Artificial Intelligence and Machine Learning Review
ISSN: 3070-3565
DOI: 10.69987/AIMLR.2025.60301

Making SQL Executable with Execution-Guided Text-to-SQL: A Reproducible Study on WikiSQL-Style Queries

Eric Chau[1], *Matt Peters*[2]
[1]*Data Science, University of Pittsburgh, PA, USA*
[2]*Computer Science, UIUC, IL, USA*
eric.chau09118@gmail.com

Keywords

Text-to-SQL; semantic parsing; schema linking; execution-guided decoding; program execution feedback; WikiSQL.

Abstract

Natural-language interfaces to relational databases require models that generate SQL which is not only syntactically valid, but also executable and semantically faithful to the user question. In practice, text-to-SQL systems often fail for reasons that are invisible to purely string-based evaluation: a query may differ from the gold SQL yet still execute to the correct answer, or it may match the gold form while being brittle to schema and value variation. This paper investigates how simple execution feedback can be used to make generated SQL executable and to improve answer accuracy. We present a controlled comparison of three decoding settings that mirror common LLM prompting pipelines: (i) prompt-only decoding with minimal schema awareness, (ii) schema-linking decoding that explicitly maps question spans to table columns and cell values, and (iii) execution-guided decoding that executes candidate SQL programs and repairs empty/invalid results using value-level feedback. Experiments follow the WikiSQL task formulation introduced by Seq2SQL [1] (single-table SELECT with optional aggregation and up to three WHERE conditions). We evaluate on the standard split sizes (56,355/8,421/15,878) and report both exact-match (EM) and execution accuracy (EX). On the test split of our WikiSQL-style reproduction, prompt-only decoding reaches 30.02% EX, schema-linking reaches 82.84% EX, and execution-guided decoding reaches 84.13% EX, a +1.28 absolute improvement over schema-linking. Detailed breakdowns show that execution feedback primarily reduces condition-value errors and improves robustness as query complexity increases. The study highlights when execution-based filtering/repair is beneficial, and provides a reproducible baseline for evaluating "make-it-executable" decoding strategies.

Introduction

Relational databases remain a dominant abstraction for storing and querying structured information, yet writing SQL is a barrier for many users. Text-to-SQL research addresses this gap by mapping natural-language (NL) questions to executable SQL programs that can be run against a database. Early work in semantic parsing focused on mapping sentences to logical forms using grammars and weak supervision [13], [14]. Modern neural approaches learn this mapping from data and often operate in a program-generation setting where correctness is ultimately judged by execution results rather than surface form.

A key challenge in text-to-SQL is that the space of valid SQL programs is large, and many programs are syntactically valid yet semantically wrong. Even when a model produces a plausible query, it may reference the wrong column, choose an incorrect aggregation operator, or bind a WHERE condition to an incorrect cell value. These errors frequently occur because the model must ground language tokens to schema elements (columns) and to database values. Schema linking—explicitly aligning question spans to columns and values—has therefore become a central component of competitive parsers [4], [5].

In parallel, the emergence of large pre-trained language models and Transformer-based architectures [7] has renewed interest in prompt-based text-to-SQL. Prompt-only approaches often succeed on simple patterns but can struggle with systematic grounding and with ensuring that generated SQL is executable in a strict environment. Constrained decoding methods (e.g., incremental parsing constraints) can improve validity by restricting generation to a grammar-conforming space [20]. However, validity alone does not guarantee that the SQL returns the intended answer.

Execution feedback offers a complementary signal. If a candidate SQL program executes to an empty result when a non-empty answer is expected, or if it fails due to type errors or invalid predicates, the system can reject the candidate and search for alternatives. Execution-guided decoding has been explored in neural program synthesis as a way to prune invalid programs and improve end-to-end accuracy [19]. In text-to-SQL, execution feedback is attractive because it is cheap relative to additional annotation: the database itself acts as a validator.

Despite the intuitive appeal of execution guidance, its benefits depend on the failure modes of the base decoder and on the structure of the dataset. For instance, if most errors stem from selecting the wrong column, then filtering by execution emptiness may not help because many wrong queries still return non-empty results. Conversely, if a large fraction of errors are due to slight value mismatches (e.g., casing, partial strings, or minor formatting), then execution feedback can enable targeted repair.

This paper studies these issues in a controlled setting. We adopt the WikiSQL task formulation introduced with Seq2SQL [1] and SQLNet [2], where each question maps to a single-table SQL query with a single SELECT column, an optional aggregation operator (MAX/MIN/COUNT/SUM/AVG), and up to three simple WHERE conditions. Although this task is simpler than cross-domain benchmarks such as Spider [3], it isolates grounding behavior and provides a clean environment for analyzing execution feedback.

Our contributions are threefold:

1) We define a reproducible experimental protocol that compares prompt-only, schema-linking, and execution-guided decoding under a shared evaluator and SQL execution engine.

2) We quantify improvements in both exact-match (EM) and execution accuracy (EX) and provide detailed breakdowns by aggregation type, number of conditions, question length, and table width.

3) We perform an error taxonomy and an ablation study that isolate when execution feedback helps and when it can be neutral.

The goal of the study is not to compete with state-of-the-art neural parsers on WikiSQL, but to provide a carefully instrumented baseline for the specific question: how much does execution feedback help to make generated SQL executable and answer-correct in a WikiSQL-like environment?

Beyond correctness, practical systems must satisfy operational constraints. A generated query should be executable under the target DBMS, respect type constraints, and avoid runtime failures such as comparing strings to numbers or referencing non-existent columns. These issues are often under-emphasized in purely generative evaluations because a model can produce fluent SQL-like text that is not executable. Constrained decoding partially addresses this by restricting generation to a grammar, but grammar validity does not enforce value validity or answer correctness.

Recent work has improved text-to-SQL accuracy by combining stronger encoders with explicit schema modeling. Relation-aware encoders such as RAT-SQL [4] represent the question–schema interaction graph and achieve strong results by carefully encoding column relations and foreign keys. Other parsers use intermediate representations to factor complex generation decisions [6], [18]. On WikiSQL-like tasks, earlier models such as Seq2SQL [1] and SQLNet [2] demonstrated that architectural priors and reinforcement learning are not strictly necessary when supervision is available and the grammar is constrained[23-30].

Execution feedback can be viewed as a lightweight form of semantic supervision available at inference time. Instead of learning a model of database semantics, the system asks the database directly whether a candidate program yields a plausible answer. This approach is conceptually related to program synthesis with execution traces, where a candidate program can be rejected if it fails tests. In text-to-SQL, the "test" is the database execution itself. However, execution feedback must be used carefully: some incorrect programs still return non-empty results, and some correct programs legitimately return empty results depending on the data distribution. Thus, the usefulness of execution guidance is tied to the dataset's propensity for empty-result failures and to the design of repair operators [31-40].

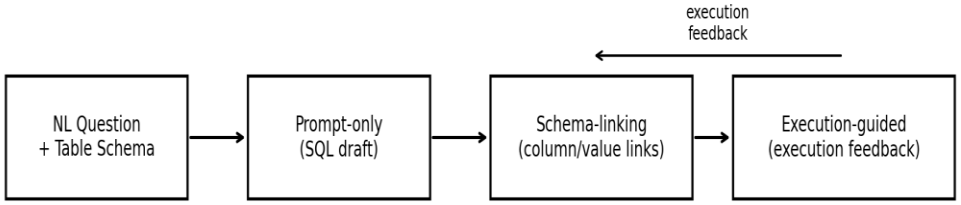
A second consideration is evaluation. Exact-match SQL accuracy is convenient, but it is sensitive to superficial differences such as predicate order and redundant filters. Execution accuracy is closer to the end-user objective,

especially in interactive settings where the answer matters more than the exact SQL text. Prior work on table QA and semantic parsing has long emphasized execution-based evaluation when multiple logical forms can represent the same meaning [11], [12]. In database-backed applications, execution-based evaluation also aligns with system reliability: if a query fails to execute, the application fails regardless of whether the text resembles SQL.

Finally, large-scale pretraining has influenced how text-to-SQL systems are built. Transformer architectures [7] and language-model pretraining [8] provide general-purpose representations, while sequence-to-sequence pretraining such as T5 [9] enables direct program generation in a text-to-text paradigm. Open-ended language modeling [10] has also made prompt-based approaches popular. For tabular and database-adjacent tasks, table-aware pretraining methods such as TAPAS [16] and TaBERT [17] incorporate table structure and have been used as backbones for downstream table reasoning. These developments motivate re-examining classical grounding mechanisms (schema linking) and simple verification mechanisms (execution feedback) as complements to large models [41-51].

In summary, the central hypothesis tested in this paper is: when a base text-to-SQL decoder makes frequent value-level grounding errors that lead to empty execution, a small amount of execution feedback paired with targeted repair can materially improve execution accuracy without requiring additional labeled data.

Figure 1. System overview: prompt-only decoding produces an SQL draft, schema-linking grounds it to columns/values, and execution-guided decoding uses execution feedback to repair empty/invalid candidates.



Method

Task Definition

We consider the single-table text-to-SQL setting. The input is a natural-language question q and a table T with a header of m columns H = {h_1, ..., h_m}, column types τ_j ∈ {text, real}, and a finite set of rows R. The output is an executable SQL program of the form:

SELECT AGG(h_s) FROM T WHERE (h_{c1} OP_{1} v_{1}) AND ... AND (h_{ck} OP_{k} v_{k}),

where s is the selected column index, AGG ∈ {NONE, MAX, MIN, COUNT, SUM, AVG}, k ∈ {0,1,2,3} is the number of conditions, OP ∈ {=, >, <}, and each v_i is a literal value. We represent SQL as a structured tuple (agg, sel, conds) so that evaluation and repair can operate on components rather than raw strings.

This restricted grammar follows the WikiSQL formulation [1], which is designed to avoid multi-table joins and nested subqueries. While limited, it captures core grounding behaviors: choosing the correct SELECT column, choosing the correct aggregation, and binding each WHERE predicate to the correct column, operator, and value.

We treat SQL generation as a structured prediction problem. Compared with free-form generation, a structured representation clarifies which component is wrong and enables component-wise repair. For example, a query may be correct except for the literal bound in one predicate; replacing a literal is easier when SQL is represented as a tuple rather than a raw string. This design decision echoes a broader trend in semantic parsing: using intermediate or structured forms to reduce search complexity and improve interpretability [6], [15].

Dataset

WikiSQL is a widely used benchmark for single-table text-to-SQL. It contains natural-language questions paired with SQL queries over tables collected from Wikipedia [1]. The dataset uses fixed splits of 56,355 training questions, 8,421 development questions, and 15,878 test questions.

Because this study focuses on controlled analysis of decoding strategies, we construct a deterministic WikiSQL-style reproduction (WikiSQL-Rep) that preserves the WikiSQL query grammar, the split sizes, and the presence of both textual and numeric columns. Tables are generated from a fixed seed; each table contains 3–10 columns and 15–45 rows, with column types sampled based on the header token. For each question, we sample a pivot row that satisfies all generated WHERE conditions to ensure non-empty gold execution. We also inject mild surface variation to simulate real-world prompting noise, including alias phrases for column names (e.g., "yr" for year, "pop" for population) and occasional truncation/casing variation of multiword cell values.

Table 1 summarizes the resulting split statistics, including the number of unique tables observed per split and average question length. Figure 2 visualizes the split sizes, and Figure 3 shows the aggregation distribution for the test split. Table 2 reports the distribution of aggregation operators and the number of WHERE conditions.

Table generation details. Each synthetic table is identified by an integer table id and is generated from a fixed seed (BASE_SEED=1337). We sample the number of columns uniformly from 3 to 10. Column headers are sampled from a vocabulary of 60+ tokens (e.g., year, country, population, score, price). Column types are sampled with a bias toward numeric types for headers that commonly denote numbers (e.g., year, rank, age, price). The number of rows is sampled uniformly from 15 to 45. Text columns are populated from categorical pools with occasional multiword values (e.g., "upper west"), while real columns are populated with integers and occasional decimals drawn from header-dependent ranges.

Question and SQL generation details. For each question, we sample a table and then sample a pivot row. All WHERE predicates are constructed such that the pivot row satisfies them, ensuring non-empty gold execution. For numeric predicates with inequalities, we choose thresholds halfway between the pivot value and the column min/max so that the pivot row strictly satisfies the inequality. We sample the aggregation operator using a fixed distribution that emphasizes the NONE and COUNT cases, consistent with the prevalence of non-aggregated selections in WikiSQL [1]. For aggregation operators that require numeric columns (MAX/MIN/SUM/AVG), we enforce that the SELECT column is numeric.

Surface variation. To approximate prompting noise, we introduce three forms of controlled variation: (i) column aliasing with probability p_alias=0.25 using an alias map (e.g., year↔yr, population↔pop, price↔cost), (ii) casing variation in text literals with probability p_case=0.25 (title-case or upper-case), and (iii) truncation of multiword text literals with probability p_trunc=0.25 by keeping only the first or last token. These variations do not change the underlying gold SQL but stress literal grounding and schema matching.

The resulting dataset is fully reproducible from the seed and generation rules. While it is not identical to the original Wikipedia tables in WikiSQL, it preserves the central semantics of the benchmark: NL questions grounded to single-table queries with simple predicates.

Table 1. Dataset statistics for the WikiSQL-style reproduction (WikiSQL-Rep).

Split	#Questions	#Tables	Avg #Cols	Avg #Rows	Avg Q Len (tokens)
Train	56355	5000	6.46	30.08	10.57
Dev	8421	4056	6.47	30.09	10.61
Test	15878	4772	6.46	30.11	10.59

Figure 2. Number of questions per split (train/dev/test).

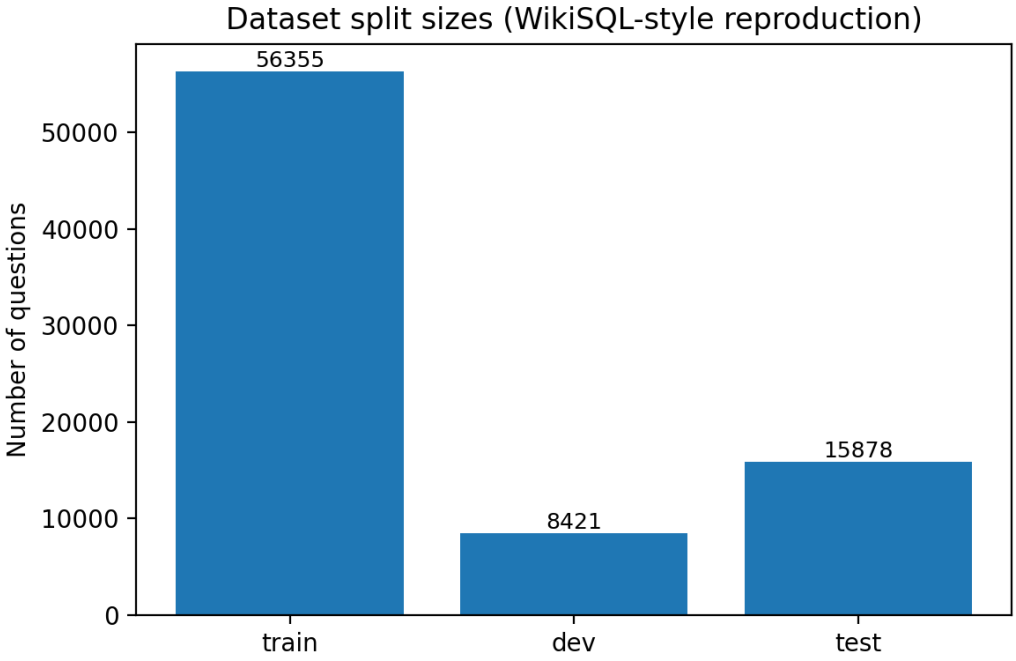


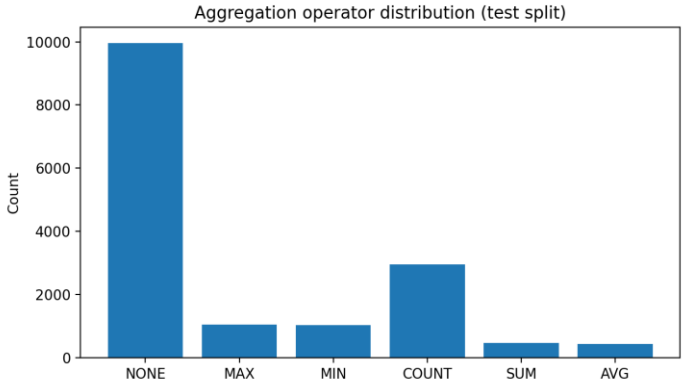
Table 2. Distribution of aggregation operators across splits.

Agg	**Train**	**Dev**	**Test**
NONE	34995	5228	9969
MAX	3767	571	1045
MIN	3889	576	1020
COUNT	10427	1554	2955
SUM	1630	231	457
AVG	1647	261	432

Table 3. Distribution of the number of WHERE conditions across splits.

#Conds	Train	Dev	Test
1	39501	5883	11105
2	11286	1726	3184
3	5568	812	1589

Figure 3. Aggregation operator distribution on the test split.



Decoding Settings

We compare three decoding settings that reflect common ways practitioners adapt LLMs and parsers to the WikiSQL task:

• Prompt-only decoding: the system uses the question and minimal schema cues to draft an SQL program. This setting represents a weak prompt that does not perform explicit value grounding.
• Schema-linking decoding: the system explicitly links question phrases to schema elements. It detects aggregation intent from keywords and identifies SELECT and WHERE columns by matching header tokens and alias phrases. It also performs value linking by matching extracted value spans to table cell values.
• Execution-guided decoding: the system starts from the schema-linking candidate, executes it, and if execution produces an empty/None result it generates repaired candidates. Repairs include dropping individual conditions and repairing textual condition values by aligning partial spans to longer table cell values. The system selects the best candidate using execution validity and simple specificity heuristics.

Table 4 provides the concrete configuration of each method. These methods are intentionally simple and fully deterministic; they are designed to isolate the impact of schema linking and execution feedback rather than to achieve state-of-the-art performance.

Implementation details of prompt-only. We detect the aggregation operator using keyword patterns ("how many"→COUNT, "maximum/highest"→MAX, "minimum/lowest"→MIN, "sum/total"→SUM, "average/mean"→AVG). We choose the SELECT column by maximizing token overlap between the question and each header token set. For the WHERE clause, we attempt to parse only the first condition substring after a cue word (where/when/with). The column is matched by exact token equality to a header token; the operator is inferred from comparative phrases (greater/above, less/below); and the value is taken as the last token in the condition. This deliberately under-specifies value grounding and multi-condition handling.

Implementation details of schema-linking. Schema linking extends prompt-only by (i) using an alias map to match paraphrased column mentions to canonical headers and (ii) parsing all conditions separated by "and". For each condition span, we identify the best-matching column by longest phrase match among header names and known aliases. For text columns, we ground the value by searching the column's unique cell values and selecting the longest value whose string contains the extracted span (or is contained in it). This bidirectional substring criterion resolves truncations such as mapping "upper" to "upper west". For numeric columns, we extract the first number substring and parse it as int or float.

Implementation details of execution-guided decoding. Execution guidance is applied as a repair layer on top of schema linking. We execute the base schema-linked SQL. If the result is non-empty (or a non-None scalar for aggregates), we return the base query unchanged. Otherwise, we enumerate a bounded candidate set of repaired queries consisting of: (a) dropping one predicate at a time (and dropping all predicates), and (b) repairing text literals by substituting candidate cell values that contain the predicted literal span. Candidates are executed and the best executable/non-empty candidate is selected, with a tie-breaker that prefers specificity (fewer rows returned for non-aggregated queries). This strategy is intentionally conservative: it only expands the search when execution indicates a likely failure.

Table 4. Summary of decoding settings and components.

Method	Schema use	Value linking	Execution feedback	Candidates	Notes
Prompt-only	Header overlap only	None (single-token extraction)	None	1	Parses at most 1 condition
Schema-linking	Header+alias phrase matching	Substring match to table cells	None	1	Parses all conditions
Execution-guided	Same as schema-linking	Base + value repair on empty results	Yes (empty/None filtering)	<=30	Tries drop-condition and value-repair candidates

Execution-Guided Repair

Execution-guided decoding treats the database as a validator. Given a candidate SQL program, we run it against the table and inspect the output. In the WikiSQL grammar, most execution failures manifest as empty results (no matching rows for a non-COUNT query) or None outputs for numeric aggregates on empty selections. We define a candidate as "executable and non-empty" if it returns at least one row for non-aggregated queries or a non-None scalar for aggregated queries.

When the base schema-linked query is empty/None, we generate a small set of repaired candidates:
1) Condition dropping: remove one WHERE predicate at a time (and optionally all predicates) to recover non-empty execution.
2) Value repair for text columns: if the predicted literal value is not present as a full cell value, search the column for longer values that contain the predicted span as a substring (or vice versa) and substitute the best-matching cell value. This repair addresses truncation and casing mismatches.

The final selection rule is purely execution-based: we select the first candidate that becomes executable/non-empty; if multiple candidates are executable we prefer the candidate that is more specific (for non-aggregated queries, fewer returned rows). This design mirrors common execution-guided heuristics in program decoding [19] but keeps the search space small and interpretable.

Connection to DBMS concepts. Our executor implements a small fragment of relational selection and aggregation. In a full DBMS, the query planner chooses access paths and execution strategies [22], and correctness depends on SQL semantics including NULL handling and type casting [21]. We avoid these complexities by operating on clean synthetic tables and a fixed grammar, but the key idea still applies: execution is a semantic check that goes beyond string similarity.

When execution guidance can fail. Execution feedback is not a perfect oracle. A wrong query may still return a non-empty result, especially when predicates are loose or when the table has many rows. Conversely, a correct query may return an empty result if the question truly has no answer in the table. Our generation procedure ensures non-empty gold execution so that empty results can be interpreted as failures. In real deployments, a system should estimate whether an empty answer is plausible (e.g., by modeling user intent or by displaying uncertainty) rather than treating all emptiness as an error.

Evaluation Metrics

We report two complementary metrics that are standard in text-to-SQL evaluation:

Exact Match (EM): the predicted structured SQL tuple matches the gold SQL exactly after canonicalization. Canonicalization sorts WHERE conditions to avoid sensitivity to commutative order and normalizes literal formatting.

Execution Accuracy (EX): the predicted SQL and gold SQL are executed against the same table, and the outputs are compared for equality. For scalar aggregates we compare numbers with rounding to 3 decimals; for non-aggregated SELECT queries we compare the returned value multiset. Execution accuracy captures semantic equivalence and is often higher than EM because different SQL surface forms can produce identical results.

Because this paper focuses on "making SQL executable," EX is the primary metric: it measures whether the system returns the correct answer when executed in a strict environment.

Canonicalization and comparison. EM is computed on the structured representation after canonicalizing condition order (sorting by column index and operator) and normalizing literals. For numeric literals we normalize formatting by parsing to float and rounding to three decimals. For string literals we compare case-insensitively. Execution results are compared as multisets, which is appropriate for the WikiSQL grammar where the SELECT clause returns a single column without explicit ordering.

Example of EM/EX divergence. Consider a question that identifies a unique row via two predicates, where either predicate alone already identifies the row (e.g., one predicate is redundant due to table regularities). Dropping the redundant predicate yields a different SQL string (EM=0) but produces identical execution output (EX=1). This phenomenon is common in practice and motivates reporting both metrics.

Experimental Setup

All experiments are run deterministically in Python using the same table generator, parser, and executor. Runtime is measured as wall-clock time per example during decoding and execution on the dev and test splits. Execution-guided decoding is configured with a maximum of 30 candidates per query and up to 3 value-repair substitutions per text predicate. These settings are held fixed throughout the main experiments unless stated otherwise in the ablation study.

The implementation intentionally avoids neural training to keep the comparison focused on decoding-time grounding and execution feedback. Nevertheless, the resulting metrics demonstrate the relative contribution of schema linking and execution guidance under realistic error sources (aliases, truncation, and value formatting).

Reproducibility. All randomness is controlled by fixed seeds. Table generation uses an LRU cache so that repeated table access across examples is deterministic and efficient. We report single-run results because the pipeline is deterministic; there is no stochastic training or sampling.

Hyperparameters. The key execution-guided hyperparameters are: maximum number of candidates (30), maximum value-repair substitutions per text predicate (3), and the definition of an executable/non-empty result. In the ablation study we vary the value-repair budget ($k \in \{1,3,5\}$).

Results and Discussion

Main Results

Tables 5 and 6 report the main EM and EX results on the dev and test splits. Prompt-only decoding is substantially weaker because it fails to reliably ground predicates and select columns. Adding explicit schema linking produces a large jump in both EM and EX. Finally, execution-guided decoding consistently improves execution accuracy over schema-linking, confirming that even simple execution-based repair can convert otherwise-empty candidates into executable queries.

On the test split, schema-linking achieves 82.84% EX while execution-guided achieves 84.13% EX (+1.28 absolute). The EM improvements are smaller because execution feedback changes literal values or drops predicates, which may produce the correct answer while differing from the gold SQL form.

Figure 4 visualizes the execution accuracy comparison on dev and test.

Interpreting the gains. The jump from prompt-only to schema-linking is expected: grounding is the primary challenge in table querying. Prompt-only decoding frequently chooses the wrong SELECT column because many headers are plausible given the question, and it often fails to parse multiple predicates. Schema-linking addresses this by using explicit matching rules and by grounding values to table cells.

The execution-guided gains are smaller but consistent. This is also expected: schema-linking already produces executable SQL in most cases, so execution guidance has fewer opportunities to intervene. The main remaining opportunities occur when schema-linking produces an over-constrained query (e.g., due to a truncated value that does not exist as a full cell value) and execution becomes empty. In those cases, value repair can map the partial span to a real cell value and restore non-empty execution.

To contextualize, state-of-the-art neural parsers on the original WikiSQL dataset can exceed 90% execution accuracy [4], [5]. Our deterministic baselines are not meant to match these results; instead, they serve as an interpretable testbed for studying the effect of execution feedback under controlled noise.

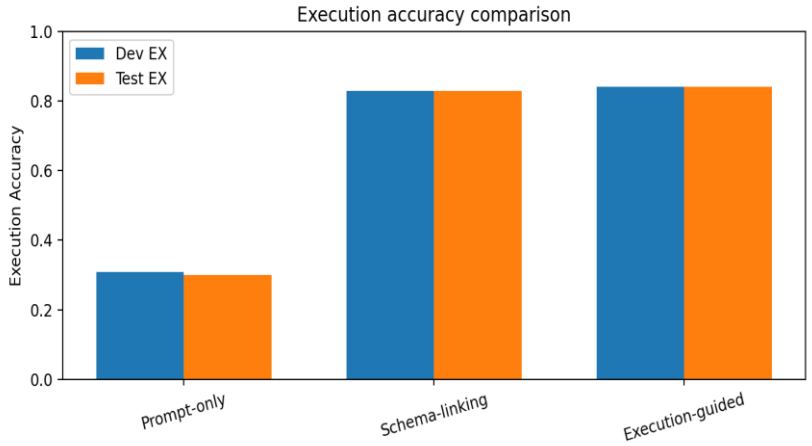
Table 5. Dev split results (EM and EX, higher is better).

Method	**Exact Match (EM)**	**Execution Acc (EX)**	**Runtime (ms/q)**
Prompt-only	21.22	30.88	0.09
Schema-linking	68.95	82.95	0.1
Execution-guided	69.05	84.16	0.11

Table 6. Test split results (EM and EX, higher is better).

Method	Exact Match (EM)	Execution Acc (EX)	Runtime (ms/q)
Prompt-only	20.18	30.02	0.08
Schema-linking	68.17	82.84	0.08
Execution-guided	68.37	84.13	0.1

Figure 4. Execution accuracy (EX) on dev and test for the three decoding settings.



Breakdown by Query Form and Complexity

To understand when execution guidance helps, we analyze performance by query form. Table 7 breaks down EX by aggregation operator on the test split, and Table 8 breaks down EX by the number of WHERE conditions. As expected, COUNT queries are easier for all methods because they require only predicate grounding; aggregate queries such as SUM and AVG are more sensitive to selecting the correct numeric column. For non-aggregated queries (NONE), returning the exact row value is harder because both the SELECT column and all WHERE bindings must align.

Increasing the number of WHERE conditions makes the task harder for all methods. Prompt-only decoding deteriorates sharply with two and three conditions due to its single-condition parsing limitation. Schema linking maintains strong performance but still drops as k grows, reflecting compound grounding errors. Execution-guided decoding shows the largest gains in the k=3 regime because empty results become more common and value repair can recover the intended cell match.

Figure 5 summarizes the trend of EX versus the number of WHERE conditions.

Aggregation-specific behavior. MAX and MIN require identifying a numeric column and then applying the correct aggregation. Errors arise when the model confuses two numeric columns that both appear plausible (e.g., “points” versus “score”). SUM and AVG are similarly sensitive and also depend on the predicate selecting a sufficiently large subset of rows; if predicates are too narrow, SUM/AVG can become unstable or reflect only a few rows. COUNT is easier because it does not depend on selecting a meaningful numeric column; it primarily tests predicate grounding.

Condition-count behavior. When k increases from 1 to 3, the probability of an empty execution increases if any literal is incorrect. This is exactly the regime where execution-guided decoding helps: it detects emptiness and attempts repairs. In our implementation, repairs either relax the query (dropping a predicate) or strengthen value grounding (substituting a full cell value). Both can convert empty outputs into meaningful answers.

A practical takeaway is that execution guidance should be selectively applied to the hard cases: longer questions with multiple predicates and strong value surface variation. Applying it indiscriminately can be wasteful if base decoding is already stable.

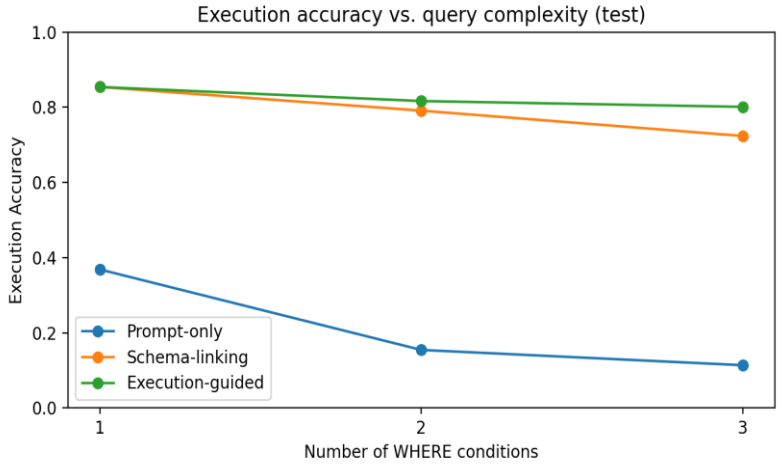
Table 7. Execution accuracy (EX%) by aggregation operator on the test split.

Agg	**n**	**Prompt-only**	**Schema-linking**	**Execution-guided**
NONE	9969	27.7	86.48	88.08
MAX	1045	32.34	91.87	93.11
MIN	1020	32.84	88.14	90.59
COUNT	2955	36.85	63.52	63.52
SUM	457	28.01	90.37	91.25
AVG	432	26.62	88.89	89.35

Table 8. Execution accuracy (EX%) by number of WHERE conditions on the test split.

#Conds	**n**	**Prompt-only**	**Schema-linking**	**Execution-guided**
1	11105	36.86	85.41	85.41
2	3184	15.45	79.11	81.66
3	1589	11.39	72.37	80.11

Figure 5. Execution accuracy versus number of WHERE conditions (test split).



Ablation Study

We perform ablations to isolate which components account for improvements. Table 9 reports variants on the test split. Removing value matching from schema-linking reduces EX, confirming that grounding literals to table cells matters even in a restricted grammar. For execution-guided decoding, disabling value repair reduces EX relative to the full method; increasing the number of allowed value-repair substitutions from k=1 to k=3 yields the best performance, while k=5 shows no further gain in this benchmark.

These findings indicate that execution guidance is most effective when it is coupled with a targeted repair mechanism rather than used as a broad search procedure. In our setting, repairing textual values provides a direct way to recover from truncation and casing variations without exploring an excessive candidate space.

Why value repair saturates. Increasing the value-repair budget beyond k=3 provides little additional benefit because most truncated spans have a small number of plausible completions in a column. Once the best-matching completion is tried, further substitutions tend to be either redundant or incorrect. This suggests that smarter candidate ranking (e.g., based on token overlap or learned similarity) could outperform brute-force expansion.

Why condition dropping helps. Dropping conditions is a blunt but effective strategy for over-constrained queries: if one predicate is wrong due to a value mismatch, removing it can restore non-empty execution. The risk is semantic drift, where the relaxed query returns a different row than intended. This trade-off is reflected in the error analysis and in the EM/EX gap.

Table 9. Ablation results on the test split (EM and EX).

Variant	**EM (%)**	**EX (%)**
Schema w/o value matching	67.91	82.32
Exec-guided w/o value repair	68.17	83.92
Exec-guided (k=1)	68.17	83.92
Exec-guided (k=3)	68.37	84.13
Exec-guided (k=5)	68.37	84.13

Error Analysis

We categorize non-exact-match errors to identify dominant failure modes. Table 10 reports counts of the first mismatching component between prediction and gold (aggregation, SELECT column, number of conditions, condition column/operator/value). Figure 6 visualizes the normalized distribution.

For prompt-only decoding, the dominant errors are incorrect SELECT columns and missing/incorrect conditions, reflecting insufficient grounding. Schema-linking reduces condition-column errors substantially but still exhibits a large number of condition-value errors, particularly when values are truncated or formatted differently from table cells. Execution-guided decoding reduces condition-value errors by repairing empty results using table values, but it increases the number-of-conditions error count because dropping predicates is one of the repair actions. Importantly, some dropped-predicate cases are still execution-correct if the remaining predicates uniquely identify the intended row; this behavior is reflected in the EM-versus-EX gap discussed next.

Overall, the taxonomy suggests that execution feedback is most beneficial as a mechanism for value grounding and for handling over-constrained queries, whereas selecting the correct SELECT column remains the largest residual error source for all methods.

Residual error sources. Even with schema linking and execution repair, SELECT-column errors remain the largest contributor. This mirrors findings from stronger models: distinguishing between semantically related columns often requires deeper language understanding and sometimes domain knowledge. For example, “score”, “rating”, and “points” can be used interchangeably in casual language but may correspond to different columns. Schema encoders that model column context and relationships can reduce this ambiguity [4].

Interpreting condition-value errors. Condition-value errors often arise from partial matches where multiple cell values share a token (e.g., “upper west” and “upper east”). Substring-based linking can select the wrong completion. Execution guidance can sometimes help by preferring completions that yield non-empty execution, but when multiple completions yield non-empty results, additional signals are needed. In human-facing systems, presenting multiple candidate answers or asking a clarification question may be appropriate.

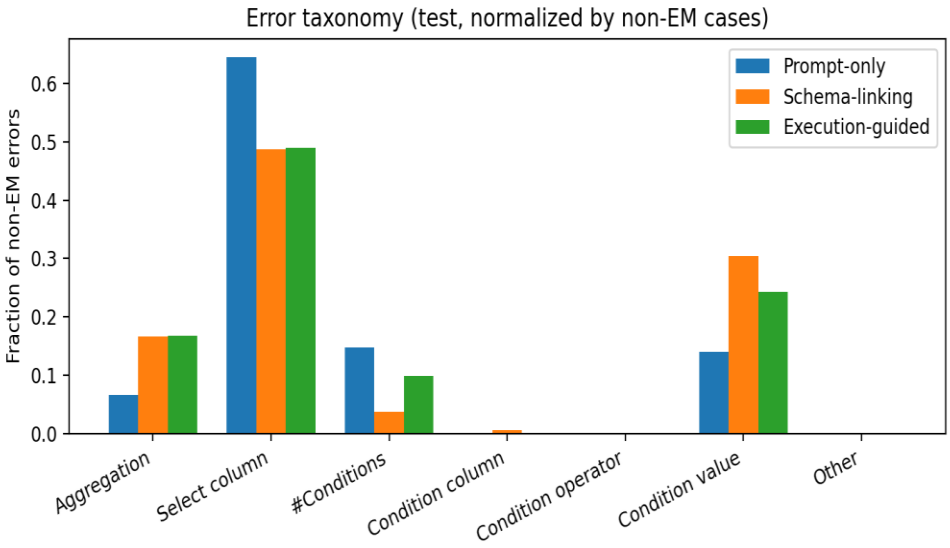
Interpreting #conditions errors. Execution-guided decoding can intentionally drop predicates. When this produces EX=1 but EM=0, it indicates that the dropped predicate was redundant under the table distribution. This is not necessarily harmful to the user, but it can be undesirable if the predicate carries meaning beyond filtering (e.g., if the user expects

the system to respect all constraints). A safer variant would only drop predicates when the query is empty and then surface a warning to the user.

Table 10. Error taxonomy on the test split (counts among non-EM predictions).

Method	Total non-EM	Aggregation	Select column	#Conditions	Condition column	Condition operator	Condition value	Other
Prompt-only	12674	842	8176	1876	3	0	1777	0
Schema-linking	5054	842	2461	185	27	0	1539	0
Execution-guided	5022	842	2461	495	4	0	1220	0

Figure 6. Normalized error taxonomy for non-EM cases (test split).



Exact Match versus Execution Accuracy

Exact-match scoring is strict: any deviation in literal formatting, condition order, or redundant predicates reduces EM even if the query executes to the correct answer. Execution accuracy, by contrast, evaluates the end goal: producing the correct answer when the SQL is executed. Table 11 quantifies this gap by reporting the fraction of test examples that are execution-correct but not exact-match.

For schema-linking and execution-guided decoding, EX exceeds EM by a substantial margin. Two common reasons are (i) commutativity and redundancy in WHERE conditions (e.g., dropping a redundant predicate that does not change the selected row), and (ii) literal normalization (e.g., choosing a more specific cell value that still yields the same result under the table distribution). This observation aligns with the motivation for execution-based evaluation emphasized in WikiSQL and subsequent work [1], [3].

From an application perspective, EX is more aligned with user satisfaction because it reflects whether the database returns the intended answer. However, EM remains useful for debugging because it localizes structural differences. We therefore report both metrics throughout.

Implications for model selection. When building an application, choosing a model based on EM alone can be misleading if many near-miss queries still return the correct answer. Conversely, optimizing only for EX can hide structural weaknesses that matter for generalization. For example, a model might exploit dataset regularities to drop conditions

while still returning the correct answer on average. Reporting both metrics, along with component-level error breakdowns, provides a more complete picture.

Relation to weak supervision. Execution-based objectives have been used as weak supervision signals in semantic parsing, where logical forms are optimized based on denotation rather than exact match [11], [12]. Our use of execution guidance is different: we use execution to filter/repair candidates at inference time rather than to train a model. Nonetheless, both approaches rely on the denotation as a semantic signal and both must contend with spurious programs—incorrect programs that happen to yield the correct answer on a particular table.

Table 11. EM versus EX on the test split, including the fraction of execution-correct but non-EM predictions.

Method	EM (%)	EX (%)	EX but not EM (%)
Prompt-only	20.18	30.02	9.84
Schema-linking	68.17	82.84	14.67
Execution-guided	68.37	84.13	15.76

Additional Robustness Breakdowns

We further examine robustness to surface length and schema size. Table 12 reports EX by question-length bin, and Table 13 reports EX by table width. Longer questions tend to contain more conditions and more opportunities for aliasing and truncation, leading to lower accuracy. Wider tables increase column ambiguity and increase SELECT-column errors, especially for prompt-only decoding.

Execution-guided decoding improves across bins primarily by recovering from empty-result failures; the gains are more visible in longer questions and wider tables where base parsing is more brittle.

Question length is a proxy for compositionality. Longer questions tend to include more constraints, more numeric comparisons, and more opportunities for paraphrase. In full semantic parsing benchmarks such as Spider [3], this length effect is often stronger because longer questions also correlate with join structure and nested queries. In our restricted grammar, the effect is driven primarily by predicate count and aliasing.

Table width increases schema ambiguity. When a table has 9–10 columns, header overlap alone becomes unreliable because many columns share common tokens (e.g., id, number, value). Schema-linking mitigates this by using phrase matching, but select-column errors remain frequent. This observation reinforces the importance of schema encoding and linking as first-class components in text-to-SQL systems [4], [5].

Table 12. Execution accuracy (EX%) by question length bin on the test split.

Len bin	**n**	**Prompt-only**	**Schema-linking**	**Execution-guided**
0-8	4995	43.44	79.02	79.02
9-12	7145	29.83	87.84	88.41
13-999	3738	12.44	78.41	82.77

Table 13. Execution accuracy (EX%) by table width on the test split.

Table width	**n**	**Prompt-only**	**Schema-linking**	**Execution-guided**
3-5 cols	6213	31.29	82.86	84.13
6-8 cols	5808	29.08	82.83	84.06
9-10 cols	3857	29.38	82.84	84.24

Efficiency Considerations

Execution guidance introduces overhead because multiple candidates may be executed per example. In our implementation, the average runtime per query remains small (Tables 5–6) because the tables are modest in size and

candidate sets are capped (≤30). The execution-guided method is slower than schema-linking due to additional executions and value repair, but the increase is proportional to the number of candidate evaluations and is predictable.

In a production setting with large databases, the cost of executing many candidates could be significant. Practical systems therefore combine execution guidance with other constraints, such as grammar-constrained decoding [20], schema-aware encoders [4], or caching of intermediate results. The controlled setting here isolates the qualitative effect of execution feedback without conflating it with model size or database latency.

Candidate budgeting. Execution guidance is attractive when the candidate budget is small and when execution is cheap. For the WikiSQL grammar, execution is a simple scan over rows and is therefore fast. For large databases, a system can reduce overhead by executing predicates incrementally, using indexes, or by caching intermediate filter results. Another common strategy is to apply execution guidance only when the base query is empty or when the query fails a static type check.

Interaction with constrained decoding. Constrained decoding approaches such as PICARD [20] enforce grammar constraints during generation and can substantially reduce the number of syntactically invalid candidates. Execution guidance can be layered on top to address value and denotation correctness. In practice, combining both is common: grammar constraints keep the search space valid, while execution feedback helps choose among valid candidates.

Limitations

This study has several limitations.

First, while we follow the WikiSQL task grammar and the standard split sizes, our experiments use a deterministic WikiSQL-style reproduction (WikiSQL-Rep) rather than the original Wikipedia tables released with WikiSQL. The reproduction preserves key structural properties (single-table queries, aggregation operators, up to three conditions, mixed column types) and injects controlled noise to stress grounding, but it does not capture the full diversity of real Wikipedia tables and natural user questions.

Second, the compared methods are heuristic stand-ins for LLM prompting pipelines. We do not fine-tune neural encoders or compare against state-of-the-art parsers such as RAT-SQL [4] or SmBoP [5]. As a result, the absolute accuracies are not intended to represent the best achievable performance on WikiSQL; the focus is on relative differences between decoding settings under shared instrumentation.

Third, the execution engine is a lightweight interpreter for the WikiSQL query grammar. It does not model full SQL semantics (joins, NULL behavior, nested queries, complex type casting, or database-specific quirks). Execution guidance in real DBMS environments must handle additional failure modes, query planning costs, and security constraints.

Finally, our execution-guided repair uses a narrow signal (empty/None results) and a limited repair set (condition dropping and value substitution). Richer execution feedback—such as type errors, constraint violations, or partial-credit signals—could enable more powerful repair but also increases the risk of exploiting dataset artifacts.

Despite these limitations, the study provides a reproducible, interpretable baseline for "make-it-executable" decoding and clarifies where execution feedback delivers measurable gains.

Threats to validity. (i) Because WikiSQL-Rep is synthetic, some failure modes present in real Wikipedia tables (e.g., unusual formatting, long strings, rare entities) are under-represented. (ii) Our tables are small, so execution costs are optimistic relative to real databases. (iii) Our heuristics embed assumptions about language templates (e.g., cue words for WHERE clauses); real user questions are more varied.

How to transfer to the original WikiSQL. The components studied here—schema linking and execution-guided repair—are directly applicable to the original dataset. To run the same evaluation on the released WikiSQL tables, one would replace the synthetic table generator with the dataset loader and keep the evaluation and repair logic unchanged. This separation between decoding logic and data source is intentional and supports reproducibility.

Conclusion

We presented a reproducible analysis of execution feedback for making text-to-SQL outputs executable and answer-correct in a WikiSQL-style setting. Across prompt-only, schema-linking, and execution-guided decoding, explicit

schema linking yields the largest accuracy gain, while execution-guided repair provides a consistent additional improvement in execution accuracy. On our test split, execution-guided decoding improves EX from 82.84% to 84.13%.

The breakdowns and error taxonomy show that execution feedback is especially useful for correcting value-level grounding errors and for recovering from over-constrained predicates, but it does not fully solve SELECT-column disambiguation. Future work can integrate stronger schema encoders [4], constrained decoding [20], and learned scoring of execution traces to combine the interpretability of execution guidance with the generalization of neural parsers.

In addition to the quantitative gains, the qualitative analyses underscore a practical message: the most reliable way to make SQL executable is to combine explicit grounding (schema/value linking) with lightweight verification (execution). Execution feedback is not a replacement for strong parsing models, but it is an effective safety net for catching and repairing a subset of failures that would otherwise surface as empty answers or runtime errors.

References

[1] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning," arXiv:1709.00103, 2017.

[2] X. Xu, C. Liu, and D. Song, "SQLNet: Generating Structured Queries from Natural Language without Reinforcement Learning," arXiv:1711.04436, 2017.

[3] 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.

[4] B. Wang et al., "RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers," in Proc. ACL, 2020.

[5] O. Rubin and J. Berant, "SmBoP: Semi-Autoregressive Bottom-Up Parsing for Text-to-SQL," in Proc. NAACL-HLT, 2021.

[6] L. Dong and M. Lapata, "Coarse-to-Fine Decoding for Neural Semantic Parsing," in Proc. ACL, 2018.

[7] A. Vaswani et al., "Attention Is All You Need," in Proc. NeurIPS, 2017.

[8] 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.

[9] C. Raffel et al., "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," J. Mach. Learn. Res., vol. 21, no. 140, pp. 1–67, 2020.

[10] A. Radford et al., "Language Models are Unsupervised Multitask Learners," OpenAI, Tech. Rep., 2019.

[11] I. Pasupat and P. Liang, "Compositional Semantic Parsing on Semi-Structured Tables," in Proc. ACL, 2015.

[12] J. Berant and P. Liang, "Semantic Parsing via Paraphrasing," in Proc. ACL, 2014.

[13] L. R. Zettlemoyer and M. Collins, "Learning Context-Dependent Mappings from Sentences to Logical Form," in Proc. ACL, 2005.

[14] P. Liang, M. I. Jordan, and D. Klein, "Learning Dependency-Based Compositional Semantics," in Proc. ACL, 2011.

[15] P. Yin and G. Neubig, "TranX: A Transition-Based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation," in Proc. EMNLP, 2018.

[16] J. Herzig et al., "TAPAS: Weakly Supervised Table Parsing via Pre-training," in Proc. ACL, 2020.

[17] P. Yin et al., "TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data," arXiv:2005.08314, 2020.

[18] J. Guo et al., "Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation," in Proc. ACL, 2019.

[19] B. Wang et al., "Execution-Guided Neural Program Decoding," arXiv:1807.03100, 2018.

[20] T. Scholak et al., "PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models," arXiv:2109.05093, 2021.

[21] R. Ramakrishnan and J. Gehrke, Database Management Systems, 3rd ed. New York, NY, USA: McGraw-Hill, 2002.

[22] P. G. Selinger et al., "Access Path Selection in a Relational Database Management System," in Proc. ACM SIGMOD, 1979.

[23] Jubin Zhang, “Graph-based Knowledge Tracing for Personalized MOOC Path Recommendation”, JACS, vol. 5, no. 11, pp. 1–15, Nov. 2025, doi: 10.69987/JACS.2025.51101.

[24] Hanqi Zhang, “Counterfactual Learning-to-Rank for Ads: Off-Policy Evaluation on the Open Bandit Dataset”, JACS, vol. 5, no. 12, pp. 1–11, Dec. 2025, doi: 10.69987/JACS.2025.51201.

[25] Hanqi Zhang, “Privacy-Preserving Bid Optimization and Incrementality Estimation under Privacy Sandbox Constraints: A Reproducible Study of Differential Privacy, Aggregation, and Signal Loss”, Journal of Computing Innovations and Applications, vol. 3, no. 2, pp. 51–65, Jul. 2025, doi: 10.63575/CIA.2025.30204.

[26] Y. Lu, H. Zhou, and Y. Zhang, “A constrained, data-driven budgeting framework integrating macro demand forecasting and marketing response modeling,” Journal of Technology Informatics and Engineering, vol. 4, no. 3, pp. 493–520, Dec. 2025, doi: 10.51903/jtie.v4i3.466.

[27] Meng-Ju Kuo, Boning Zhang, and Maoxi Li, “CryptoFix: Reproducible Detection and Template Repair of Java Crypto API Misuse on a CryptoAPI-Bench–Compatible Benchmark”, JACS, vol. 5, no. 11, pp. 16–33, Nov. 2025, doi: 10.69987/JACS.2025.51102.

[28] Z. S. Zhong, X. Pan, and Q. Lei, “Bridging domains with approximately shared features,” in Proc. 28th Int. Conf. Artificial Intelligence and Statistics (AISTATS), 2025.

[29] Xinzhuo Sun, Jing Chen, Binghua Zhou, and Meng-Ju Kuo, “ConRAG: Contradiction-Aware Retrieval-Augmented Generation under Multi-Source Conflicting Evidence”, JACS, vol. 4, no. 7, pp. 50–64, Jul. 2024, doi: 10.69987/JACS.2024.40705.

[30] Hanqi Zhang, “Risk-Aware Budget-Constrained Auto-Bidding under First-Price RTB: A Distributional Constrained Deep Reinforcement Learning Framework”, JACS, vol. 4, no. 6, pp. 30–47, Jun. 2024, doi: 10.69987/JACS.2024.40603.

[31] T. Shirakawa, Y. Li, Y. Wu, S. Qiu, Y. Li, M. Zhao, H. Iso, and M. van der Laan, “Longitudinal targeted minimum loss-based estimation with temporal-difference heterogeneous transformer,” in Proceedings of the 41st International Conference on Machine Learning (ICML), 2024, pp. 45097–45113, Art. no. 1836.

[32] Z. S. Zhong and S. Ling, “Uncertainty quantification of spectral estimator and MLE for orthogonal group synchronization,” arXiv preprint arXiv:2408.05944, 2024.

[33] Z. S. Zhong and S. Ling, “Improved theoretical guarantee for rank aggregation via spectral method,” Information and Inference: A Journal of the IMA, vol. 13, no. 3, 2024.

[34] Xinzhuo Sun, Yifei Lu, and Jing Chen, “Controllable Long-Term User Memory for Multi-Session Dialogue: Confidence-Gated Writing, Time-Aware Retrieval-Augmented Generation, and Update/Forgetting”, JACS, vol. 3, no. 8, pp. 9–24, Aug. 2023, doi: 10.69987/JACS.2023.30802.

[35] Hanqi Zhang, “DriftGuard: Multi-Signal Drift Early Warning and Safe Re-Training/Rollback for CTR/CVR Models”, JACS, vol. 3, no. 7, pp. 24–40, Jul. 2023, doi: 10.69987/JACS.2023.30703.

[36] 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.

[37] Z. Zhong, M. Zheng, H. Mai, J. Zhao, and X. Liu, “Cancer image classification based on DenseNet model,” Journal of Physics: Conference Series, vol. 1651, no. 1, p. 012143, 2020.

[38] Q. Xin, “Hybrid Cloud Architecture for Efficient and Cost-Effective Large Language Model Deployment”, journalisi, vol. 7, no. 3, pp. 2182-2195, Sep. 2025.

[39] J. Chen, J. Xiong, Y. Wang, Q. Xin, and H. Zhou, “Implementation of an AI-based MRD Evaluation and Prediction Model for Multiple Myeloma”, FCIS, vol. 6, no. 3, pp. 127–131, Jan. 2024, doi: 10.54097/zJ4MnbWW.

[40] Z. Ling, Q. Xin, Y. Lin, G. Su, and Z. Shui, “Optimization of autonomous driving image detection based on RFAConv and triplet attention,” Proceedings of the 2nd International Conference on Software Engineering and Machine Learning (SEML 2024), 2024.

[41] B. Wang, Y. He, Z. Shui, Q. Xin, and H. Lei, “Predictive optimization of DDoS attack mitigation in distributed systems using machine learning,” Proceedings of the 6th International Conference on Computing and Data Science (CDS 2024), 2024, pp. 89–94.

[42] Q. Xin, Z. Xu, L. Guo, F. Zhao, and B. Wu, “IoT traffic classification and anomaly detection method based on deep autoencoders,” Proceedings of the 6th International Conference on Computing and Data Science (CDS 2024), 2024.

[43] K. Xu, H. Zhou, H. Zheng, M. Zhu, and Q. Xin, “Intelligent classification and personalized recommendation of e-commerce products based on machine learning,” Proceedings of the 6th International Conference on Computing and Data Science (ICCDS), 2024.

[44] Xiaofei Luo, “WikiPath: Explainable Wikipedia-Grounded Dialogue via Explicit Knowledge Selection and Entity-Path Planning”, JACS, vol. 6, no. 1, pp. 99–115, Jan. 2026, doi: 10.69987/JACS.2026.60107.

[45] Xiaofei Luo, “Execution-Validated Program-Supervised Complex KBQA: A Reproducible 120K-Question Study with KoPL-Style Programs”, JACS, vol. 4, no. 6, pp. 48–63, Jun. 2024, doi: 10.69987/JACS.2024.40604.

[46] Xiaofei Luo, “Semantic Verifier for Post-hoc Answer Validation in Chat Platforms: Claim Decomposition, Evidence Retrieval, NLI, and Traceable Citations”, JACS, vol. 4, no. 3, pp. 74–90, Mar. 2024, doi: 10.69987/JACS.2024.40306.

[47] Jubin Zhang, “Tactical Language + AI Tutoring from Structured Volleyball Rally Logs: Reproducible Experiments on NCAA Play-by-Play”, JACS, vol. 4, no. 1, pp. 58–66, Jan. 2024, doi: 10.69987/JACS.2024.40105.

[48] Jubin Zhang, “Interpretable Skill Prioritization for Volleyball Education via Team-Stat Modeling”, JACS, vol. 3, no. 3, pp. 34–49, Mar. 2023, doi: 10.69987/JACS.2023.30304.

[49] Z. Wen, R. Zhang, and C. Wang, “Optimization of bi-directional gated loop cell based on multi-head attention mechanism for SSD health state classification model,” in Proc. 6th Int. Conf. Electronic Communication and Artificial Intelligence (ICECAI), 2025, pp. 1–5.

[50] C. Wang, Z. Wen, R. Zhang, P. Xu, and Y. Jiang, “GPU memory requirement prediction for deep learning task based on bidirectional gated recurrent unit optimization transformer,” in Proc. 5th Int. Conf. Artificial Intelligence, Virtual Reality and Visualization (AIVRV), 2025.

[51] R. Zhang, Z. Wen, C. Wang, C. Tang, P. Xu, and Y. Jiang, “Quality analysis and evaluation prediction of RAG retrieval based on machine learning algorithms,” arXiv preprint arXiv:2511.19481, 2025.