WHY, WHERE, HOW: TAXONOMY-GUIDED ERROR GROUNDING FOR CODE REPAIR IN NL2SQL

Anonymous authors
Paper under double-blind review

ABSTRACT

SQL queries that large language models write from natural language questions can execute successfully yet produce incorrect results, so execution alone does not reveal what to fix. An error taxonomy says *why* the query is wrong, but not *where* to look or *how* to change it. Existing methods can guide SQL correction through feedback, error reports, or generated plans alongside an unmasked query. We introduce **TEG** (**T**axonomy-guided **E**rror **G**rounding), which turns a supplied diagnosis into a structured correction input for natural language-to-SQL (NL2SQL) correction. Type-specific rules map each error type to construct classes to reconsider and an edit operation to request. TEG masks the selected constructs in the query when applicable and states that operation in an edit instruction. TEG generates candidate corrections from this input, uses execution feedback to guide candidate selection, and repeats the process one annotation at a time for queries with several errors. On NL2SQL-BUGs, TEG reaches 47.3 single-error execution accuracy and 37.0 overall with Qwen2.5-7B-Instruct. Across the model sizes and thinking modes evaluated in the main comparison, TEG outperforms all evaluated baselines on single-error queries, even when the baselines receive the same error-type annotations. With predicted types, TEG stays above direct LLM correction and ErrorLLM on single-error queries.

1 INTRODUCTION

Large language models (LLMs) now generate code from natural language specifications (Chen et al., 2021; Jiang et al., 2026). A generated program can be syntactically valid and run without an exception while still returning a wrong result, so execution alone does not establish correctness and semantic errors must be repaired (Liu et al., 2023; Pan et al., 2023; Chen et al., 2024a; Li et al., 2026a). Natural language-to-SQL (NL2SQL) is an acute case: agents increasingly rely on generated SQL over large relational databases (Lei et al., 2025), and a wrong query can return a plausible table rather than an error (Liu et al., 2025b). A diagnosis from an error taxonomy tells the LLM *why* the query is wrong, but not *where* to look in the query or *how* to change it.

Reasoning and self-correction methods (Kojima et al., 2022; Madaan et al., 2023; Chen et al., 2024b) supply no error-type diagnosis, so the model must discover the why, the where, and the how from the code and available feedback. NL2SQL pipelines such as DIN-SQL, MAC-SQL, and SQLFixAgent combine SQL generation with self-correction or execution-guided refinement (Pourreza & Rafiei, 2023; Wang et al., 2025a; Cen et al., 2025). ErrorLLM (Hong et al., 2026) is a closely related approach: it starts from an error type, locates candidate syntax-tree nodes and schema elements under type-specific guidelines, and supplies them as context next to the issued query for a single refinement call. TEG turns the supplied diagnosis into a correction input: deterministic rules select construct classes to mask when applicable and specify the requested edit.

We call this mapping *error grounding* (Figure 1): translating a taxonomy-level diagnosis into the classes of query constructs to reconsider and the edit operation to request. For example, a diagnosis of an incorrect reference selects the class of expressions that carry references and requests a replacement, whereas a missing construct selects no existing element and requests an insertion. A syntax tree makes these classes addressable, so the selection can be applied to the query directly instead of predicting the exact faulty occurrence. SQL offers both an error taxonomy and structural editing rules:

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

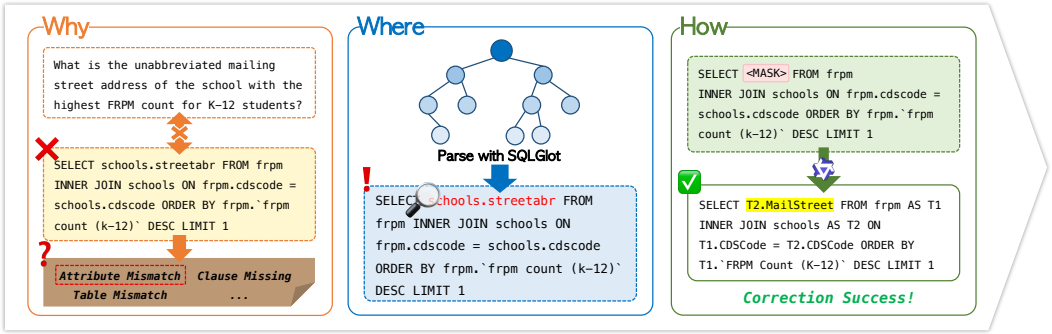

Figure 1: **Error grounding maps *why* a query is wrong to *where* and *how* to fix it.** TEG masks the constructs selected by the error type and supplies an edit instruction. For an *Attribute Mismatch*, TEG masks column expressions, allowing the LLM to replace the abbreviated street field `schools.streetabr` with the required unabbreviated mailing street field `T2.MailStreet`.

it exposes recurring constructs over an explicit schema, and a SQL error taxonomy labels semantic errors directly in terms of these constructs (Liu et al., 2025b).

We propose **TEG** (**T**axonomy-guided **E**rror **G**rounding), which realizes error grounding for SQL on the syntax tree that SQLGlot parses from the query. The *where* rule maps each supplied error type to the constructs to reconsider: TEG masks the associated syntax-tree nodes when applicable, such as column expressions for *Attribute Mismatch*. The *how* rule maps the same type to an edit instruction, such as replacing the masked construct for a *Mismatch* or inserting an absent one for a *Missing* error. Because the rules select construct classes, masks can cover correct constructs; the LLM generates a complete query from the grounded input.

Grounding sets where and how, but in our development runs a single output often named a column or value not in the database. TEG therefore generates several candidates, prefers those that execute and return rows, and lets an LLM refiner select or rewrite among them, a stage we develop from SQLFixAgent (Cen et al., 2025) for grounded prompts. With several errors, one prompt carrying every error type would mix several masks and instructions. TEG instead processes one annotation at a time, editing existing constructs before inserting new ones, and re-grounds the output of each step.

We evaluate TEG on incorrect queries from NL2SQL-BUGs (Liu et al., 2025b), derived from BIRD (Li et al., 2023), against the methods above and our adaptation of ErrorLLM. TEG outperforms every evaluated baseline on single-error queries across the model sizes and thinking modes evaluated in the main comparison, and the lead holds when baselines receive the same annotations. On Qwen2.5-7B, it reaches 47.3 execution accuracy on single-error queries and 37.0 over all queries, against 33.6 and 24.0 for ErrorLLM, and leads every baseline on multi-error queries. With predicted types, TEG stays above direct LLM correction and ErrorLLM on single-error queries across all evaluated configurations. The w/o masking variant retains the error types but reduces single-error accuracy from 47.3 to 40.9, supporting the masking-based editing configuration within TEG.

In summary, our main contributions are:

- **We formulate error grounding for SQL correction.** We map a supplied diagnosis (*why*) to construct classes to reconsider (*where*) and an edit instruction (*how*). This mapping constructs the correction input without requiring exact fault localization.

- **We design type-specific rules and sequential re-grounding in TEG.** Four edit groups cover the 29 evaluated NL2SQL-BUGs sub-types, with masking when applicable and type-specific edit instructions. For multiple errors, TEG reapplies the rules to the updated query for each annotation.

- **We demonstrate improved SQL correction with TEG.** TEG achieves the highest observed accuracy in the main comparisons, including those with shared annotations. We also evaluate predicted-type correction on single-error queries and compare component variants to examine grounding, sequential correction, and candidate selection within the pipeline.

2 Related Work

Reasoning and self-correction. Reasoning prompts such as CoT (Kojima et al., 2022; Wei et al., 2022) elicit step-by-step analysis, while Self-Refine (Madaan et al., 2023) and Self-Debug (Chen et al., 2024b) guide correction with natural language critiques, code explanations, or execution feedback. Self-correction can also be learned (Kumar et al., 2025), although improvement through intrinsic feedback alone is not guaranteed (Huang et al., 2024). These methods supply no error-type diagnosis, so the model must decide why, where, and how to correct from the code and its feedback alone; TEG instead assumes the diagnosis is given and asks how best to use it.

SQL-specific refinement. LLMs have enabled diverse NL2SQL approaches (Liu et al., 2025a). These span prompt design, task-specific training, structure-guided generation, and test-time scaling (Dong et al., 2023; Li et al., 2024; Zhang et al., 2024; Liao et al., 2026; He et al., 2025; Li et al., 2025; Wang et al., 2025b). Refinement and debugging pipelines use self-correction, execution feedback, or alternative query representations (Talaei et al., 2024; Askari et al., 2025; Xu et al., 2025b; Li et al., 2026b; Qu et al., 2025). DIN-SQL (Pourreza & Rafiei, 2023) couples decomposed generation with self-correction, MAC-SQL (Wang et al., 2025a) coordinates schema selection, decomposition, and execution-guided refinement, and SQLFixAgent (Cen et al., 2025) separates query review, candidate generation, and refinement across agents. Candidate validation and selection are also studied (Fan et al., 2025; Gorti et al., 2025; Lee et al., 2025; Pourreza et al., 2025). TEG develops its execution-based candidate selection from SQLFixAgent (Cen et al., 2025) for grounded inputs.

Program repair and code infilling. DrRepair localizes faulty lines (Yasunaga & Liang, 2020), while LecPrompt uses CodeBERT for token-level localization and repair (Xu & Sheng, 2024; Feng et al., 2020). Masked-span reconstruction (Guo et al., 2022) and code infilling (Fried et al., 2022; Allal et al., 2023) recover missing code from surrounding context. TEG selects construct classes by error type without exact fault localization and uses masks when applicable and edit instructions as correction inputs. It trains no infilling model; the LLM generates full queries and may edit unmasked constructs.

Error taxonomies and detection. Error taxonomies give diagnoses reusable categories, from software defect classifications (Chillarege et al., 1992) to the semantic SQL errors annotated in NL2SQL-BUGs (Liu et al., 2025b). Related SQL error detection methods assess generated queries using learned structural representations (Chen et al., 2023) or metamorphic testing (Yang et al., 2025b). Multi-grained error identification (Xu et al., 2025a) distinguishes system, skeleton, and value errors and uses type-specific feedback to guide SQL correction. SQLens (Gong et al., 2026) combines database and LLM signals into clause-level error reports with correction instructions and uses these reports to guide sequential refinement of the generated query.

Taxonomy-guided refinement. SQL-of-Thought (Chaturvedi et al., 2025) uses an error taxonomy to guide an LLM-generated correction plan, which a separate agent uses to revise the query. ErrorLLM (Hong et al., 2026) is closely related: its error taxonomy draws on NL2SQL-BUGs, and its refinement stage uses error types to guide structural correction. It trains an error detector that predicts taxonomy categories with dedicated error tokens from the question, schema, and SQL structure, and localizes relevant syntax-tree nodes and schema elements under type-specific guidelines. The error type also governs the scope of context extraction, and the resulting subtrees, schema context, and analyses are supplied alongside the issued query for joint refinement. We adapt its refinement stage as a principal baseline, using the same supplied error annotations as TEG. TEG maps the supplied diagnosis (*why*) through deterministic rules to construct-class masks (*where*) and edit instructions (*how*), removing selected constructs from the query input when masking applies.

3 TEG: Taxonomy-guided Error Grounding

TEG turns error annotations into grounded SQL correction steps (Figure 2, Algorithm 1). Each step masks the constructs that the error type selects in the SQLGlot syntax tree, when there are any, states the requested edit, and selects the corrected query from candidates by execution; a query with several annotations takes one step per annotation in a fixed order.

Appendix A gives the full rules, masking algorithms, prompts, and candidate selection details.

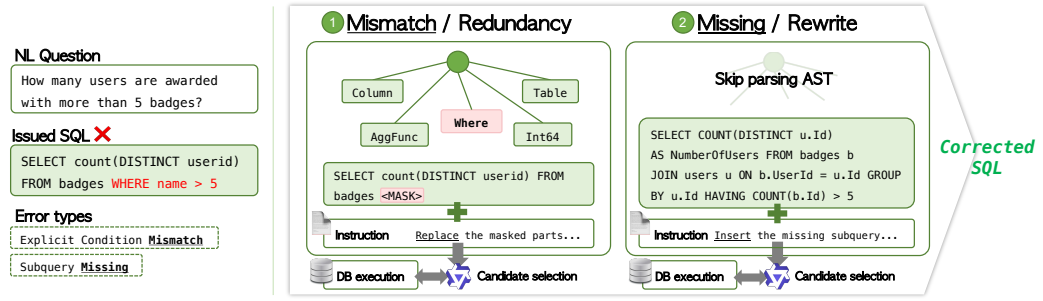

Figure 2: **TEG maps error types to masking rules and edit instructions.** The *where* rule masks constructs for *Mismatch* and *Redundancy* (1); *Missing* and *Rewrite* (2) leave the query unmasked. The *how* rule specifies the edit. Here, step 1 replaces the masked `WHERE` predicate for an *Explicit Condition Mismatch*; step 2 inserts the missing subquery for *Subquery Missing* into step 1's output.

3.1 Grounding error types into masks and edit instructions

Grounding turns an error type, which says *why* a query is wrong, into two parts of the correction input rather than of its output: a mask over the constructs to reconsider, which tells the LLM *where* to look, and an edit instruction, which tells it *how* to change them. The mask removes the selected constructs from the query the LLM sees, and the edit instruction names the requested edit, so one error type determines both what the LLM reconsiders and what it is asked to do.

TEG prompts an off-the-shelf LLM with an issued query, its question, schema, and error annotations, and returns a corrected query. Let q be the issued query, n its natural language question, and S the database schema. We assume an annotation sequence $\mathcal{E} = ((t_1, s_1), \ldots, (t_m, s_m))$, where each pair specifies an error category (e.g., *Attribute-Related Errors*) and sub-type (e.g., *Attribute Mismatch*). The goal is a corrected SQL query q^* semantically aligned with n under schema S. We call $|\mathcal{E}| = 1$ the single-error setting and $|\mathcal{E}| > 1$ the multi-error setting.

The grounding rules act on four edit groups that TEG defines over the error taxonomy of NL2SQL-BUGs (Liu et al., 2025b). The queries we evaluate cover nine categories and 29 observed sub-types, classified by the SQL construct they affect, such as *Attribute-Related*, *Table-Related*, and *Clause-Related*. The four groups are *Mismatch*, *Redundancy*, *Missing*, and *Rewrite*, and the mapping from sub-types to groups is part of TEG and specifies the default edit operation and correction order. The annotations retain their original taxonomy categories and sub-types.

The *where* rule maps the category and sub-type to classes of nodes in the abstract syntax tree (AST) to mask. For *Mismatch* and *Redundancy*, TEG masks matching existing constructs. *Missing* requests an absent construct and *Rewrite* requests broader restructuring, so TEG leaves the query unmasked for both. For example, *Attribute Mismatch* hides column expressions, *Explicit Condition Mismatch* hides the `WHERE` predicate, and *Aggregate Functions* hides the aggregate together with the grouping and filtering clauses that depend on it. The rules also set the granularity of masking: hiding a single node can leave a dependent clause inconsistent after repair, whereas hiding a whole clause can discard context the LLM would otherwise keep, so each rule sets the unit per error type.

The *how* rule specifies an edit instruction for the supplied error type: replace for *Mismatch*, remove for *Redundancy*, insert for *Missing*, and restructure for *Rewrite*, which covers incomplete queries and errors that need broader changes. Sub-types such as *Aggregate Functions* and *ASC/DESC* are grouped with *Mismatch* for the correction order, but their edit instructions depend on whether a mask is present: replacement or deletion with a mask, and insertion without one. For the *DISTINCT* sub-type, the instruction also permits restructuring joins, projection, or grouping.

3.2 Grounded correction with candidate selection

A correction step applies both rules to one annotation and asks the LLM for a repaired query. For *Mismatch* and *Redundancy*, TEG parses the current query with SQLGlot and replaces the selected constructs with `<MASK>` placeholders; *Missing* and *Rewrite* leave the query unmasked. Attribute masking locates column expressions in the parsed tree and replaces their SQL strings, whereas the other masking rules replace the AST nodes themselves. If parsing or masking fails, the query is

Algorithm 1 TEG re-grounds the current query for each error annotation. TEG(q, n, S, $\mathcal{E}$) processes the nonempty sequence $\mathcal{E}$ in edit-group order and applies candidate selection at each step; the retry branch of Section 3.3 is omitted. Each updated query becomes the input to the next step.

Require: SQL q, question n, schema S, annotation sequence $\mathcal{E}$, $|\mathcal{E}| > 0$
Ensure: Corrected SQL q^*

$N \leftarrow \{n\} \cup \textsc{Craft}(n)$ ▷ question paraphrases, produced once
Stable-sort $\mathcal{E}$ by edit group (Mismatch, Redundancy, Missing, Rewrite)
for each (t, s) in $\mathcal{E}$, in order **do**
  $\tilde{q} \leftarrow \textsc{Masking}(q, t, s)$ ▷ type-specific masking; no mask for Missing or Rewrite
  $C \leftarrow \{\text{LLM}(\textsc{Prompt}(\tilde{q}, n', S, t, s)) : n' \in N\}$ ▷ one candidate per question variant
  $q \leftarrow \textsc{Select}(C, n, S, q)$ ▷ execution filter and refiner
end for
return q

passed unchanged to the LLM. The prompt includes the possibly masked query, together with the question, the schema, and the edit instruction. When masking is applied, the prompt does not include the unmasked query, so the LLM resolves each placeholder without seeing the removed constructs.

Masking removes the selected constructs from the input, whereas naming the error type leaves them in place. The rules select node classes deterministically, but the LLM decides what to change: several nodes may match, and some may already be correct. The LLM generates a complete query, so the output can revise unmasked constructs as well as fill the placeholders.

TEG selects the corrected query from several candidates with execution feedback, because one answer to a grounded prompt can name a column or value absent from the database. We develop this selection stage from the QueryCrafter and SQLRefiner of SQLFixAgent (Cen et al., 2025) for grounded prompts, combining candidate generation, execution filtering, and LLM-based selection or refinement. The original question and up to three paraphrases yield at most four candidates per step. The paraphrases are generated once and reused as later steps correct the updated query.

3.3 Sequential re-grounding for multiple errors

We design sequential correction to process one annotation at a time and re-ground the query after each update, so that each step targets one error type with its own mask and instruction.

TEG uses the order *Mismatch*, *Redundancy*, *Missing*, *Rewrite*, preserving input order within groups. We edit existing constructs first because an insertion or restructuring can move or remove the constructs a later rule would select. Re-grounding applies the current annotation to the query produced by earlier steps, masking it only when applicable. TEG retains the supplied annotations without predicting new error types, and later edits can undo an earlier repair. If masking yields no placeholder and a previous correction step exists, TEG uses the current query without masking, together with a retry instruction that names both the current and the previous errors. If $\mathcal{E}$ is empty, as when no error type is predicted, TEG falls back to one unmasked correction call and keeps q if it yields no SQL.

4 Experiments

Our experiments ask three questions about error grounding, in the order a reader would raise them:

- **How does TEG compare with existing correction methods?** We compare TEG with correction baselines, including versions given the same error types (Section 4.2).
- **Does the gain survive predicted error types?** We replace the benchmark's error annotations with error types predicted by Opus 4.8 and ask how much of the lead remains (Section 4.3).
- **How do component variants compare, and what is the computational cost?** We ablate the mask, the sequential steps, and the candidate selection, compare error categories, and measure model-call time, excluding execution and preprocessing (Section 4.4).

Table 1: **TEG leads single-error correction across the evaluated models.** Entries are execution accuracies (%) on single-error queries from NL2SQL-BUGs. In the Error types column, ✓ denotes supplied oracle error types from NL2SQL-BUGs; ✗ denotes no error types. ErrorLLM denotes our adaptation (Section 4.1). Qwen3-8B is evaluated with thinking OFF and ON. Bold marks the best accuracy per Qwen setting. TEG's margin over the strongest baseline ranges from 0.9 to 11.8 points.

Method	Error types	Qwen2.5-Instruct 7B	14B	32B	Qwen3-8B Thinking OFF	Thinking ON
Base	✗	30.0	35.5	41.8	31.8	43.6
Reasoning and self-correction						
CoT (Kojima et al., 2022)	✗	35.5	37.3	45.5	33.6	48.2
Self-Debug (Simple) (Chen et al., 2024b)	✗	33.6	38.2	48.2	38.2	50.9
Self-Debug (Expl) (Chen et al., 2024b)	✗	29.1	35.5	44.5	44.5	47.3
SQL-specific refinement						
DIN-SQL (Pourreza & Rafiei, 2023)	✗	24.5	42.7	53.6	21.8	43.6
MAC-SQL (Wang et al., 2025a)	✗	18.2	15.5	18.2	25.5	30.0
SQLFixAgent (Cen et al., 2025)	✗	32.7	36.4	43.6	37.3	42.7
Taxonomy-guided refinement						
ErrorLLM (Hong et al., 2026)	✓	33.6	51.8	53.6	39.1	45.5
Taxonomy-guided error grounding						
TEG (ours)	✓	**47.3**	**52.7**	**57.3**	**46.4**	**57.3**

4.1 Setup

We evaluate on NL2SQL-BUGs (Liu et al., 2025b), derived from BIRD (Li et al., 2023), using incorrect queries paired with reference SQL; the evaluation subset covers 29 error sub-types across nine categories. Execution accuracy measures whether the corrected query and the reference query return the same set of row tuples on the same database, ignoring row order and duplicates while preserving column order. Queries that raise execution errors or exceed the 5s time limit are counted as incorrect, and all accuracy scores are reported on a 0–100 scale. During correction, TEG executes its own candidates with a separate 5s limit and never sees the reference SQL or its result.

For the main comparisons, we use Qwen2.5-Instruct (Yang et al., 2024) at 7B, 14B, and 32B and Qwen3-8B (Yang et al., 2025a) with thinking OFF and ON, all with greedy decoding, as the LLM for every method. With each model, we compare TEG against the LLM itself and three baseline groups:

- **Base**: the LLM itself, asked once to correct the unmasked query with no error type, as the reference for what each correction method adds beyond a single call.
- **Reasoning and self-correction**: CoT (Kojima et al., 2022) and Self-Debug in its Simple and Expl variants (Chen et al., 2024b), which we adapt to SQL correction by prompting them to analyze or critique the given erroneous query and produce a corrected query.
- **SQL-specific refinement**: DIN-SQL (Pourreza & Rafiei, 2023), MAC-SQL (Wang et al., 2025a), and SQLFixAgent (Cen et al., 2025), which we adapt to SQL correction by providing the given erroneous query as input or context, without an error-type diagnosis.
- **Taxonomy-guided refinement**: our adaptation of ErrorLLM (Hong et al., 2026), which uses AST-node localization, error-specific subtree and schema-context extraction, and joint refinement. It uses the supplied annotations instead of its trained error detector, and database metadata with lexical matches to the question instead of the learned question–schema encoder.

We additionally evaluate Base, ErrorLLM, and TEG with Opus 4.8 as the corrector in Section 4.3. Unless otherwise stated, supplied error types are oracle annotations from NL2SQL-BUGs. ErrorLLM and TEG receive these annotations in the main comparisons, while Base and the other six baselines receive no error types. We additionally evaluate all baselines with the same annotations. Appendix B gives dataset statistics, thinking modes, baseline and ablation implementations, and hardware.

Table 2: **TEG leads every error-count group with Qwen2.5-7B-Instruct.** Entries are execution accuracies (%) on NL2SQL-BUGs, grouped by error count; Overall covers all queries. In the Error types column, ✓ denotes supplied oracle error types from NL2SQL-BUGs; ✗ denotes no error types. ErrorLLM denotes our adaptation (Section 4.1). Bold marks the best value per column. TEG retains its lead on multi-error queries, although its accuracy is lower for queries with more errors.

Method	Error types	1 error	2 errors	3+ errors	Overall
Base	✗	30.0	25.9	14.1	23.7
Reasoning and self-correction					
CoT (Kojima et al., 2022)	✗	35.5	21.5	21.8	24.4
Self-Debug (Simple) (Chen et al., 2024b)	✗	33.6	20.9	16.9	22.4
Self-Debug (Expl) (Chen et al., 2024b)	✗	29.1	24.6	18.3	23.9
SQL-specific refinement					
DIN-SQL (Pourreza & Rafiei, 2023)	✗	24.5	20.2	15.5	19.9
MAC-SQL (Wang et al., 2025a)	✗	18.2	14.1	8.5	13.5
SQLFixAgent (Cen et al., 2025)	✗	32.7	26.9	19.7	26.2
Taxonomy-guided refinement					
ErrorLLM (Hong et al., 2026)	✓	33.6	23.6	17.6	24.0
Taxonomy-guided error grounding					
TEG (ours)	✓	**47.3**	**36.7**	**29.6**	**37.0**

4.2 Main results

Single-error queries. Each query tests one grounded correction step without sequential re-grounding. TEG has the highest single-error accuracy in every configuration reported in Table 1. At 7B, it exceeds the strongest baseline, CoT, by 11.8 points (47.3 versus 35.5). Across the Qwen2.5 scales, it reaches 47.3, 52.7, and 57.3, against 33.6, 51.8, and 53.6 for ErrorLLM. On Qwen3-8B, it reaches 46.4 with thinking OFF and 57.3 with thinking ON, above the strongest baseline in each mode, Self-Debug (Expl) at 44.5 and Self-Debug (Simple) at 50.9. TEG scores above CoT at every Qwen2.5 scale and above Base in both Qwen3-8B thinking modes. On Qwen3-8B, thinking lifts Base from 31.8 to 43.6 and TEG from 46.4 to 57.3. TEG can complement model reasoning.

Multi-error queries. The lead extends to queries with several errors (Table 2). With Qwen2.5-7B-Instruct on all queries, TEG reaches 37.0 overall against 26.2 for SQLFixAgent, the strongest baseline, and 24.0 for ErrorLLM, and it leads every error-count group, although its accuracy declines as the number of errors increases. It also leads every error category on multi-error queries, by 5.9 to 15.4 points over the next best method.

Baselines given error types. TEG's lead persists when all methods receive the same error types. To distinguish access to the error type from how it is used, we add the same annotations to each baseline's prompts while retaining its correction procedure. TEG stays ahead in all five configurations on single-error queries, by 0.9 points at Qwen2.5-14B and Qwen3-8B with thinking OFF and by 11.8 points at Qwen2.5-7B, and on all queries it reaches 37.0 against 26.0 for the strongest baseline given error types, Self-Debug (Simple), leading in every error-count group. Error types help some baselines but not others: at 7B, CoT falls from 35.5 to 32.7, while Self-Debug (Simple) rises from 33.6 to 35.5. Access to error types alone does not explain TEG's advantage. Appendix C reports these comparisons in full, together with the per-category breakdowns. These comparisons evaluate complete correction pipelines: supplying the same annotations controls diagnostic information, but does not equalize candidate-generation and refinement procedures.

4.3 End-to-end correction with predicted error types

We evaluate single-error correction using the same Opus 4.8 type predictions for TEG and ErrorLLM (Table 3). TEG outperforms Base and ErrorLLM in every evaluated corrector configuration. Across the five configurations in Table 1, its accuracy decreases by 0.9–4.6 points relative to the oracle-type results. These gains hold even though the predicted types exactly match the benchmark annotations on only 52.7% of single-error queries.

Table 3: **Single-error correction with predicted error types.** Entries are single-error execution accuracies (%) on NL2SQL-BUGs. Opus 4.8 supplies the same predicted types to TEG and ErrorLLM; it also corrects queries in the Opus 4.8 column. Base receives no types. Bold marks the best value per setting. On empty predictions, TEG attempts unmasked correction; ErrorLLM retains the query.

Method	Qwen2.5-Instruct 7B	14B	32B	Qwen3-8B Thinking OFF	Thinking ON	Opus 4.8
Base	30.0	35.5	41.8	31.8	43.6	63.6
ErrorLLM	32.7	45.5	47.3	38.2	44.5	64.5
TEG (ours)	**45.5**	**48.2**	**52.7**	**45.5**	**53.6**	**69.1**

Table 4: **Every ablation variant has lower overall accuracy than TEG.** Entries are execution accuracies (%) by error count on NL2SQL-BUGs with Qwen2.5-7B-Instruct; Overall covers all queries. Bold marks the best value per column, including ties. Without candidate selection or sequential correction, single-error accuracy ties the full method.

Method	1 error	2 errors	3+ errors	Overall
TEG (ours)	**47.3**	**36.7**	**29.6**	**37.0**
w/o error annotation	38.2	34.7	21.1	31.9
w/o masking	40.9	34.3	27.5	33.9
w/o DB execution	40.9	27.3	19.0	27.9
w/o candidate selection	**47.3**	25.9	20.4	28.8
w/o sequential correction	**47.3**	32.3	23.9	33.2

With Opus 4.8 for type prediction and correction, TEG scores 69.1, ErrorLLM 64.5, and Base 63.6.

The aggregate comparison includes different behavior on empty predictions: TEG makes an unmasked correction call, whereas ErrorLLM retains the issued query. On queries with nonempty predictions, TEG leads ErrorLLM by 1.1–12.6 points across the five locally hosted configurations; with Opus 4.8, the corresponding accuracies are 83.9 and 81.6. These results compare complete correction pipelines, including their fallback policies. With oracle types on Opus 4.8, ErrorLLM and TEG achieve similar execution accuracy at 73.6 and 72.7, respectively. The fallback contributes substantially to the predicted-type lead at Qwen2.5-14B, Qwen3-8B with thinking OFF, and Opus 4.8. These comparisons do not isolate the effect of masking. Appendix D adds prediction and category analyses.

4.4 Ablation study and analysis

Ablation study. Table 4 compares TEG with ablation variants using Qwen2.5-7B-Instruct. The w/o masking variant retains the error annotations and candidate-generation and selection procedures, but leaves the query unmasked. Accuracy falls from 47.3 to 40.9 on single-error queries and from 37.0 to 33.9 overall. The variant also disables the mask-failure retry branch on multi-error queries, and mask-dependent edit instructions can change. These results support the masking-based editing configuration within TEG; they do not isolate the effect of masking the input alone. Removing the annotation as well lowers single-error and overall accuracy to 38.2 and 31.9. This variant also drops sequential correction and does not match compute on multi-error queries, so the rows measure interacting components rather than additive contributions.

Correcting all errors in one pass with cumulative masks, while retaining candidate selection, lowers accuracy from 36.7 to 32.3 on two errors and from 29.6 to 23.9 on three or more, for 33.2 overall. Single-error accuracy is unchanged because both variants take one correction step. These results favor sequential re-grounding: each step applies type-specific rules to the query produced by earlier corrections. The one-pass variant lists all annotations together and uses the first annotation's category to select a single prompt template.

The w/o candidate selection variant uses one generation call per correction step without the crafter or refiner; it ties TEG at 47.3 on single-error queries but falls to 28.8 overall. The w/o DB execution variant retains multiple candidates but limits refinement to one round without execution feedback, lowering accuracy to 40.9 on single-error queries and 27.9 overall. A single candidate per step retains

Table 5: **TEG leads on Table, Condition, and Attribute errors with Qwen2.5-7B.** Entries are single-error execution accuracies (%) by oracle error category with Qwen2.5-7B-Instruct. In the Error types column, ✓ denotes supplied oracle error types from NL2SQL-BUGs; ✗ denotes no error types. Others pools the less frequent categories. Bold marks the best value per column, including ties. The largest gains over Base are on Table and Condition errors.

Method	Error types	Attribute	Value	Condition	Table	Others	Overall
Base	✗	24.3	41.4	29.4	33.3	16.7	30.0
CoT	✗	35.1	37.9	52.9	26.7	16.7	35.5
Self-Debug (Simple)	✗	24.3	41.4	35.3	53.3	16.7	33.6
Self-Debug (Expl)	✗	24.3	37.9	35.3	33.3	8.3	29.1
DIN-SQL	✗	10.8	**51.7**	23.5	13.3	16.7	24.5
MAC-SQL	✗	10.8	27.6	23.5	6.7	**25.0**	18.2
SQLFixAgent	✗	24.3	41.4	41.2	40.0	16.7	32.7
ErrorLLM	✓	16.2	44.8	41.2	60.0	16.7	33.6
TEG (ours)	✓	**37.8**	**51.7**	**58.8**	**66.7**	**25.0**	**47.3**

TEG's single-error lead over the baselines; candidate selection helps on multi-error queries. On single-error queries, the w/o DB execution variant trails the single-candidate variant (40.9 versus 47.3), showing that generating more candidates does not by itself ensure better correction.

Per-category results. TEG outperforms every baseline on Table, Condition, and Attribute errors with Qwen2.5-7B (Table 5). The largest gains over Base are on Table errors (66.7 versus 33.3) and Condition errors (58.8 versus 29.4). On Attribute errors, TEG reaches 37.8, compared with 24.3 for Base and 16.2 for ErrorLLM, which receives the same error types. TEG also ties DIN-SQL on Value errors at 51.7 and MAC-SQL on Others at 25.0, so its lead varies across categories. For maskable sub-types in these categories, the *where* rule focuses masking on the relevant construct classes—table and join nodes, column expressions, or `WHERE` predicates—while the *how* rule specifies the edit. This direct correspondence may help explain the gains. For the *Missing* group, TEG requests insertion without masking. These results support the effectiveness of the full TEG pipeline in these categories, but do not isolate masking from differences in edit instructions and candidate selection.

Computational cost. The multi-error gain from several candidates costs extra model computation. With Qwen2.5-7B-Instruct, mean summed model-call time per query is 42.9s for TEG, 8.6s for the single-candidate variant, and 3.3s for Base, excluding execution and preprocessing. Compared with the single-candidate variant, TEG gains 10.8 points on two errors and 9.2 on three or more, while the two methods have the same accuracy on single-error queries.

5 Conclusion and Discussion

We introduced error grounding for SQL correction: mapping *why* a query is wrong to construct classes (*where*) and edit instructions (*how*) that shape the LLM's correction input. On NL2SQL-BUGs, TEG leads the main comparisons even with shared annotations and outperforms direct correction and ErrorLLM with predicted types on single-error queries. The w/o masking variant retains the types but lowers single-error accuracy by 6.4 points with Qwen2.5-7B. These results support using error types to construct correction inputs through masking and edit instructions.

Beyond SQL. Extending error grounding beyond SQL would retain the mapping from error types to editable structures (*where*) and requested operations (*how*). In general-purpose code, an API-misuse diagnosis could select call expressions and their arguments for replacement. In configuration files, an incorrect-value diagnosis could select fields for replacement, whereas a missing-key diagnosis would request insertion without masking. Each domain would need identifiable constructs, its own grounding rules and correction order, and candidate checks such as tests, type checks, or schema validation. These extensions remain untested; Appendix E discusses their requirements.

Limitations. We evaluate on one SQL benchmark, also used to select the correction order; predicted-type evaluation covers only single-error queries. TEG requires diagnoses and database access and adds inference cost. Execution accuracy does not guarantee semantic equivalence.

AI USE STATEMENT

AI tools assisted manuscript drafting and editing, consistency checks, and the development of analysis and experiment scripts, including ablations, aggregation, and diagnostics. They also assisted interpretation of experimental outputs. The authors reviewed the assisted work and take responsibility for the manuscript, code, and reported results.

REPRODUCIBILITY STATEMENT

The implementation of TEG, including the masking rules, the prompt templates, the error-type prediction prompt, the ablation scripts, and the evaluation script, together with the evaluation dataset, the BIRD development schemas, and the predicted error types, is available at https://anonymous.4open.science/r/TEG-4AC1. The appendix documents the masking rules, prompts, evaluation dataset, baseline configurations, and hardware. Locally hosted models use greedy decoding, and execution accuracy is measured by set agreement of row tuples with the reference result.

REFERENCES

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023.

Arian Askari, Christian Poelitz, and Xinye Tang. Magic: Generating self-correction guideline for in-context text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 23433–23441, 2025.

Omri Avrahami, Dani Lischinski, and Ohad Fried. Blended diffusion for text-driven editing of natural images. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 18187–18197. IEEE, 2022.

Jipeng Cen, Jiaxin Liu, Zhixu Li, and Jingjing Wang. Sqlfixagent: towards semantic-accurate text-to-sql parsing via consistency-enhanced multi-agent collaboration. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 49–57, 2025.

Saumya Chaturvedi, Aman Chadha, and Laurent Bindschaedler. Sql-of-thought: Multi-agentic text-to-sql with guided error correction. *arXiv preprint arXiv:2509.00581*, 2025.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

QiHong Chen, Jiachen Yu, Jiawei Li, Jiecheng Deng, Justin Tian Jin Chen, and Iftekhar Ahmed. A deep dive into large language model code generation mistakes: What and why? *arXiv preprint arXiv:2411.01414*, 2024a.

Shijie Chen, Ziru Chen, Huan Sun, and Yu Su. Error detection for text-to-sql semantic parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 11730–11743, 2023.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, volume 2024, pp. 8746–8825, 2024b.

Ram Chillarege, Inderpal S Bhandari, Jarir K Chaar, Michael J Halliday, Diane S Moebus, Bonnie K Ray, and Man-Yuen Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on software Engineering*, 18(11):943–956, 1992.

Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Jinshu Lin, Dongfang Lou, et al. C3: Zero-shot text-to-sql with chatgpt. *arXiv preprint arXiv:2307.07306*, 2023.

Yuankai Fan, Tonghui Ren, Can Huang, Zhenying He, and X Sean Wang. Grounding natural language to sql translation with data-based self-explanations. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pp. 29–42. IEEE, 2025.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Yue Gong, Chuan Lei, Xiao Qin, Kapil Vaidya, Balakrishnan Narayanaswamy, and Tim Kraska. Sqlens: An end-to-end framework for error detection and correction in text-to-sql. *Advances in Neural Information Processing Systems*, 38:135571–135604, 2026.

Satya Krishna Gorti, Ilan Gofman, Zhaoyan Liu, Jiapeng Wu, Noël Vouitsis, Guangwei Yu, Jesse C Cresswell, and Rasa Hosseinzadeh. Msc-sql: Multi-sample critiquing small language models for text-to-sql translation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 2145–2160, 2025.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7212–7225, 2022.

Mingqian He, Yongliang Shen, Wenqi Zhang, Qiuying Peng, Jun Wang, and Weiming Lu. Star-sql: Self-taught reasoner for text-to-sql. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 24365–24375, 2025.

Zijin Hong, Hao Chen, Zheng Yuan, Qinggang Zhang, Luyao Zhuang, Qing Liao, Feiran Huang, Yangqiu Song, and Xiao Huang. Errorllm: Modeling sql errors for text-to-sql refinement. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pp. 1710–1721, 2026.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International conference on learning representations*, volume 2024, pp. 32808–32824, 2024.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2): 1–72, 2026.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213, 2022.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, et al. Training language models to self-correct via reinforcement learning. In *International Conference on Learning Representations*, volume 2025, pp. 54523–54549, 2025.

Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 337–353, 2025.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. In *International Conference on Learning Representations*, volume 2025, pp. 28691–28735, 2025.

Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. Alpha-sql: Zero-shot text-to-sql using monte carlo tree search. *arXiv preprint arXiv:2502.17248*, 2025.

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

Jia Li, Ruiqi Bai, Yangkang Luo, Yiran Zhang, Wentao Yang, Zeyu Sun, Tiankuo Zhao, Dongming Jin, Lei Li, and Zhi Jin. Bridging the gap between user intent and llm: A requirement alignment approach for code generation. *arXiv preprint arXiv:2604.16198*, 2026a.

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

Jinyang Li, Xiaolong Li, Ge Qu, Per Jacobsson, Bowen Qin, Binyuan Hui, Shuzheng Si, Nan Huo, Xiaohan Xu, Yue Zhang, et al. Swe-sql: Illuminating llm pathways to solve user sql issues in real-world applications. *Advances in Neural Information Processing Systems*, 38:97085–97120, 2026b.

Weibin Liao, Xin Gao, Tianyu Jia, Rihong Qiu, Yifan Zhu, Yang Lin, Xinyu Ma, Junfeng Zhao, and Yasha Wang. Learnat: Learning nl2sql with ast-guided task decomposition for large language models. In *International Conference on Learning Representations*, volume 2026, pp. 87420–87452, 2026.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572, 2023.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. A survey of text-to-sql in the era of llms: Where are we, and where are we going? *IEEE Transactions on Knowledge and Data Engineering*, 2025a.

Xinyu Liu, Shuyu Shen, Boyan Li, Nan Tang, and Yuyu Luo. Nl2sql-bugs: A benchmark for detecting semantic errors in nl2sql translation. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pp. 5662–5673, 2025b.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 3806–3824, 2023.

Mohammadreza Pourreza and Davood Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in neural information processing systems*, 36:36339–36348, 2023.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan Arik. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. In *International Conference on Learning Representations*, volume 2025, pp. 60385–60415, 2025.

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

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*, 2024.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, et al. Mac-sql: A multi-agent collaborative framework for text-to-sql. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 540–557, 2025a.

Pengfei Wang, Baolin Sun, Xuemei Dong, Yaxun Dai, Hongwei Yuan, Mengdie Chu, Yingqi Gao, Xiang Qi, Peng Zhang, and Ying Yan. Agentar-scale-sql: Advancing text-to-sql through orchestrated test-time scaling. *arXiv preprint arXiv:2509.24403*, 2025b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Bo Xu, Shufei Li, Hongyu Jing, Ming Du, Hui Song, Hongya Wang, and Yanghua Xiao. Boosting text-to-sql through multi-grained error identification. In *Proceedings of the 31st international conference on computational linguistics*, pp. 4282–4292, 2025a.

Wenbo Xu, Haifeng Zhu, Liang Yan, Chuanyi Liu, Peiyi Han, Shaoming Duan, and Jeff Z Pan. Ts-sql: Test-driven self-refinement for text-to-sql. In *EMNLP (Findings)*, pp. 2864–2889, 2025b.

Zhenyu Xu and Victor S Sheng. Lecprompt: A prompt-based approach for logical error correction with codebert. *arXiv preprint arXiv:2410.08241*, 2024.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

Bo Yang, Yinfen Xia, Weisong Sun, and Yang Liu. Hallucination detection for llm-based text-to-sql generation via two-stage metamorphic testing. *arXiv preprint arXiv:2512.22250*, 2025b.

Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, pp. 10799–10808. PMLR, 2020.

Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas S Huang. Generative image inpainting with contextual attention. In *2018 IEEE/CVF conference on computer vision and pattern recognition*, pp. 5505–5514. IEEE, 2018a.

Licheng Yu, Zhe Lin, Xiaohui Shen, Jimei Yang, Xin Lu, Mohit Bansal, and Tamara L Berg. Mattnet: Modular attention network for referring expression comprehension. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1307–1315. IEEE, 2018b.

Qinggang Zhang, Hao Chen, Junnan Dong, Shengyuan Chen, Feiran Huang, and Xiao Huang. Structure guided large language model for sql generation. *arXiv preprint arXiv:2402.13284*, 2024.

Appendix

A A Closer Look at TEG **15**

A.1 Why: Error taxonomy and edit groups 15

A.2 Where: Masking rules and algorithms 16

A.3 How: Prompt templates for each edit 20

A.4 Execution-based candidate selection 20

B Experimental Details **22**

B.1 Dataset statistics and examples 22

B.2 Qwen3-8B thinking modes 22

B.3 Baseline implementations 23

B.4 Ablation variants 24

B.5 Computing resources 24

C Additional Main Results **25**

C.1 Single-error results by error category 25

C.2 Multi-error results by error category 26

C.3 Baselines given oracle error types 27

D Additional Results on End-to-end Correction **30**

D.1 Accuracy of predicted error types 30

D.2 Single-error results by error category with predicted types 30

E Extended Discussion **32**

E.1 Using error diagnoses to construct correction inputs 32

E.2 Relation to visual grounding and inpainting 32

E.3 Carrying error grounding beyond SQL 32

A A Closer Look at TEG

This section expands the description of TEG in Section 3. Sections A.1 to A.3 give the grounding rules behind the *why*, the *where*, and the *how*: the taxonomy and its edit groups, the masking rules and algorithms for each error type, and the prompt templates that state each edit. Section A.4 covers candidate selection, which turns a grounded prompt into the corrected query by issuing it under the question and its paraphrases, executing the candidates, and keeping or refining one.

A.1 Why: Error taxonomy and edit groups

Table 6 lists the 31 sub-types supported by the masking and prompt rules across nine categories. Of these, 29 occur in our 549-query NL2SQL-BUGs evaluation subset (Liu et al., 2025b). *Join Type Mismatch* and *Window Functions* have no annotations in this subset. For prompt construction and the correction order, the implementation groups sub-types under *Mismatch* (M), *Redundancy* (R), *Missing* (Mi), or *Rewrite* (Rw). These groups summarize the requested edits. For sub-types such as the function errors, *ASC/DESC*, and *DISTINCT*, the prompt also depends on whether the corresponding construct is present and can be masked. *Mismatch* is the largest group, accounting for 64.7% of all error annotations, followed by *Missing* (21.3%), *Redundancy* (13.5%), and *Rewrite* (0.5%).

Table 6: **Four edit groups turn the 31 taxonomy sub-types into edit operations.** Sub-types follow the nine NL2SQL-BUGs categories. Edit groups: M = Mismatch (replace), R = Redundancy (remove), Mi = Missing (insert), Rw = Rewrite (restructure). Mismatch covers 64.7% of all annotations, so replacement is the most frequent requested edit.

Category	Sub-type	Edit group
Attribute-Related	Attribute Mismatch	M
	Attribute Redundancy	R
	Attribute Missing	Mi
Table-Related	Table Mismatch	M
	Table Redundancy	R
	Table Missing	Mi
	Join Condition Mismatch	M
	Join Type Mismatch	M
Value-Related	Value Mismatch	M
	Data Format Mismatch	M
Operator-Related	Comparison Operator Mismatch	M
	Logical Operator Mismatch	M
Condition-Related	Explicit Condition Missing	Mi
	Implicit Condition Missing	Mi
	Explicit Condition Mismatch	M
	Explicit Condition Redundancy	R
Function-Related	Aggregate Functions	M
	Window Functions	M
	Date/Time Functions	M
	Conversion Functions	M
	Math Functions	M
	String Functions	M
	Conditional Functions	M
Clause-Related	Clause Missing	Mi
	Clause Redundancy	R
Subquery-Related	Subquery Missing	Mi
	Subquery Mismatch	M
	Partial Query	Rw
Other	ASC/DESC	M
	DISTINCT	M
	Other	Rw

A.2 WHERE: MASKING RULES AND ALGORITHMS

Deciding which SQL constructs each error type implicates is part of the method: the taxonomy names the kind of error, and the rules below name the SQL constructs that can carry it. Table 7 gives that mapping for every error category and sub-type. For maskable sub-types, SQLGlot traversal identifies matching node classes and replaces their occurrences with <MASK>. The annotation supplies an error type, not the identity of a unique faulty node. *Attribute-Related* masking extracts column expressions through `Column.sql()` and replaces their SQL strings, whereas the other categories use AST node replacement. *Table-Related* rows describe multi-error mode, the rules for queries with several annotations. Single-error mode, for queries with one annotation, masks both `Table` and `Join` nodes for every non-*Missing Table-Related* sub-type. Comparison-operator masks are restricted to `Where`/`Having`, and logical masks apply to outermost logical expressions. Generic function masks exclude the specialized function classes listed separately. Entries marked with "–" correspond to sub-types in the *Missing* or *Rewrite* edit groups, which do not select existing constructs for masking. The query remains unmasked, and the edit instruction requests insertion or broader restructuring.

Table 7: **Each maskable error sub-type selects the AST node classes that TEG masks.** Targets are SQLGlot node classes, and Table-Related rows describe multi-error mode. "–" marks sub-types in the Missing and Rewrite edit groups, which pass the query through unmasked. The rule names a node class rather than a unique faulty node, so the LLM still decides which masked construct to change.

Category	Sub-type	Target AST Nodes
Attribute	Mismatch, Redundancy	`Column` (string-level replace)
	Missing	–
Table	Mismatch, Redundancy	`Table`
	Join Cond./Type Mismatch	`Join`
	Missing	–
Value	Mismatch, Data Format	`Literal`, `String`, `Int64`
Operator	Comparison Mismatch	`EQ`, `GT`, `LT`, `GTE`, `LTE`, `NEQ`, `Like`, `ILike`, `In`, `Between`, `Is`
	Logical Mismatch	`And`, `Or`, `Not`, `Xor`
Condition	Mismatch, Redundancy	`Where`, plus `Having` in multi-error mode
	Missing	–
Function	Aggregate	`AggFunc`, `Group`, `Having`
	Window	`Window`
	Conversion	`Cast`
	Conditional	`Case`, `If`
	Date/Time, Math, String	`Func`
Clause	Redundancy	`Group`, `Order`, `Having`, `Limit`
	Missing	–
Subquery	Mismatch	`Subquery`
	Missing, Partial	–
Other	ASC/DESC	`Ordered`
	DISTINCT	`Distinct`
	Other (Rw)	–

Masking algorithms. Algorithm 2 gives the generic procedure and Algorithms 3 to 11 the per-category rules that decide which SQL constructs an error type implicates. Given an issued SQL q and its error category and sub-type, TEG parses q into an AST via SQLGlot, looks up the target node classes of Table 7, and replaces every matching node with <MASK>. For sub-types in the *Missing* group and for *Partial Query*, TEG returns the input query without parsing or masking. For *Other*, it parses and serializes the query but selects no construct to mask. If this produces no placeholder and a previous correction step exists, TEG uses the retry template described in Section A.3. All serialization uses the SQLite dialect, and parsing or masking exceptions return the query unchanged. Attribute masking operates on the SQL string as written in single-error mode and on the SQLGlot-serialized SQL in multi-error mode. Condition masking targets `Where` nodes and, in multi-error mode, `Having`

nodes as well. Table masking distinguishes `Table` from `Join` targets in multi-error mode, whereas single-error mode masks both for every non-*Missing Table-Related* sub-type. Two rules go beyond a plain node-class lookup. For *Aggregate Functions*, TEG first masks `AggFunc`, `Group`, and `Having` nodes, then locates the first `Where` subtree in SQLGlot traversal order. Within that subtree, each comparison containing a placeholder is masked in full, so the LLM can regenerate the predicate together with the aggregate. Comparison masks are limited to `Where` and `Having`, and logical masks cover only the outermost logical expression, so nested operators are masked once.

Algorithm 2 MASKING(q, t, s)

Require: SQL query q, error category t, sub-type s
Ensure: Masked query $\tilde{q}$

if s is a *Missing* sub-type or s = *Partial Query* **then**
  return q
end if
AST ← PARSE(q)
if parse fails **then return** q
end if
$\mathcal{N}$ ← TARGETNODES(t, s)
if $\mathcal{N} = \emptyset$ **then**
  return AST.TOSQL() ▷ Other: no mask
end if
for all node $v \in$ AST matching $\mathcal{N}$ **do**
  replace v with <MASK>
end for
return AST.TOSQL()

Algorithm 3 MASKCLAUSE(q, s)

Require: SQL q, sub-type s

if s = *Clause Missing* **then**
  return q
end if
AST ← PARSE(q)
for all $v \in$ AST.FINDALL(`Group`, `Order`, `Having`, `Limit`) **do**
  v.REPLACE(<MASK>)
end for
return AST.TOSQL()

Algorithm 4 MASKATTRIBUTE(q, s)

Require: SQL q, sub-type s

if s = *Attribute Missing* **then**
  return q
end if
AST ← PARSE(q)
$C \leftarrow \{c.\text{SQL}() \mid c \in \text{AST.FINDALL}(\texttt{Column})\}$
for all $c \in C$ **do**
  $q \leftarrow q$.REPLACE(c, <MASK>) ▷ String-level replacement
end for
return q

Algorithm 5 MaskSubquery(q, s)

Require: SQL q, sub-type s
1: **if** $s \in$ {*Subquery Missing*, *Partial Query*} **then**
2: **return** q
3: **end if**
4: AST $\leftarrow$ Parse(q)
5: **for all** $v \in$ AST.FindAll(`Subquery`) **do**
6: v.Replace(<MASK>)
7: **end for**
8: **return** AST.ToSQL()

Algorithm 6 MaskTable(q, s)

Require: SQL q, sub-type s
1: **if** s = *Table Missing* **then**
2: **return** q
3: **end if**
4: AST $\leftarrow$ Parse(q)
5: **if** $s \in$ {*Table Mismatch*, *Table Redundancy*} **then** ▷ Single-error mode masks both
6: **for all** $v \in$ AST.FindAll(`Table`) **do**
7: v.Replace(<MASK>)
8: **end for**
9: **else if** $s \in$ {*Join Condition Mismatch*, *Join Type Mismatch*} **then**
10: **for all** $j \in$ AST.FindAll(`Join`) **do**
11: j.Replace(<MASK>)
12: **end for**
13: **end if**
14: **return** AST.ToSQL()

Algorithm 7 MaskValue(q, s)

Require: SQL q, sub-type s
1: AST $\leftarrow$ Parse(q)
2: $\mathcal{L} \leftarrow$ AST.FindAll(`Literal`, `String`, `Int64`)
3: **for all** $\ell \in \mathcal{L}$ **do**
4: ℓ.Replace(<MASK>)
5: **end for**
6: **return** AST.ToSQL()

Algorithm 8 MaskOther(q, s)

Require: SQL q, sub-type s
1: AST $\leftarrow$ Parse(q)
2: **if** s = *ASC/DESC* **then**
3: Mask all `Ordered` nodes
4: **else if** s = *DISTINCT* **then**
5: Mask all `Distinct` nodes
6: **else**
7: Leave the AST unmasked ▷ Other (Rewrite)
8: **end if**
9: **return** AST.ToSQL()

Algorithm 9 MaskOperator(q, s)

Require: SQL q, sub-type s
AST ← Parse(q)
if s = *Comparison Operator Mismatch* **then**
    $\mathcal{S}$ ← AST.FindAll(`Where`, `Having`)
    for all scope ∈ $\mathcal{S}$ **do**
        for all comparison c ∈ scope **do**
            c.Replace(<MASK>)
        end for
    end for
else if s = *Logical Operator Mismatch* **then**
    for all v ∈ AST.FindAll(`And`, `Or`, `Not`, `Xor`) **do**
        if parent of v is not a logical operator **then** ▷ Avoid double-masking
            v.Replace(<MASK>)
        end if
    end for
end if
return AST.ToSQL()

Algorithm 10 MaskCondition(q, s)

Require: SQL q, sub-type s
if s ∈ {*Explicit Condition Missing*, *Implicit Condition Missing*} **then**
    return q
end if
AST ← Parse(q)
$\mathcal{T}$ ← {`Where`}, plus `Having` in multi-error mode
for all w ∈ AST.FindAll($\mathcal{T}$) **do**
    w.Replace(<MASK>)
end for
return AST.ToSQL()

Algorithm 11 MaskFunction(q, s)

Require: SQL q, sub-type s
AST ← Parse(q)
if s = *Aggregate Functions* **then**
    Mask all `AggFunc`, `Group`, `Having` nodes
    w ← first `Where` subtree, if present
    Mask comparisons within w containing <MASK>, if w exists
else if s = *Window Functions* **then**
    Mask all `Window` nodes
else if s = *Conversion Functions* **then**
    Mask all `Cast` nodes
else if s = *Conditional Functions* **then**
    Mask all `Case`, `If` nodes
else ▷ Date/Time, Math, String Functions
    Mask all `Func` nodes except `AggFunc`, `Cast`, `Convert`, `Case`, `If`, `Window`
end if
return AST.ToSQL()

A.3 How: Prompt templates for each edit

The correction prompt is constructed from four components: the (possibly masked) SQL query $\tilde{q}$, the natural language question n, the database schema S, and the error category and sub-type (t, s). TEG uses five correction templates, each covering one or more error categories, plus a retry template for multi-error steps whose mask is empty and a base template for queries without an annotation. The masked query replaces the issued query in the prompt, and no separate copy of the unmasked query is supplied. Every template that uses an annotation supplies its category and sub-type labels without the taxonomy definitions, and the model returns the corrected query as JSON: `{"sql": "..."}`.

Default template. Used for most categories (*Attribute-Related*, *Table-Related*, *Value-Related*, *Condition-Related*, *Clause-Related*). The instruction reflects the requested edit. For *Mismatch* errors, the model is asked to fix the masked part. For *Redundancy*, it is asked to delete the masked part. For *Missing* errors, it is asked to insert the absent SQL fragment at the appropriate position.

Operator-Related template. If masked parts are present, the model fixes or deletes them. If no mask is present, the model inserts the appropriate operator (e.g., `AND`, `OR`, comparison operators).

Function-Related template. If masked parts are present, the model fixes or deletes them. If no mask is present, the model inserts the appropriate function (e.g., `CAST`, `CASE`, aggregate functions).

Subquery-Related template. The instruction follows the same *Mismatch*/*Redundancy*/*Missing* structure as the default template, with two additions: *Missing* errors specifically prompt insertion of the missing subquery, and *Partial Query* errors prompt completion of an incomplete SQL query.

Other template. If masked parts are present, the model fixes or deletes them. If no mask is present, the behavior depends on the sub-type. *ASC/DESC* errors prompt insertion of the missing fragment. *DISTINCT* errors prompt a rewrite that may involve adding or removing the `DISTINCT` keyword, or restructuring joins, `SELECT` columns, or grouping. *Other* errors prompt a rewrite from scratch unless the retry condition below applies.

Retry template. When a masking attempt yields no <MASK> and a previous correction step exists, the step uses this template with the unmasked current query. It lists the previous and current annotations, describes the first as a previously failed correction attempt and the second as the error to fix, and asks for a regenerated query. It carries no operation-specific instruction. At the first step, an empty mask falls back to the correction template for the current error category instead.

Base template. The base template supplies the query, question, and schema with no error information and asks the model to regenerate the query if it is semantically misaligned. It is the prompt of the Base rows in the tables, TEG's one-call fallback when no error type is predicted, and the correction prompt of the w/o error annotation ablation, which keeps paraphrases and execution-based selection.

A.4 Execution-based candidate selection

Grounding determines the input of a correction step, and candidate selection decides its output. A single answer to a grounded prompt can name a column or value that the database does not contain, so TEG generates several candidate queries, filters them by executing them, and asks an LLM to select or refine one, a stage developed from SQLFixAgent (Section 3.2). The crafter is called once per query and asked for four paraphrases of the natural language question, of which the first three parsed paraphrases are kept, so each correction step issues its prompt at most four times, under the original question and under each paraphrase, and yields at most four candidate queries; every step reuses the same paraphrases with its current query and edit instruction. Candidates are executed against the database with a 5s timeout per candidate. The execution filter prefers candidates that run, return at least one row, and contain no NULL in the first five returned rows. The filter is a heuristic, since valid results can be empty or contain NULL. If none passes, all generated candidates remain eligible, and duplicate SQL strings are then collapsed. With no generated candidate, TEG retains the current query. With one distinct candidate, it returns that candidate directly, even if it failed the execution filter. With multiple candidates, the refiner receives the question, schema, candidate SQL strings, and

accumulated failure messages, each recording the selected SQL, the SQLite error, and the exception class. The database-content field of the SQLFixAgent prompt is left empty. The refiner returns a reasoning string and one SQL, selected or rewritten, for up to three rounds, and stops when its SQL passes the same execution filter or times out. The timeout stop is inherited from the SQLFixAgent implementation and does not certify correctness. A round that returns no SQL is skipped. If no round meets either stopping condition, TEG returns the last candidate in the pool. Evaluation still scores timeouts as incorrect. The crafter and refiner prompts are those of SQLFixAgent (Cen et al., 2025), with its similar-example memory left uninitialized, which is the authors' no-memory branch.

B Experimental Details

This section extends Section 4.1 with the data, the Qwen3-8B thinking modes, the baseline implementations, the ablation variants of Section 4.4, and the hardware.

B.1 Dataset statistics and examples

Our evaluation subset consists of 549 issued SQL queries from NL2SQL-BUGs (Liu et al., 2025b), derived from BIRD (Li et al., 2023) across 11 SQLite databases. We retain queries labeled incorrect that have a reference SQL. Across this filtered subset, 1,160 error annotations cover nine categories and 29 observed sub-types. These are the sub-types observed after filtering, not the full dataset taxonomy. Table 8 summarizes the distribution of errors by errors per query, category, and edit group. Most queries (54.1%) contain exactly two errors, and single-error queries are 20.0%.

Table 8: **Most evaluation queries carry more than one error.** The 1,160 error annotations in our 549-query NL2SQL-BUGs subset by errors per query, error category, and edit group. Single-error queries are 20.0% of the subset, and Attribute and Table errors are about half of all annotations.

Distribution	**Count**	**%**
Errors per Query *(549 queries)*		
1 error	110	20.0
2 errors	297	54.1
3 errors	114	20.8
4 errors	26	4.7
5 errors	2	0.4
Error Category *(1,160 annotations)*		
Attribute	340	29.3
Table	247	21.3
Condition	166	14.3
Value	132	11.4
Function	88	7.6
Clause	83	7.2
Other	58	5.0
Subquery	26	2.2
Operator	20	1.7
Edit Group *(1,160 annotations)*		
Mismatch (M)	750	64.7
Redundancy (R)	157	13.5
Missing (Mi)	247	21.3
Rewrite (Rw)	6	0.5

Examples. Table 9 shows representative single-error and multi-error cases from several error categories, with the erroneous and reference spans marked.

B.2 Qwen3-8B thinking modes

Qwen3-8B runs with thinking OFF and ON in every table. Both modes use greedy decoding and the same correction procedures. OFF disables thinking in the chat template and permits up to 4,096 generated tokens per model call. ON adds a 4,096-token thinking budget, for a total generation limit of 8,192 tokens. The implementation forces the end-of-thinking token when that budget is reached. The modes therefore differ in their available generation budget.

Every method improves with thinking ON, in the main comparison (Table 1), with predicted types (Table 3), and with baselines given error types (Table 13), and TEG leads in both modes in each.

Table 9: **An error can sit in a column, a value, or several places at once.** Representative single-error and multi-error queries from NL2SQL-BUGs. Red marks the erroneous span in the issued SQL and green the corresponding span in the reference SQL. Example 3 needs both an insertion and a replacement, which TEG handles in two sequential steps.

Example 1: Attribute Mismatch (single error)	db: california_schools
Question	What is the unabbreviated mailing street address of the school with the highest FRPM count for K-12 students?
Issued SQL	`SELECT schools.streetabr FROM frpm INNER JOIN schools ON frpm.cdscode = schools.cdscode ORDER BY frpm.‘frpm count (k-12)‘ DESC LIMIT 1`
Reference SQL	`SELECT T2.MailStreet FROM frpm AS T1 INNER JOIN schools AS T2 ON T1.CDSCode = T2.CDSCode ORDER BY T1.‘FRPM Count (K-12)‘ DESC LIMIT 1`

Example 2: Value Mismatch (single error)	db: codebase_community
Question	Which user has the website URL listed at ’http://stackoverflow.com’
Issued SQL	`SELECT displayname FROM users WHERE websiteurl = 'http://stackoverflow.com/u/1114'`
Reference SQL	`SELECT displayname FROM users WHERE websiteurl = 'http://stackoverflow.com'`

Example 3: Attribute Missing + Attribute Mismatch (multi-error)	db: california_schools
Question	Please specify all of the schools and their related mailing zip codes that are under Avetik Atoian’s administration.
Issued SQL	`SELECT mailzip FROM schools WHERE admfname1 = 'Avetik' AND admlname2 = 'Atoian'`
Reference SQL	`SELECT school, mailzip FROM schools WHERE admlname1 = 'Atoian' AND admfname1 = 'Avetik'`

B.3 Baseline implementations

The baselines share TEG’s LLM and decoding settings within each comparison column. Base and the reasoning and SQL-specific baselines are evaluated both with and without oracle error types. ErrorLLM always receives the same annotations as TEG. We re-implemented the methods and chose iteration counts in preliminary experiments. TEG’s crafter and refiner reuse the SQLFixAgent authors’ prompts, whereas the reported SQLFixAgent baseline uses our revised prompts. The reported rows are as follows.

- **CoT** (Kojima et al., 2022): The model is prompted to analyze what the question requires, where the given SQL deviates from it, and what fix is needed, then return its reasoning and corrected SQL. The main comparison uses no error-type information or masking.
- **DIN-SQL** (Pourreza & Rafiei, 2023): A full NL2SQL pipeline that decomposes the task into schema linking, query classification, and SQL generation, using the issued SQL as context. The generated SQL is then reviewed and revised via a self-correction stage. The schema context includes up to three sample rows per table when available; generated queries are not executed for correction feedback.
- **Self-Debug (Simple)** (Chen et al., 2024b): The simple-feedback variant, which uses no execution signal. After an initial single-pass correction, each turn asks the model to judge whether its current query is correct and, if not, to emit a revision. We allow up to 3 feedback-correction turns and stop early as soon as the model judges the current query correct.
- **Self-Debug (Expl)** (Chen et al., 2024b): Adds execution feedback and self-explanation on top of the simple variant. Each turn executes the current query and issues three calls: one explaining what the query returns given its execution result, one inferring what the question asks for, and one comparing the two to judge and revise. Up to 3 turns with the same early stopping.
- **MAC-SQL** (Wang et al., 2025a): Our adaptation supplies the issued SQL to a Decomposer that decomposes the question and generates corrected SQL. A Selector prunes the schema only when

its representation exceeds 3,000 tokens, counted with a Qwen2.5 tokenizer. A Refiner revises the generated SQL on execution errors, timeouts, or empty results for up to 3 rounds, stopping when execution returns a nonempty result.

- **SQLFixAgent** (Cen et al., 2025): A multi-agent pipeline that first repairs the issued SQL in a single pass, then prompts a reviewer to check the result clause by clause for semantic errors. Its error check is prompt-based and uses no execution signal, and queries the reviewer passes are returned unchanged. When an error is flagged, a crafter produces up to 3 question paraphrases, and a separate generation call for each paraphrase uses the issued SQL to produce a candidate, keeping those that execute without error, and a refiner then selects or rewrites among the candidates using execution-based verification with a failure memory, for up to 3 rounds. The candidate filter accepts any executable query, including empty results; the refiner stops on a nonempty execution result and records execution failures or empty results in its failure memory for the next attempt. TEG additionally prefers nonempty results without NULL in the first five rows, and keeps all candidates when none passes.
- **ErrorLLM** (Hong et al., 2026): We use a structural refinement adaptation with the same LLM and supplied annotations as TEG. A localization call receives the question, schema, SQLGlot AST with node IDs, and error-specific guidelines, and returns relevant AST nodes, schema elements, and filled analyses. The implementation extracts the corresponding AST subtrees and related schema context. A second call receives these contexts, the filled guidelines, correction examples, and issued SQL, and corrects the supplied errors jointly in priority order. The schema graph uses database metadata and lexical links to the question, rather than the original ELECTRA/RGAT QSS encoder. We do not reproduce the trained error-token detector. When no valid AST-node or schema-element references remain, context extraction falls back to the full AST or schema graph, respectively, and invalid refinement output or an empty prediction retains the issued query. The method uses no database execution feedback during generation.

B.4 Ablation variants

Table 4 removes or alters one part of TEG per row, with Qwen2.5-7B-Instruct and oracle error types. The variants are implemented as follows. The w/o error annotation ablation drops the annotation and therefore masking and sequential correction as well: it issues one correction step with the base template of Section A.3, retaining paraphrases and execution-based candidate selection. The w/o masking ablation keeps the annotations and category-specific templates but leaves the query unmasked and disables the retry branch. Removing the masks can also change which conditional edit instruction applies. The w/o DB execution ablation removes database execution during correction. Candidates are de-duplicated without execution-based filtering. With multiple distinct candidates, the refiner is called once with an empty failure memory, falling back to the last candidate if it returns no SQL. A sole candidate is returned directly; if none is generated, the current query is retained. The w/o candidate selection ablation removes the crafter and the refiner, leaving one prompt and one candidate per correction step. The w/o sequential correction ablation applies the masks of all annotations cumulatively to one query and issues a single prompt built from the template of the first annotation's category, listing every annotation. The candidate selection stage is unchanged and runs once on that single prompt.

B.5 Computing resources

The locally hosted Qwen2.5 experiments were conducted on a compute node with an AMD EPYC 9654 96-core processor and four NVIDIA RTX PRO 6000 Blackwell GPUs (96 GB VRAM each). Each local experimental run used a single dedicated GPU. Opus 4.8 was accessed through an API on hardware we did not control.

C Additional Main Results

This section extends Section 4.2 with per-category single-error and multi-error results and the full results for the baselines given oracle error types.

C.1 Single-error results by error category

Tables 10 and 11 break down the main single-error comparison by the oracle error category for all five Qwen settings. Access to error types matches the main comparison: Base, CoT, Self-Debug, DIN-SQL, MAC-SQL, and SQLFixAgent receive no error types, while ErrorLLM and TEG receive oracle error types. Each block uses the same single-error queries, comprising Attribute, Value, Condition, Table, and Others. Others combines *Operator-Related*, *Function-Related*, *Clause-Related*, and catch-all Other errors. There are no *Subquery-Related* queries in the single-error subset. Overall accuracy weights queries equally rather than averaging the category accuracies.

The relative performance varies by category. TEG leads or ties on Attribute errors in all five Qwen settings. DIN-SQL is stronger on Value errors at Qwen2.5-14B and 32B. ErrorLLM leads on Condition and Table errors at 14B with 70.6 and 73.3, compared with 58.8 and 66.7 for TEG, despite TEG's higher overall accuracy. The small Table and Others subsets contain 15 and 12 queries, respectively, so a difference of one or two queries can change their category rankings.

Table 10: **TEG leads or ties on Attribute errors at every Qwen2.5 scale.** Single-error execution accuracy by oracle error category on Qwen2.5-Instruct. In the Error types column, ✓ denotes supplied oracle error types from NL2SQL-BUGs; ✗ denotes no error types. Others pools the less frequent categories, and Overall counts all queries. Bold marks the best value per column within each model block, including ties. Other methods win smaller categories, such as DIN-SQL on Value errors at 14B and 32B.

Method	Error types	Attribute	Value	Condition	Table	Others	Overall
Qwen2.5-7B							
Base	✗	24.3	41.4	29.4	33.3	16.7	30.0
CoT	✗	35.1	37.9	52.9	26.7	16.7	35.5
Self-Debug (Simple)	✗	24.3	41.4	35.3	53.3	16.7	33.6
Self-Debug (Expl)	✗	24.3	37.9	35.3	33.3	8.3	29.1
DIN-SQL	✗	10.8	**51.7**	23.5	13.3	16.7	24.5
MAC-SQL	✗	10.8	27.6	23.5	6.7	**25.0**	18.2
SQLFixAgent	✗	24.3	41.4	41.2	40.0	16.7	32.7
ErrorLLM	✓	16.2	44.8	41.2	60.0	16.7	33.6
TEG (ours)	✓	**37.8**	**51.7**	**58.8**	**66.7**	**25.0**	**47.3**
Qwen2.5-14B							
Base	✗	27.0	37.9	41.2	40.0	**41.7**	35.5
CoT	✗	24.3	37.9	35.3	66.7	**41.7**	37.3
Self-Debug (Simple)	✗	37.8	37.9	47.1	40.0	25.0	38.2
Self-Debug (Expl)	✗	29.7	41.4	29.4	40.0	**41.7**	35.5
DIN-SQL	✗	24.3	**65.5**	58.8	33.3	33.3	42.7
MAC-SQL	✗	8.1	31.0	11.8	0.0	25.0	15.5
SQLFixAgent	✗	29.7	41.4	35.3	46.7	33.3	36.4
ErrorLLM	✓	43.2	48.3	**70.6**	**73.3**	33.3	51.8
TEG (ours)	✓	**48.6**	55.2	58.8	66.7	33.3	**52.7**
Qwen2.5-32B							
Base	✗	32.4	48.3	35.3	66.7	33.3	41.8
CoT	✗	40.5	37.9	47.1	80.0	33.3	45.5
Self-Debug (Simple)	✗	40.5	55.2	41.2	66.7	**41.7**	48.2
Self-Debug (Expl)	✗	43.2	44.8	35.3	66.7	33.3	44.5
DIN-SQL	✗	35.1	**79.3**	**58.8**	53.3	**41.7**	53.6
MAC-SQL	✗	13.5	27.6	0.0	26.7	25.0	18.2
SQLFixAgent	✗	37.8	48.3	35.3	66.7	33.3	43.6
ErrorLLM	✓	43.2	51.7	**58.8**	86.7	**41.7**	53.6
TEG (ours)	✓	**56.8**	55.2	52.9	**93.3**	25.0	**57.3**

Table 11: **On Qwen3-8B, TEG leads or ties on Attribute errors with thinking both OFF and ON.** Single-error execution accuracy by oracle error category. In the Error types column, ✓ denotes supplied oracle error types from NL2SQL-BUGs; ✗ denotes no error types. Others pools the less frequent categories, and Overall weights queries equally, not category columns. Bold marks the best value per column within each thinking mode, including ties. Other methods win smaller categories, such as Self-Debug (Simple) on Value errors with thinking ON.

Method	Error types	Attribute	Value	Condition	Table	Others	Overall
Qwen3-8B (Thinking OFF)							
Base	✗	27.0	34.5	29.4	46.7	25.0	31.8
CoT	✗	18.9	37.9	41.2	66.7	16.7	33.6
Self-Debug (Simple)	✗	40.5	44.8	29.4	33.3	**33.3**	38.2
Self-Debug (Expl)	✗	**43.2**	44.8	35.3	66.7	**33.3**	44.5
DIN-SQL	✗	13.5	41.4	17.6	13.3	16.7	21.8
MAC-SQL	✗	8.1	37.9	**47.1**	26.7	16.7	25.5
SQLFixAgent	✗	32.4	44.8	35.3	46.7	25.0	37.3
ErrorLLM	✓	24.3	44.8	41.2	66.7	**33.3**	39.1
TEG (ours)	✓	**43.2**	**48.3**	41.2	**80.0**	16.7	**46.4**
Qwen3-8B (Thinking ON)							
Base	✗	37.8	44.8	35.3	73.3	33.3	43.6
CoT	✗	45.9	44.8	41.2	**80.0**	33.3	48.2
Self-Debug (Simple)	✗	45.9	**55.2**	35.3	**80.0**	**41.7**	50.9
Self-Debug (Expl)	✗	48.6	44.8	41.2	60.0	**41.7**	47.3
DIN-SQL	✗	29.7	**55.2**	41.2	60.0	**41.7**	43.6
MAC-SQL	✗	18.9	41.4	47.1	26.7	16.7	30.0
SQLFixAgent	✗	32.4	41.4	41.2	73.3	**41.7**	42.7
ErrorLLM	✓	29.7	48.3	47.1	**80.0**	**41.7**	45.5
TEG (ours)	✓	**62.2**	48.3	**58.8**	73.3	**41.7**	**57.3**

C.2 Multi-error results by error category

Table 12 reports the 439 multi-error queries by category. TEG leads every reported category and reaches 34.4 overall. A query can occur in several category columns, so the columns overlap.

Table 12: **On multi-error queries, TEG leads every error category and reaches 34.4 overall.** Execution accuracy by error category on Qwen2.5-7B-Instruct over the 439 multi-error queries. ErrorLLM and TEG use oracle types, while the other baselines run without them. A query counts in every category containing one of its errors, so the columns overlap (Attribute 275, Value 103, Condition 140, Table 194, Others 213), and bold marks the best value per column. The lead is largest on Table errors, 43.8 against 28.4 for the next best method.

Method	Attribute	Value	Condition	Table	Others	Overall
Base	19.6	19.4	15.7	24.2	22.1	22.1
Reasoning and self-correction						
CoT (Kojima et al., 2022)	20.7	13.6	15.7	28.4	21.6	21.6
Self-Debug (Simple) (Chen et al., 2024b)	16.7	19.4	17.1	23.2	19.7	19.6
Self-Debug (Expl) (Chen et al., 2024b)	22.2	17.5	15.7	25.3	22.1	22.6
SQL-specific refinement						
DIN-SQL (Pourreza & Rafiei, 2023)	18.5	22.3	13.6	18.6	20.2	18.7
MAC-SQL (Wang et al., 2025a)	11.3	12.6	11.4	12.9	11.3	12.3
SQLFixAgent (Cen et al., 2025)	23.6	20.4	17.9	28.4	24.4	24.6
Taxonomy-guided refinement						
ErrorLLM (Hong et al., 2026)	20.7	17.5	15.7	27.8	17.8	21.6
Taxonomy-guided error grounding						
TEG (ours)	**31.3**	**28.2**	**27.9**	**43.8**	**31.0**	**34.4**

C.3 Baselines given oracle error types

This subsection reports in full the comparisons summarized in Section 4.2. ErrorLLM and TEG use oracle error types, whereas the other baselines in the main comparison do not. Here every baseline receives the same oracle error types, and TEG stays the most accurate in every Qwen setting.

C.3.1 Single-error results

Table 13 reports the single-error comparisons on the same 110 queries and five Qwen settings as Table 1, with every method receiving the oracle error types that ErrorLLM and TEG use.

We add the error types to the prompts that generate or revise the issued SQL, retaining each method's correction procedure, iteration limits, and decoding settings. Base and CoT receive the error types directly in their correction prompts. Self-Debug receives them in its initial correction and feedback prompts. For DIN-SQL and MAC-SQL, we add them to SQL generation and decomposition, respectively. SQLFixAgent receives them in its initial correction and candidate-generation prompts. These changes supply the error types without adding TEG's type-specific masking or sequential correction. ErrorLLM and TEG already use these error types, so their rows are unchanged.

Table 13: **TEG leads single-error correction when all methods receive oracle types.** Single-error execution accuracy. Base, reasoning and self-correction, and SQL-specific refinement methods use prompts augmented with those types, and ErrorLLM and TEG already use the types in their own procedures. Qwen3-8B is evaluated with thinking OFF and ON, and bold marks the best accuracy per Qwen setting. The single-error margin over the strongest baseline given error types ranges from 0.9 points, at Qwen2.5-14B and at Qwen3-8B with thinking OFF, to 11.8 points at Qwen2.5-7B.

	Qwen2.5-Instruct			**Qwen3-8B (thinking)**	
Method	7B	14B	32B	OFF	ON
Base	28.2	36.4	41.8	35.5	48.2
Reasoning and self-correction					
CoT (Kojima et al., 2022)	32.7	40.9	42.7	39.1	52.7
Self-Debug (Simple) (Chen et al., 2024b)	35.5	45.5	50.9	40.9	48.2
Self-Debug (Expl) (Chen et al., 2024b)	28.2	38.2	45.5	45.5	48.2
SQL-specific refinement					
DIN-SQL (Pourreza & Rafiei, 2023)	26.4	40.9	52.7	24.5	46.4
MAC-SQL (Wang et al., 2025a)	17.3	17.3	26.4	11.8	24.5
SQLFixAgent (Cen et al., 2025)	32.7	41.8	46.4	41.8	48.2
Taxonomy-guided refinement					
ErrorLLM (Hong et al., 2026)	33.6	51.8	53.6	39.1	45.5
Taxonomy-guided error grounding					
TEG (ours)	**47.3**	**52.7**	**57.3**	**46.4**	**57.3**

TEG remains above every baseline given error types in all five Qwen settings. On Qwen2.5, Self-Debug (Simple) is the strongest such baseline at 7B with 35.5, while ErrorLLM is strongest at 14B and 32B with 51.8 and 53.6. Giving CoT the same types yields 32.7, 40.9, and 42.7 at 7B, 14B, and 32B, compared with 47.3, 52.7, and 57.3 for TEG. On Qwen3-8B, the strongest such baselines are Self-Debug (Expl) with thinking OFF at 45.5 and CoT with thinking ON at 52.7, compared with TEG at 46.4 and 57.3. Access to error types alone therefore does not account for TEG's advantage. The procedures also differ in their candidate selection stages, including generation and refinement, so these comparisons should be read together with the component ablations when assessing error grounding.

C.3.2 Results by error count

We apply the same prompt augmentation to all 549 queries with Qwen2.5-7B-Instruct. Table 14 reports execution accuracy by error count when every method receives the same oracle error types. The correction procedures, iteration limits, and decoding settings are retained. ErrorLLM and TEG use annotations in their standard procedures, so their results are repeated from the main comparison.

Table 14: **With the same oracle error types for every baseline, TEG leads at every error count.** Execution accuracy on all 549 queries using Qwen2.5-7B-Instruct. Base, reasoning and self-correction, and SQL-specific refinement methods use prompts augmented with those types, and ErrorLLM and TEG already use the types in their own procedures. Bold marks the best value per column. TEG reaches 37.0 overall against 26.0 for the strongest such baseline, Self-Debug (Simple).

Method	1 error	2 errors	3+ errors	Overall
Base	28.2	23.2	16.9	22.6
Reasoning and self-correction				
CoT (Kojima et al., 2022)	32.7	25.9	14.8	24.4
Self-Debug (Simple) (Chen et al., 2024b)	35.5	24.6	21.8	26.0
Self-Debug (Expl) (Chen et al., 2024b)	28.2	26.9	19.7	25.3
SQL-specific refinement				
DIN-SQL (Pourreza & Rafiei, 2023)	26.4	19.9	15.5	20.0
MAC-SQL (Wang et al., 2025a)	17.3	10.4	9.2	11.5
SQLFixAgent (Cen et al., 2025)	32.7	26.3	19.0	25.7
Taxonomy-guided refinement				
ErrorLLM (Hong et al., 2026)	33.6	23.6	17.6	24.0
Taxonomy-guided error grounding				
TEG (ours)	**47.3**	**36.7**	**29.6**	**37.0**

TEG reaches 37.0 overall, compared with 24.0 for ErrorLLM and 26.0 for Self-Debug (Simple), the strongest method among the baselines augmented with error types. TEG remains above every such baseline in each error-count group. These comparisons show that its advantage persists when all methods receive the same error types, while their correction procedures still differ.

C.3.3 Single-error results by error category

Table 15 compares the Qwen2.5-7B methods with and without oracle types by error category. The 110 single-error queries comprise Attribute (37), Value (29), Condition (17), Table (15), and Others (12). Others combines Operator-, Function-, *Clause-Related*, and catch-all Other errors. No *Subquery-Related* errors occur in this single-error subset. Among the six reasoning and SQL-specific baselines, oracle types improve two, reduce three, and leave one unchanged, and they lower Base from 30.0 to 28.2. DIN-SQL with error types leads on Value errors, while MAC-SQL and TEG tie on Others. TEG has the highest overall accuracy, 47.3 against 35.5 for the best baseline given error types.

Table 15: **Oracle types lift only two of six baselines, and TEG stays highest overall.** Single-error execution accuracy by error category on Qwen2.5-7B-Instruct. Each pair shows results without and with the oracle error type. ErrorLLM and TEG require types, so their entries without types are dashes. Bold marks the best value per category with and without oracle error types, including ties. With error types, DIN-SQL leads on Value, and SQLFixAgent and MAC-SQL tie TEG on Table and Others.

Method	Attribute		Value		Condition		Table		Others		Overall	
Base	24.3	18.9	41.4	37.9	29.4	29.4	33.3	46.7	16.7	8.3	30.0	28.2
Reasoning and self-correction												
CoT	35.1	21.6	37.9	34.5	52.9	47.1	26.7	53.3	16.7	16.7	35.5	32.7
Self-Debug (Simple)	24.3	24.3	41.4	48.3	35.3	41.2	53.3	46.7	16.7	16.7	33.6	35.5
Self-Debug (Expl)	24.3	16.2	37.9	31.0	35.3	35.3	33.3	53.3	8.3	16.7	29.1	28.2
SQL-specific refinement												
DIN-SQL	10.8	10.8	51.7	**55.2**	23.5	29.4	13.3	13.3	16.7	16.7	24.5	26.4
MAC-SQL	10.8	8.1	27.6	20.7	23.5	23.5	6.7	20.0	**25.0**	**25.0**	18.2	17.3
SQLFixAgent	24.3	10.8	41.4	41.4	41.2	47.1	40.0	**66.7**	16.7	16.7	32.7	32.7
Taxonomy-guided refinement												
ErrorLLM	–	16.2	–	44.8	–	41.2	–	60.0	–	16.7	–	33.6
Taxonomy-guided error grounding												
TEG (ours)	–	**37.8**	–	51.7	–	**58.8**	–	**66.7**	–	**25.0**	–	**47.3**

D Additional Results on End-to-end Correction

This section extends Section 4.3: how well the predicted error types match the oracle error types, and how single-error correction changes by error category when TEG uses them instead.

D.1 Accuracy of predicted error types

We predict error types with one call to Opus 4.8 (`claude-opus-4-8`) per query, using adaptive thinking and an output limit of 8,192 tokens. The prompt provides the question, issued SQL, and schema DDL, together with short definitions of all 31 taxonomy sub-types. It requests a JSON response containing a correctness decision and a list of error category and sub-type pairs; the reference SQL and oracle annotations are not supplied.

We parse the predicted labels, normalize category and sub-type names to the taxonomy, and discard unrecognized pairs. The resulting predictions are saved and shared across all correctors in the predicted-type comparisons.

Table 16 evaluates the error types predicted by Opus 4.8 against the oracle error types on all 549 queries. Exact match falls from 52.7 on single-error queries to 8.8 on two-error queries and 3.5 on three or more errors. Ignoring sub-types raises overall exact match from 16.2 to 20.0. Opus 4.8 predicts 726 errors against 1,160 oracle annotations, under-counting on 58.7% of queries and over-counting on 6.2%. It predicts no error on 16.6% of all queries. On the evaluated single-error subset, empty predictions trigger TEG's unmasked fallback. For nonempty predictions, missed error types can omit needed correction steps. These counts describe prediction quality on all 549 queries; predicted-type correction itself is evaluated only on the single-error subset.

Table 16: **Predicted types match about half of single-error annotations, few multi-error ones.** Accuracy of the error types predicted by Opus 4.8 on all 549 queries, by the oracle number of errors (110 / 297 / 142). Exact match requires the predicted annotation to equal the oracle one as a multiset of (category, sub-type) pairs; the second row ignores the sub-type. Ignoring sub-types raises overall exact match only from 16.2 to 20.0, and the predictions under-count errors on 58.7% of queries.

	1 error	2 errors	3+ errors	Overall
Exact match	52.7	8.8	3.5	16.2
Exact match, category only	55.5	14.8	3.5	20.0

D.2 Single-error results by error category with predicted types

Table 17 breaks down the results with oracle and predicted types by category. The Opus 4.8 block uses only the types it predicts itself. For TEG, predicted types reduce Attribute accuracy by 13.5 points at 14B and 13.6 at 32B, and reduce Condition accuracy at all three Qwen2.5 scales. Table and Others results vary in direction across scales. Their small sample sizes (15 and 12 queries) limit interpretation of these reversals, and the oracle row is not a strict upper bound for every category.

Table 17: **Predicted error types lower TEG's Condition accuracy at every Qwen2.5 scale.** Single-error execution accuracy by the oracle error category. Each model row is Base, a single unmasked correction call, and the indented rows add TEG with predicted or oracle types. The Opus 4.8 block uses only the types it predicts itself, and Qwen3-8B has separate thinking OFF/ON blocks. Bold marks the best overall accuracy within each model block. Attribute accuracy also drops at 14B and 32B, whereas Table and Others change in both directions on only 15 and 12 queries.

Method	Attribute	Value	Condition	Table	Others	Overall
Qwen2.5-7B	24.3	41.4	29.4	33.3	16.7	30.0
TEG (predicted types)	37.8	55.2	52.9	53.3	25.0	45.5
TEG (oracle types)	37.8	51.7	58.8	66.7	25.0	**47.3**
Qwen2.5-14B	27.0	37.9	41.2	40.0	41.7	35.5
TEG (predicted types)	35.1	58.6	47.1	80.0	25.0	48.2
TEG (oracle types)	48.6	55.2	58.8	66.7	33.3	**52.7**
Qwen2.5-32B	32.4	48.3	35.3	66.7	33.3	41.8
TEG (predicted types)	43.2	58.6	47.1	86.7	33.3	52.7
TEG (oracle types)	56.8	55.2	52.9	93.3	25.0	**57.3**
Qwen3-8B (Thinking OFF)	27.0	34.5	29.4	46.7	25.0	31.8
TEG (predicted types)	37.8	51.7	35.3	80.0	25.0	45.5
TEG (oracle types)	43.2	48.3	41.2	80.0	16.7	**46.4**
Qwen3-8B (Thinking ON)	37.8	44.8	35.3	73.3	33.3	43.6
TEG (predicted types)	54.1	51.7	52.9	73.3	33.3	53.6
TEG (oracle types)	62.2	48.3	58.8	73.3	41.7	**57.3**
Opus 4.8	51.4	69.0	58.8	86.7	66.7	63.6
TEG (self-predicted types)	54.1	82.8	52.9	93.3	75.0	**69.1**

E Extended Discussion

This section expands the discussion in Section 5: what the results say about using a diagnosis, where the idea of error grounding comes from, and what carrying it beyond SQL would require.

E.1 Using error diagnoses to construct correction inputs

The comparisons with baselines given the same error types and the masking ablation support using diagnoses to construct correction inputs: TEG retains its lead, and the w/o masking variant lowers accuracy while retaining the types. These comparisons evaluate the full pipelines and editing configurations rather than isolating masking alone. The 7B gains suggest potential for locally hosted correction in organizations that keep sensitive database context on premises, provided error annotations can also be supplied locally. The size of the benefit depends on the LLM and on whether the types are oracle or predicted. The near tie with ErrorLLM under oracle types on Opus 4.8 suggests that a strong LLM also makes good use of the type-specific context that ErrorLLM extracts.

E.2 Relation to visual grounding and inpainting

Visual grounding links linguistic descriptions to image regions (Yu et al., 2018b), offering an analogy for connecting an error diagnosis to code structures. TEG grounds types in construct classes and edit instructions, so its targets can include correct occurrences rather than an exact faulty location. Its masking is inspired by image inpainting (Yu et al., 2018a) and mask-guided diffusion editing (Avrahami et al., 2022), where selected content is regenerated from its surroundings. TEG adopts this regeneration with an LLM that generates complete queries, without diffusion or a fixed edit boundary, so the LLM may also change unmasked constructs.

E.3 Carrying error grounding beyond SQL

A transfer needs recurring error categories, identifiable editable elements, and instructions that preserve the surrounding constraints. An AST can expose these elements in code, while a schema can do so for configuration data. Candidate selection could use tests, type checks, or schema validation where available, although these checks would not by themselves establish semantic correctness. Sequential correction would also need to account for dependencies between edits and the possibility that one correction invalidates another. Each domain would keep the mapping from error types to editable structures and requested operations. In general-purpose code, an API-misuse category could select call expressions and their arguments for replacement. In configuration files, an incorrect-value category could select fields for replacement, while a missing-key category has no existing construct to select and would request insertion. These applications are prospective; each domain would need its own masking rules and correction order. The results here establish evidence in SQL, leaving the effectiveness and cost of these extensions to future evaluation.