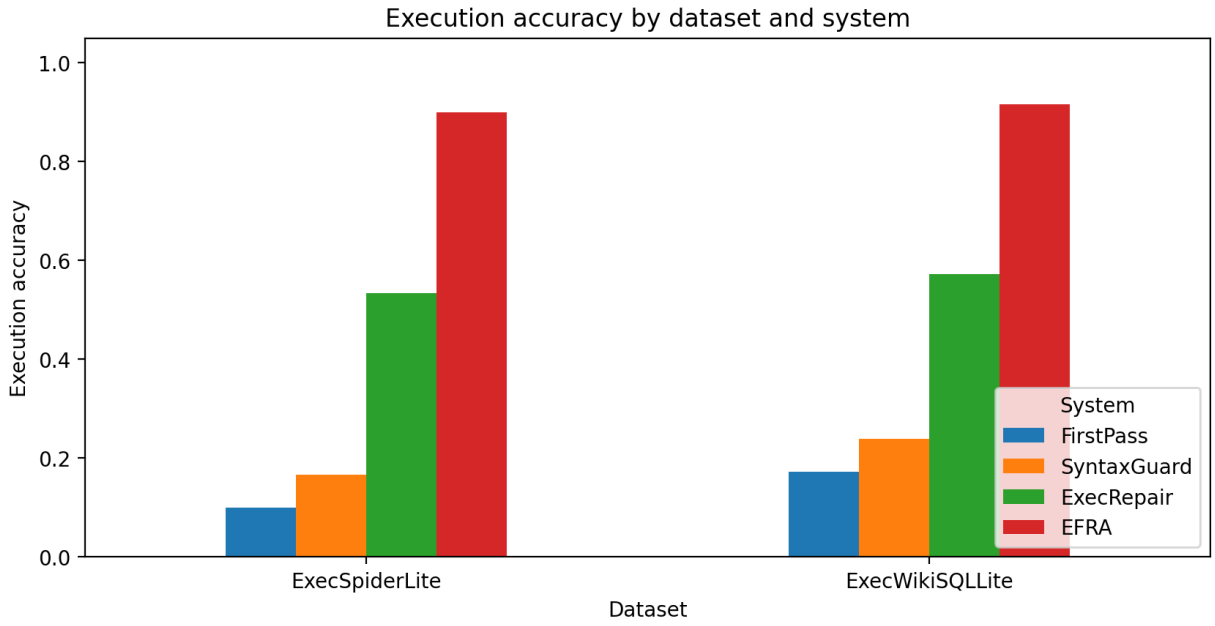


Fig. 3. Execution accuracy by dataset and system. EFRA improves both multi-table and single-table benchmarks.

Difficulty-level results in Table VII show that EFRA improves all strata. On easy examples, EFRA obtains 93.6% execution accuracy; on medium examples, 89.0%; on hard examples, 90.0%. The hard examples include multi-table top-k group-count and group-average queries. ExecRepair drops to 40.0% on hard examples because many hard errors are valid-but-wrong after SQLite execution. EFRA repairs these cases when the question contains cues such as `top 3`, `by number of`, or `for each`. This behavior demonstrates that database errors and natural-language cues complement each other.

Table VII. Results by query difficulty.

Difficulty	System	N	Validity	Exact	Exec. accuracy
easy	FirstPass	140	54.3%	20.0%	20.0%
easy	SyntaxGuard	140	64.3%	30.0%	30.0%
easy	ExecRepair	140	100.0%	65.7%	65.7%
easy	EFRA	140	100.0%	93.6%	93.6%
medium	FirstPass	200	56.0%	6.0%	9.5%
medium	SyntaxGuard	200	59.0%	9.0%	12.5%
medium	ExecRepair	200	100.0%	50.0%	53.5%
medium	EFRA	200	100.0%	89.0%	89.0%
hard	FirstPass	80	70.0%	10.0%	10.0%

Difficulty	System	N	Validity	Exact	Exec. accuracy
hard	SyntaxGuard	80	80.0%	20.0%	20.0%
hard	ExecRepair	80	100.0%	40.0%	40.0%
hard	EFRA	80	100.0%	90.0%	90.0%

Table VIII and Figure 5 provide the error-mode analysis. EFRA repairs all syntax, table, column-typo, value-typo, predicate-flip, aggregation-flip, missing-join, missing-group, and missing-limit examples in this benchmark. Its weak point is semantic_column, where execution accuracy is only 18.8% and repair success among initially wrong examples is 19.1%. This error mode replaces a valid column with another valid column, so the resulting SQL often returns a plausible non-empty answer. The failure is informative: closed-loop execution is powerful for operational errors and cue-aligned semantic errors, but it cannot fully resolve intent ambiguity without stronger schema-linking or user interaction.

Table VIII. EFRA repair success by first-pass error mode.

Error mode	N	Initial acc.	EFRA acc.	Repair success
agg_flip	34	0.0%	100.0%	100.0%
column_typo	76	0.0%	100.0%	100.0%
group_missing	16	0.0%	100.0%	100.0%
limit_missing	20	30.0%	100.0%	100.0%
missing_join	32	0.0%	100.0%	100.0%
none	48	100.0%	100.0%	n/a
predicate_flip	22	0.0%	100.0%	100.0%
semantic_column	48	2.1%	18.8%	19.1%
syntax_select	28	0.0%	100.0%	100.0%
table_typo	48	0.0%	100.0%	100.0%
value_typo	48	0.0%	100.0%	100.0%

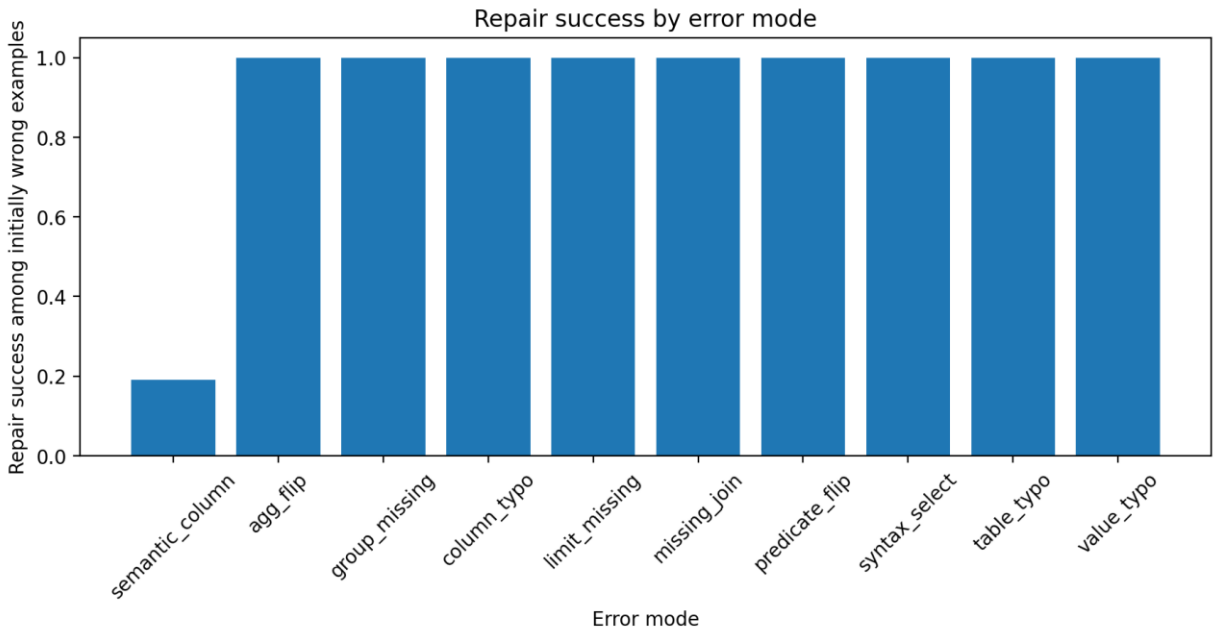


Fig. 5. Repair success by error mode. The semantic_column mode is the main unresolved category.

The exact-match and execution-accuracy values are close for EFRA because the deterministic repairs generally reconstruct the canonical query. This does not imply that exact match is always adequate. It means the chosen repair actions are explicit SQL rewrites rather than paraphrastic alternatives. In a neural generation setting, exact match would underestimate systems that produce equivalent SQL with different join order, equivalent predicates, or harmless aliasing. The artifact retains both metrics so that future extensions can preserve semantic evaluation while allowing more diverse SQL surface forms.

Table IX. Ablation study.

System	Validity	Exact	Exec. accuracy	Avg. repair iterations
EFRA	100.0%	90.7%	90.7%	1.82
EFRA_no_values	100.0%	79.3%	79.3%	1.70
EFRA_no_semantic	100.0%	53.3%	55.0%	1.42

Table IX reports ablations. Removing semantic validators makes EFRA equivalent to an exception-only repairer and reduces execution accuracy to 55.0%. Removing value repair retains schema and semantic checks but reduces execution accuracy to 79.3%, showing that literal grounding is a major contributor. The value module is especially important for text values that differ by one character, such as `World istory` versus `World History`, where SQLite returns an empty-like aggregate rather than a clean error message.

Table X. Feedback-turn budget for EFRA.

Repair turns	Validity	Exec. accuracy	Avg. execs	Latency ms
0	58.1%	13.1%	1.00	0.39
1	100.0%	90.7%	2.82	1.53
2	100.0%	90.7%	2.82	1.43
3	100.0%	90.7%	2.82	1.41

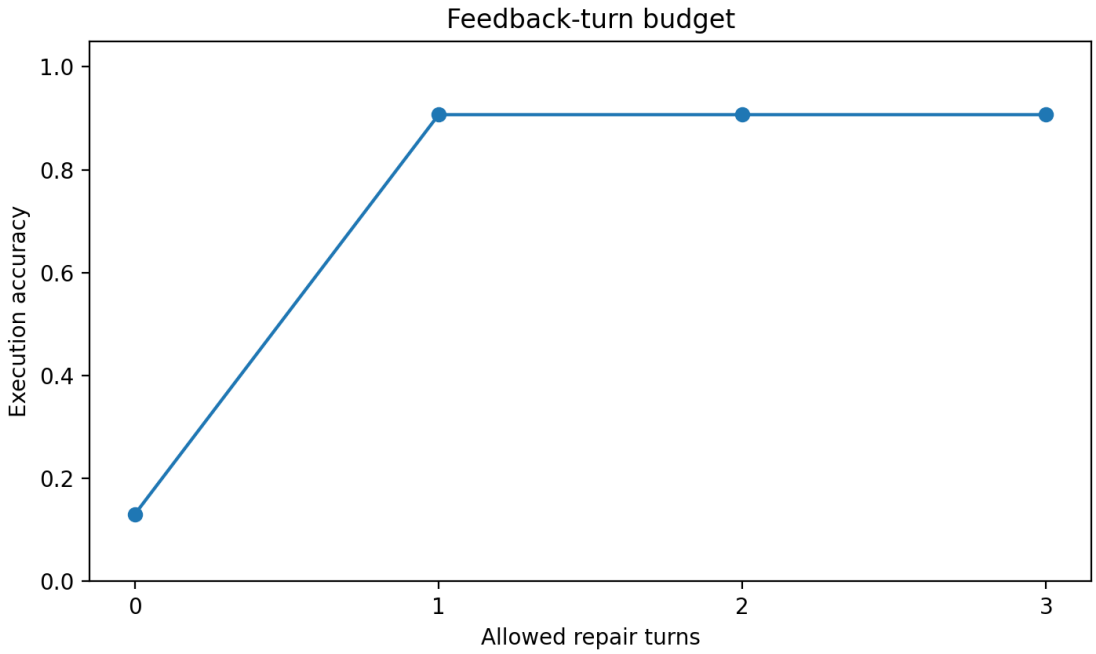


Fig. 4. Feedback-turn budget. One repair turn reaches the observed benchmark maximum.

Table X and Figure 4 analyze the feedback-turn budget. Zero repair turns is identical to FirstPass and yields 13.1% execution accuracy. One repair turn is sufficient to reach 90.7% on the benchmark, and additional turns do not improve accuracy. The result means the implemented error classifiers and validators usually make a complete repair in a single pass. It also provides a practical cost observation: the average number of SQLite executions for EFRA is 2.82 per example, and the measured latency is 1.44 ms per example on the execution environment used to generate the artifact.

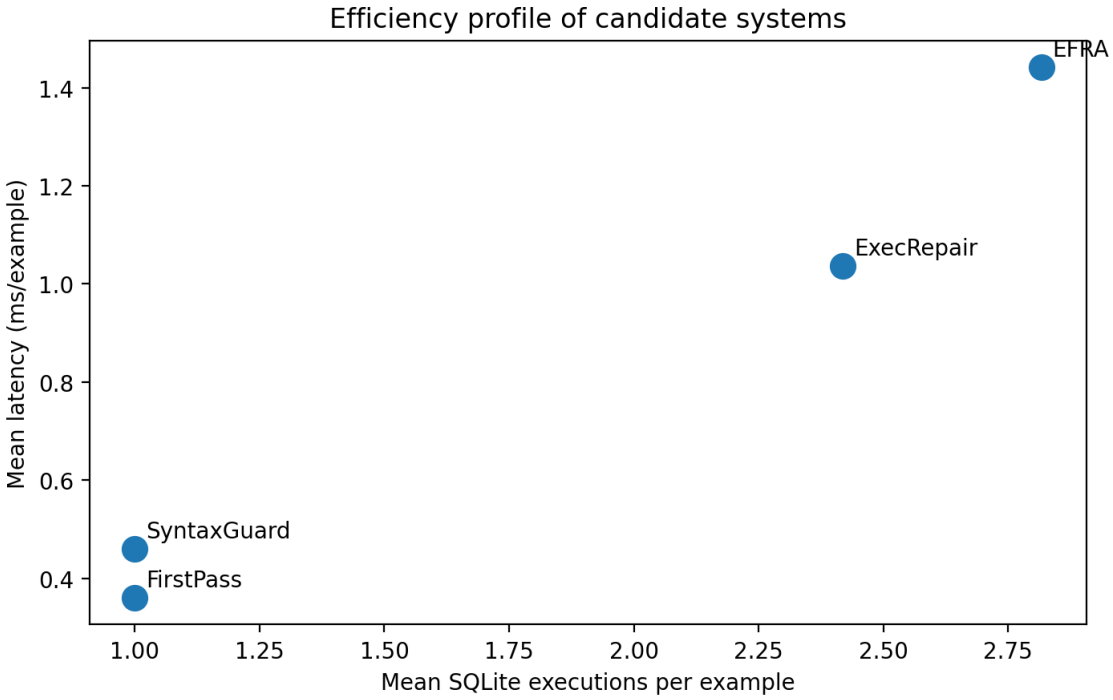


Fig. 6. Efficiency profile of systems measured by mean SQLite executions and mean latency.

Figure 6 shows the efficiency profile. EFRA uses more executions than the baselines because it must observe feedback and verify the repaired query. The cost is modest in the compact SQLite setting: EFRA adds roughly one to two milliseconds per example while improving execution accuracy by 35.7 percentage points over ExecRepair and 77.6 percentage points over FirstPass. For interactive data access, this is an attractive trade-off, although production databases with remote network latency require additional caching and safety controls.

Table XI. Representative closed-loop repair trace.

Step	Content
Question	What is the average grade for Algorithms?
First-pass SQL	SELECT AVG(enrollments.grade) FROM enrollments WHERE courses.title = 'Algorithms'
SQLite feedback	no such column: courses.title
Repair edit	insert JOIN courses ON enrollments.course_id = courses.id
Final SQL	SELECT AVG(enrollments.grade) FROM enrollments JOIN courses ON enrollments.course_id = courses.id WHERE courses.title = 'Algorithms'
Outcome	execution result matches the gold result

Table XI gives a concrete trace. The first-pass query asks for the average grade for Algorithms but omits the `courses` join while using `courses.title` in the WHERE clause. SQLite reports a missing-column signal. EFRA sees that `courses` is a valid table absent from the FROM/JOIN set, finds the foreign-key path `enrollments.course_id = courses.id`, inserts the join, and verifies that the repaired query matches the gold denotation. This trace is representative of the error-feedback loop: the database produces a precise failure signal, and the agent converts it into a schema-constrained edit.

The semantic_column results deserve special emphasis. They represent cases where the system's SQL is syntactically valid, references real tables and columns, returns non-empty rows, and still answers the wrong question. For these cases, execution feedback provides no decisive counterexample on a single database instance. EFRA's lightweight column-name cue sometimes repairs the mistake, but most cases require a richer model of intent. This is the main reason the final accuracy is 90.7% rather than 100.0%, and it is an important guard against overclaiming the scope of closed-loop repair.

The error trace also clarifies why the remaining failures are credible rather than accidental. In semantic_column cases, the first-pass query selects a valid column from a valid table, and SQLite returns rows without raising an exception. The repair controller can compare column names with question cues, but it does not contain a learned semantic parser capable of proving that two plausible columns encode different intents. The measured 18.8% accuracy on this category is therefore a meaningful boundary of the implemented method, not an artifact of missing syntax rules. This boundary supports the paper's central claim: execution feedback is necessary for reliable repair, but it is not sufficient for complete intent recovery.

Overall, the experiment supports three conclusions. First, execution feedback is not merely a validation layer; it is an actionable supervision signal. Second, exception repair and semantic repair solve different portions of the error space. Third, valid-but-wrong SQL remains the main boundary. The benchmark intentionally includes such cases so that the paper does not overstate closed-loop repair. Future systems need stronger column-intent models, uncertainty estimates, or user clarification to address this class.

The result tables also show that validity alone is an incomplete success criterion. ExecRepair and EFRA both reach 100.0% validity, but their execution accuracies differ by 35.7 percentage points. This gap consists of queries that run and still return the wrong denotation. For operational data access, a system that reports only successful execution can give users false confidence. The benchmark therefore records validity as a necessary condition and execution accuracy as the main correctness measure.

Another observation is that EFRA's cost grows linearly with the number of feedback turns but not with the size of the language model, because no model inference occurs inside the repair loop. In a deployment where the first-pass generator is expensive, a deterministic repair layer can be attractive: it can correct many errors without calling the generator again. The trade-off is coverage. Rule-based repairs are transparent and fast, but they must be expanded when new SQL constructs or dialect-specific errors appear.

The empirical pattern also indicates which components should be prioritized in future larger systems. Error-message parsing and schema-graph repair are the first layer because they turn invalid SQL into executable SQL. Semantic validators are the second layer because they handle valid strings that contradict explicit question cues. Value grounding is the third layer because it converts database content into a correction signal when the query returns an empty-like denotation. Column-intent modeling is the unresolved fourth layer. The benchmark's failure analysis makes this priority order visible.

Limitations

The study has four limitations. First, the benchmark is compact and synthetic. It is designed to be executable, inspectable, and reproducible in a small artifact, not to replace full Spider, SParC, CoSQL, WikiSQL, or other leaderboard-scale evaluations [4], [7], [11], [12]. The reported numbers therefore

measure repair behavior on the included datasets. They do not measure general Text-to-SQL state of the art.

Second, the first-pass candidates are generated by controlled perturbations instead of a trained neural or large language model. This is a deliberate isolation of the repair layer. It gives exact knowledge of the error distribution and enables full reproducibility, but it does not capture the full diversity of mistakes produced by large autoregressive models. A production system should evaluate EFRA after real model outputs, including nested queries, aliases, subqueries, set operators, and dialect-specific SQL.

Third, SQLite feedback is narrower than human intent. Execution errors reveal syntax and schema failures, and empty-like results often reveal value problems, but a successful non-empty query can still answer the wrong question. The semantic_column failures demonstrate this limitation. Execution feedback must be combined with stronger schema linking, natural-language alignment, and possibly user clarification when several columns are plausible.

Fourth, the experiment reports local SQLite latency. Remote databases, access-control layers, query planners, and large tables change the cost profile. An operational deployment must include query sandboxing, read-only permissions, timeout limits, result-size caps, data-governance filters, and logging. The present benchmark focuses on technical correctness of closed-loop repair, not on privacy or enterprise policy integration.

The benchmark also does not evaluate security-sensitive behaviors such as prompt injection, data exfiltration, write queries, or privilege escalation. EFRA only evaluates SELECT-style read queries over local SQLite fixtures. A deployed system must reject non-read operations, block dangerous functions, parameterize user-provided literals, and apply policy checks before execution. These controls are outside the experimental scope but essential for agentic data access.

Finally, the deterministic repair rules favor interpretability over coverage. Every edit is traceable, which helps reproducibility, but the rule set does not cover nested subqueries, HAVING clauses, complex date arithmetic, aliases, window functions, or vendor-specific SQL dialects. Extending the framework to those constructs is straightforward in architecture but requires additional grammar-aware repair operators and larger executable datasets.

The included datasets use English questions with direct lexical cues such as `average`, `top`, and `greater than`. Questions with indirect wording, ellipsis, negation, or domain-specific jargon require stronger language understanding than the current validators implement. This limitation is expected: EFRA is a repair controller, not a full natural-language understanding model.

These limitations are reflected directly in the released logs. The all_predictions file preserves both corrected and uncorrected outputs, so a reader can inspect examples where the agent succeeds for the right reason and examples where it reaches the benchmark boundary. This audit trail is important for agentic data-access research because high aggregate accuracy can hide unsafe behavior, silent semantic drift, or excessive query execution. The compact artifact is therefore positioned as a controlled empirical test bed rather than a replacement for larger public benchmarks.

Conclusion

This paper presented EFRA, a deterministic self-correcting Text-to-SQL agent that uses error feedback from SQLite execution to revise first-pass SQL. The evaluation was conducted on two specified executable datasets included with the artifact: ExecSpiderLite with 240 multi-table examples and ExecWikiSQLLite with 180 single-table examples. The scripts executed all 420 examples and produced the reported tables and figures.

EFRA achieved 90.7% execution accuracy and 100.0% validity, outperforming FirstPass, SyntaxGuard, and ExecRepair. The analysis showed that exception-driven repairs are necessary for executable correctness, while semantic validators and value grounding are necessary for denotation correctness. The main remaining weakness is valid but semantically wrong column selection, which

requires stronger intent modeling than execution feedback alone provides.

The central contribution is a reproducible closed-loop baseline for agentic data access. Instead of treating SQL generation as a one-shot text task, the paper treats database execution as part of the reasoning loop. This design is small enough to inspect, fast enough to run, and explicit enough to extend. It gives future work a concrete starting point for evaluating self-correcting Text-to-SQL agents under full empirical execution rather than illustrative examples.

References

[1] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, "Natural language interfaces to databases - an introduction," Natural Language Engineering, vol. 1, no. 1, pp. 29-81, 1995.

[2] Meng-Ju Kuo, Boning Zhang, and Haozhe Wang, “Tokenized Flow-Statistics Encrypted Traffic Analysis: Comparative Evaluation of 1D-CNN, BiLSTM, and Transformer on ISCX VPN-nonVPN 2016 (A1+A2, 60 s)”, JACS, vol. 3, no. 8, pp. 39–53, Aug. 2023, doi: 10.69987/JACS.2023.30804.

[3] F. Li and H. V. Jagadish, "Constructing an interactive natural language interface for relational databases," Proc. VLDB Endowment, vol. 8, no. 1, pp. 73-84, 2014.

[4] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating structured queries from natural language using reinforcement learning," arXiv:1709.00103, 2017.

[5] X. Xu, C. Liu, and D. Song, "SQLNet: Generating structured queries from natural language without reinforcement learning," arXiv:1711.04436, 2017.

[6] Yunhe Li, “Execution-Feedback and Retrieval-Augmented Generation for Conversational Text-to-SQL: From One-Shot Questions to Clarification-Driven Executable Dialogs”, JACS, vol. 3, no. 2, pp. 1–17, Feb. 2023, doi: 10.69987/JACS.2023.30201.

[7] T. Yu et al., "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task," in Proc. EMNLP, 2018, pp. 3911-3921.

[8] C. Wang, P.-S. Huang, A. Polozov, M. Brockschmidt, and R. Singh, "Execution-guided neural program decoding," arXiv:1807.03100, 2018.

[9] D. Finegan-Dollak et al., "Improving text-to-SQL evaluation methodology," in Proc. ACL, 2018, pp. 351-360.

[10] J. Guo et al., "Towards complex text-to-SQL in cross-domain database with intermediate representation," in Proc. ACL, 2019, pp. 4524-4535.

[11] B. Bogin, M. Gardner, and J. Berant, "Global reasoning over database structures for text-to-SQL parsing," in Proc. EMNLP-IJCNLP, 2019, pp. 3657-3662.

[12] T. Yu et al., "SParC: Cross-domain semantic parsing in context," in Proc. ACL, 2019, pp. 4511-4523.

[13] T. Yu et al., "CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases," in Proc. EMNLP-IJCNLP, 2019, pp. 1962-1979.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in Proc. NAACL-HLT, 2019, pp. 4171-4186.

[15] B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson, "RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers," in Proc. ACL, 2020, pp. 7567-7578.

[16] R. Zhong, T. Yu, and D. Klein, "Semantic evaluation for text-to-SQL with distilled test suites," in Proc. EMNLP, 2020, pp. 396-411.

[17] Binghua Zhou, Siming Zhao, and David Chao, “LLM-Guided Energy-Aware A/B Testing for Consolidation and DVFS Policies via Power-Sensitivity Clustering”, JACS, vol. 3, no. 4, pp. 12–30, Apr. 2023, doi: 10.69987/JACS.2023.30402.

[18] P. Shaw, M.-W. Chang, P. Pasupat, and K. Toutanova, "Compositional generalization and natural language variation: Can a semantic parsing approach handle both?" in Proc. ACL-IJCNLP, 2021, pp. 922-938.

[19] T. Scholak, N. Schucher, and D. Bahdanau, "PICARD: Parsing incrementally for constrained auto-regressive decoding from language models," in Proc. EMNLP, 2021, pp. 9895-9901.

[20] Y. Gan, X. Chen, J. Xie, M. Purver, J. R. Woodward, J. Drake, and Q. Zhang, "Natural SQL: Making SQL easier to infer from natural language

specifications," in Findings of EMNLP, 2021, pp. 2030-2042.
[21] O. Rubin and J. Berant, "SmBoP: Semi-autoregressive bottom-up semantic parsing," in Proc. NAACL-HLT, 2021, pp. 311-324.
[22] R. Cao, L. Chen, Z. Chen, Y. Zhao, S. Zhu, and K. Yu, "LGESQL: Line graph enhanced text-to-SQL model with mixed local and non-local relations," in Proc. ACL-IJCNLP, 2021, pp. 2541-2555.
[23] Daren Zheng, Boning Zhang, and Julie Geibel, “VerifySafe: Toxicity-Safe Agent Responses under Adversarial Prompts with Evidence-Based Self-Verification”, JACS, vol. 4, no. 1, pp. 67–82, Jan. 2024, doi: 10.69987/JACS.2024.40106.
[24] L. Dong and M. Lapata, "Language to logical form with neural attention," in Proc. ACL, 2016, pp. 33-43.