

BASS: Budget-Aware Structured Search for Text-to-SQL Generation

Bin Zheng, Zheng Chen*, Kefan Chen, Lulu Cui, Hao Tang, Zijun Xu, Chen anlong

University of Electronic Science and Technology of China

Chengdu, China

{bin.zheng@std.uestc.edu.cn, zchen@uestc.edu.cn}

Abstract

Text-to-SQL generation remains challenging for small and medium-sized language models, which must produce queries that are structurally valid, schema-grounded, and semantically faithful under limited inference budgets. Naive test-time scaling strategies, such as repeated sampling or unconstrained search, often spend budget on invalid, redundant, or semantically weak candidates. We present BASS, a Budget-Aware Structured Search framework for Text-to-SQL generation. BASS searches over deterministic SQL semantic units produced by rule-based Structural Boundary Detection, with lightweight admissibility filtering to reject malformed units. It scores completed candidates using an execution-gated rubric verifier: non-executable SQL receives zero reward, while executable SQL is assigned a continuous semantic score based on schema grounding, intent matching, predicate correctness, aggregation and ordering, and join/compositional structure. To improve budget usage, BASS introduces stagnation-aware branch suppression and exhaustion-aware early stopping, reallocating computation away from saturated search regions. Experiments on Spider and BIRD show strong zero-shot performance across Qwen2.5-Coder backbones. Controlled analyses further show that BASS uses fixed maximum rollout budgets adaptively, achieves better quality–cost behavior than naive repeated sampling, and benefits from semantic verification and stagnation-aware budget control.

1 Introduction

Text-to-SQL maps natural-language questions to executable SQL queries over relational databases. Despite progress on benchmarks such as Spider (Yu et al., 2018) and BIRD (Li et al., 2023), the task remains challenging because correct outputs must be structurally valid, schema-grounded, and seman-

tically faithful to the input question. These requirements are especially difficult for small and medium-sized language models, which often have limited reasoning capacity (Mitra et al., 2023). Reliable self-correction can additionally require strong verification signals (Zhang et al., 2024).

A natural way to improve generation quality is to allocate more computation at inference time. However, in Text-to-SQL, more test-time computation does not automatically yield better SQL. Repeated sampling may spend budget on near-duplicate or invalid candidates, while unconstrained token-level search can accumulate syntactic or schema-level errors. Grammar-aware decoding improves syntactic validity (Scholak et al., 2021), and recent Text-to-SQL systems explore MCTS-style search, reward-guided inference, and self-rewarded refinement (Yuan et al., 2026; Li et al., 2025a). Yet these approaches still require effective signals for deciding which structural revisions deserve further expansion.

We argue that effective inference-time scaling for Text-to-SQL requires three ingredients: a structured action space, a task-grounded verification signal, and adaptive budget allocation. Search should operate over meaningful SQL fragments rather than raw token continuations; verification should distinguish merely executable SQL from semantically correct SQL; and computation should be redirected away from branches whose verified rewards have saturated.

These requirements motivate **BASS**, a **B**udget-**A**ware **S**tructured **S**earch framework for Text-to-SQL generation. BASS searches over SQL semantic units produced by rule-based Structural Boundary Detection, uses an execution-gated rubric verifier to assign continuous semantic rewards to executable candidates, and applies stagnation-aware branch suppression with exhaustion-aware early stopping. Together, these mechanisms direct test-time computation toward executable, schema-

*Corresponding author.

grounded, and semantically promising SQL revisions.

We evaluate BASS on Spider and BIRD across Qwen2.5-Coder backbones in the zero-shot setting. Since prior systems differ in backbone, prompting strategy, verifier usage, rollout budgets, and cost accounting, we treat comparisons with previously reported systems as external performance positioning rather than fully matched-compute evaluation. We further analyze BASS through full BIRD Dev ablations, controlled quality–cost comparison, hyperparameter sensitivity, and verifier candidate analysis.

Our contributions are as follows:

- We propose **BASS**, a budget-aware structured search framework that performs Text-to-SQL inference over deterministic SQL semantic units rather than raw token continuations.
- We introduce an **execution-gated rubric verifier** that filters non-executable SQL and assigns executable candidates continuous semantic rewards.
- We design **stagnation-aware budget control** to suppress saturated branches and terminate search when the expanded frontier is exhausted.
- We evaluate BASS on **Spider** and **BIRD** across Qwen2.5-Coder backbones, with controlled analyses of component effects, quality–cost behavior, hyperparameter sensitivity, and verifier behavior.

2 Related Work

Our work is related to LLM-based Text-to-SQL, structured SQL generation, inference-time search and scaling, and verifier-guided reasoning. We situate BASS as a budget-aware Text-to-SQL method whose action space, verification signal, and budget controller are specialized for SQL generation.

LLM-based Text-to-SQL. Text-to-SQL has evolved from classical semantic parsing into a major benchmark for LLM-based structured generation. Spider evaluates cross-domain generalization over unseen schemas (Yu et al., 2018), while BIRD introduces larger and more realistic database-grounded queries (Li et al., 2023). Recent systems improve performance through prompting and context construction (Dong et al., 2023; Gao et al., 2024), schema linking (Cao et al., 2024), and

structure-aware generation (Zhang et al., 2025b). Other methods explore self-correction, search, or test-time strategies for SQL generation, including MCTS-based search (Yuan et al., 2026; Li et al., 2025a; Liu et al., 2026), verifiable or reward-guided inference (Zhu et al., 2026; Tritto et al., 2026; Lyu et al., 2025), and agentic or multi-path execution-feedback frameworks (Pham et al., 2026; Hao et al., 2026; Wang et al., 2025b). BASS belongs to this broader direction, but focuses on using limited inference-time computation more effectively through semantic-unit search, execution-gated verification, and adaptive budget allocation.

Structured and grammar-aware SQL generation. A closely related line improves Text-to-SQL by constraining or structuring the generation process. Grammar-aware decoding methods such as PICARD restrict decoder outputs to syntactically admissible continuations (Scholak et al., 2021). More broadly, structured SQL generation uses sketches, ASTs, clause-level decomposition, or template-like intermediate representations. These approaches primarily address *what can be generated* by imposing structural constraints or decomposing SQL into more manageable forms. BASS is complementary but distinct: it studies *where limited test-time computation should be spent* among alternative structural revisions. Instead of token-level grammar-constrained decoding, BASS uses rule-based Structural Boundary Detection to define a semantic-unit action space, filters malformed units with lightweight admissibility checks, and uses verifier rewards to guide search.

Inference-time search and scaling. A growing literature studies how inference-time computation improves reasoning through exploration, revision, planning, and multi-trajectory generation. Chain-of-thought and tree-of-thought style methods show that intermediate reasoning paths can improve problem solving (Wei et al., 2022; Yao et al., 2023). Recent inference-scaling studies further analyze how to allocate test-time compute effectively (Snell et al., 2025; Wu et al., 2025). Other work explores stronger reasoning trajectories, adaptive branching, dynamic parallel tree search, and meta-reasoning strategies (Huang et al., 2024; Inoue et al., 2025; Ding et al., 2025; Xiang et al., 2025).

Many of these methods adopt planning or tree-search views of inference. MCTS provides a classical foundation for balancing exploration and exploitation (Kocsis and Szepesvári, 2006; Browne et al., 2012), while later work connects tree search

to language-agent reasoning, planning, and action selection (Hao et al., 2023; Ding et al., 2024; Zhou et al., 2024). Related systems also use search and planning for autonomous agents or game-like decision processes (Silver et al., 2017; Putta et al., 2024; Zhang et al., 2025c). However, Text-to-SQL imposes constraints that are less prominent in open-ended reasoning: syntactic validity, execution validity, schema grounding, and final semantic correctness must all hold simultaneously. BASS therefore does not treat test-time scaling as blind expansion. It converts fixed maximum rollout budgets into structured SQL search, execution-gated semantic verification, and stagnation-aware budget control.

Verifier-guided reasoning. Verifier-guided reasoning uses evaluators for search, reranking, or iterative refinement. Recent work studies process reward models and verifier-style supervision for improving reasoning quality (Li and Li, 2025; Yang et al., 2025b). Other approaches develop hierarchical or label-free reward signals for multi-step reasoning (Wang et al., 2026; Yuan et al., 2025). In code generation and program synthesis, execution-grounded feedback can filter invalid programs and improve search reliability (Chen et al., 2019; Gehring et al., 2025). Execution-enhanced process reward models further show that program feedback can provide useful training or selection signals (Li et al., 2025b).

BASS follows this idea but tailors verification to Text-to-SQL. Its verifier first applies execution checking and only invokes semantic auditing on executable candidates. This is related to recent work on generative verifiers, executable verifiers, and tool-augmented reasoning (Zhang et al., 2025a; Pezeshkpour and Hruschka, 2026; Rainone et al., 2025), as well as studies showing that small models often require strong verifiers for reliable self-correction (Zhang et al., 2024). Unlike binary executability filters, the semantic audit assigns continuous scores for schema grounding, intent matching, predicate correctness, aggregation and ordering, and join/compositional structure.

3 Method

3.1 Overview and Budgeted Objective

Given a natural-language question Q , optional evidence E , and a raw database context D_{raw} , BASS first constructs a compact query-adapted schema context C through schema pruning. The pruning module combines dense retrieval, lexical backup,

graph-based table routing, and topology-aware key preservation to reduce schema noise while retaining relevant tables, columns, primary keys, and foreign keys. Schema pruning is used only for context construction; the core BASS procedure operates on C through semantic-unit search, execution-gated verification, and stagnation-aware budget control. Details and pruning effectiveness are provided in Appendix A.

On Spider and BIRD Dev, pruning retains every table and column used by the reference SQL for 98.74% and 97.46% of questions, respectively, while removing 61.19% and 59.92% of schema elements. A missed bridge table or predicate column can nevertheless make the correct join or filter unreachable; we analyze this failure mode in Appendix A.

We study Text-to-SQL as a budgeted structured generation problem over relational databases, following the cross-domain settings of Spider and BIRD (Yu et al., 2018; Li et al., 2023). Given Q and C , the goal is to produce a SQL query y that is executable and semantically faithful to the user intent. Let $\mathcal{V}(C)$ denote the candidate SQL space under context C . BASS seeks

$$\begin{aligned} y^* \in \arg \max_{y \in \mathcal{V}(C)} R(y; Q, C), \\ \text{s.t. } N_{\text{roll}}(Q) \leq B, \end{aligned} \quad (1)$$

where $R(y; Q, C)$ is a verifier-defined terminal utility, B is the maximum rollout budget, and $N_{\text{roll}}(Q)$ is the realized number of executed rollouts. Since BASS may stop early when the search frontier is exhausted, B is an upper bound rather than a guaranteed realized cost.

BASS consists of three coupled components: (i) semantic-unit search over deterministic SQL fragments, (ii) execution-gated semantic verification of completed candidates, and (iii) stagnation-aware budget control. A warm-start SQL is generated by the same backbone used for search under the same zero-shot context; no gold SQL or stronger teacher model is used, and its generation cost is included in our cost accounting.

3.2 Semantic-Unit Search

Token-level search is often unstable for Text-to-SQL because a locally plausible token continuation can still produce a globally invalid or semantically incorrect query. Prior work has addressed SQL structure through intermediate representations and structure-aware generation (Guo et al., 2019;

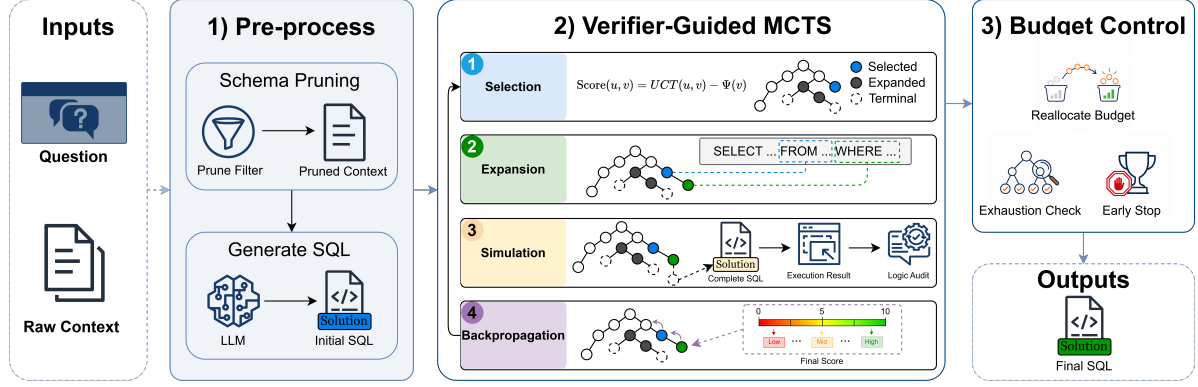


Figure 1: **Overview of BASS.** Given a question Q , optional evidence E , and raw database context $\mathcal{D}_{\text{raw}}$, BASS first constructs a compact query-adapted schema context $\mathcal{C}$ through schema pruning and generates a warm-start SQL candidate. It then performs verifier-guided MCTS over SQL semantic units: **(1) Selection** uses a stagnation-damped UCT score, $\text{Score}(u, v) = \text{UCT}(u, v) - \Psi(v)$; **(2) Expansion** proposes locally closed semantic units using deterministic Structural Boundary Detection; **(3) Verification** applies execution-gated semantic scoring, where non-executable candidates receive zero reward and executable candidates are evaluated by a rubric-based semantic audit; and **(4) Backpropagation** updates tree statistics. Finally, the budget controller monitors reward stagnation, reallocates search away from saturated branches, and triggers early stopping when the expanded frontier becomes exhausted.

Zhang et al., 2025b). Grammar-constrained decoding methods such as PICARD instead restrict token-level continuations to syntactically admissible SQL prefixes (Scholak et al., 2021). BASS takes a complementary search-time view: instead of expanding raw tokens, it searches over **semantic units**, i.e., locally closed SQL fragments aligned with clause-level structure such as projections, joins, predicates, grouping clauses, ordering clauses, nested subqueries, and set-operation segments.

A search state is represented as an ordered sequence of semantic units,

$$s_t = (a_0, a_1, \dots, a_{t-1}), \quad s_{t+1} = s_t \oplus a_t. \quad (2)$$

The action space is instantiated by Structural Boundary Detection (SBD), a deterministic rule-based segmentation procedure. Given a generated SQL candidate or continuation, SBD tracks clause keywords, parenthesis balance, subquery scope, set-operation markers, and incomplete-clause patterns to segment the SQL into candidate semantic units. For example, `SELECT name FROM singer WHERE age > 30 ORDER BY age DESC LIMIT 1` is segmented into `[SELECT name]`, `[FROM singer]`, `[WHERE age > 30]`, and `[ORDER BY age DESC LIMIT 1]`. When parser support is available, BASS uses parser-assisted normalization; otherwise, it falls back to deterministic rule-based segmentation.

The evaluated rules target the SQLite syntax used by Spider and BIRD. Parser/fallback disagreement, unsupported dialect constructs, or overly

coarse or fine boundaries can reject a valid branch or reduce revision flexibility. Porting BASS to another SQL dialect may therefore require changes to parser configuration, boundary and admissibility rules, executor support, and the verifier rubric. Full boundary rules and edge-case handling are provided in Appendix B.

BASS realizes inference-time optimization through verifier-guided Monte Carlo Tree Search (MCTS) (Kocsis and Szepesvári, 2006; Browne et al., 2012). Each rollout consists of selection, expansion, verification, and backpropagation. At a non-terminal node u , BASS selects a child v using a UCT-style score:

$$\text{UCT}(u, v) = \frac{Q(v)}{N(v)} + C_{\text{exp}} \sqrt{\frac{\log(1 + N(u))}{1 + N(v)}}, \quad (3)$$

where $Q(v)$ is the accumulated terminal reward, $N(v)$ is the visit count, and C_{exp} controls exploration. We use UCT because the deployed generator interface does not expose token-level log-probabilities. Therefore, BASS does not use a PUCT proposal prior. BASS also does not use PICARD or token-level grammar-constrained decoding; malformed semantic units are filtered post hoc by lightweight admissibility checks and terminal execution validation.

3.3 Execution-Gated Verification

Execution feedback is a natural signal for program generation (Chen et al., 2019). However, executability alone is insufficient: many SQL queries execute successfully while missing required predicates, joins, aggregation, ordering constraints, or question intent. Recent Text-to-SQL search methods also use execution or reward feedback to improve candidate selection (Yuan et al., 2026; Li et al., 2025a). BASS therefore uses an *execution-gated rubric verifier*. Given a terminal SQL candidate y , the verifier first applies an execution check:

$$W_{\text{exe}}(y) \in \{0, 1\}. \quad (4)$$

Non-executable candidates receive zero reward and bypass semantic auditing.

If $W_{\text{exe}}(y) = 1$, BASS applies a deterministic rubric-based semantic audit. The audit scores five dimensions: schema grounding, intent matching, condition correctness, aggregation/sorting, and join/compositionality. Each dimension receives a score $s_j \in \{0, 1, 2\}$, and the normalized semantic score is

$$\text{Sim}_{\text{sem}}(y; Q, C) = \frac{1}{10} \sum_{j=1}^5 s_j. \quad (5)$$

The terminal reward is

$$R(y; Q, C) = W_{\text{exe}}(y) \cdot \text{Sim}_{\text{sem}}(y; Q, C). \quad (6)$$

This multiplicative form treats executability as a hard prerequisite while using the semantic audit to rank executable candidates by their alignment with the question and schema.

All semantic-audit calls use deterministic decoding with temperature 0.0. The verifier returns structured JSON with dimension-level scores and a normalized semantic score. If the output cannot be parsed, the candidate receives $\text{Sim}_{\text{sem}} = 0$ and is marked as a verifier parse failure. The verifier prompt explicitly instructs the model not to penalize stylistically different SQL formulations when they are semantically equivalent under the given question and schema. The full prompt, output format, reward implementation, model settings, and error-handling rules are provided in Appendix C.

3.4 Stagnation-Aware Budget Control

Inference-time search can waste computation when additional rollouts repeatedly revisit low-yield branches or produce near-duplicate candidates.

This issue is especially important for test-time scaling methods, where improved performance depends on directing computation to productive search regions rather than merely increasing it. Tree-of-thought and language-agent tree search emphasize the importance of structured exploration (Yao et al., 2023; Zhou et al., 2024), while adaptive branching studies show that inference-time compute should be allocated selectively (Inoue et al., 2025).

For each node v , BASS maintains a stagnation counter $\sigma(v)$, defined as the number of consecutive rollouts during which the best terminal reward in the subtree rooted at v fails to improve. The selection score is

$$\text{Score}(u, v) = \text{UCT}(u, v) - \Psi(v), \quad (7)$$

where $\Psi(v)$ is a bounded stagnation penalty. In implementation, it is computed as a capped function of the stagnation counter $\sigma(v)$ and a recent reward-improvement indicator. It increases when the best verified reward in the subtree rooted at v has not improved for several consecutive rollouts, and is capped by C_{max} to avoid permanently suppressing a branch due to short-term verifier noise. Intuitively, branches whose verified rewards have saturated become less likely to be selected, while branches with continuing reward improvement remain competitive. The exact penalty form and hyperparameters are provided in Appendix D.

BASS also applies instance-level early stopping. Let $\mathcal{F}_{\text{exp}} = \mathcal{L}_{\text{expanded}}(\mathcal{T})$ denote the expanded frontier. A node is marked exhausted when its stagnation counter exceeds a stricter threshold, and search terminates when all expanded frontier nodes are exhausted:

$$\text{StopCond}(\mathcal{T}) = \mathbb{I} \left[\begin{array}{c} \forall v \in \mathcal{F}_{\text{exp}}, \\ \text{Exhausted}(v) \end{array} \right]. \quad (8)$$

Thus, the realized inference cost is determined by the executed rollouts rather than the nominal budget B . Easy or already-saturated instances may stop early, while harder instances continue to consume budget as long as verifier feedback remains informative. Detailed hyperparameters, run artifacts, and cost logging fields are reported in Appendix D; additional verifier analysis and error taxonomy are provided in Appendix E.

Method	Backbone	Zero-shot	Spider		BIRD	
			Dev-EX	Test-EX	Dev-EX	Dev-VES
<i>Proprietary & Frontier Models</i>						
Standard Prompting	GPT-4	✓	72.9	–	46.4	49.8
DIN-SQL (Pourreza and Rafiei, 2023)	GPT-4	✓	82.8	85.3	50.7	58.8
DAIL-SQL (Gao et al., 2024)	GPT-4	✓	83.5	86.6	54.8	56.1
MAC-SQL (Wang et al., 2025a)	GPT-4	✓	86.8	82.8	59.4	66.2
MCS-SQL (Lee et al., 2025)	GPT-4	×	89.5	89.6	63.4	64.8
CHASE-SQL (Pourreza et al., 2025)	Gemini 1.5 Pro	×	–	87.6	73.0	–
<i>Open-Source Baselines</i>						
SFT (Qwen Team, 2024)	Qwen2.5-7B	×	80.9	82.8	51.4	53.1
SQL-o1 (Lyu et al., 2025)	Qwen2.5-7B	×	84.7	85.1	66.7	70.4
<i>Search-enhanced Baselines</i>						
MCTS-SQL (Yuan et al., 2026)	Qwen2.5-Coder-1.5B	×	67.5	71.7	40.7	44.9
MCTS-SQL (Yuan et al., 2026)	Qwen2.5-Coder-3B	×	74.0	74.0	46.7	48.2
MCTS-SQL (Yuan et al., 2026)	Qwen2.5-Coder-7B	×	–	–	53.6	52.2
Alpha-SQL (Li et al., 2025a)	Qwen2.5-Coder-7B	✓	84.0	–	66.8	–
Alpha-SQL (Li et al., 2025a)	Qwen2.5-Coder-14B	✓	87.0	–	68.7	–
BASS (Ours)	Qwen2.5-Coder-1.5B	✓	75.0±0.4	78.8±0.2	52.2±0.4	56.4±0.3
BASS (Ours)	Qwen2.5-Coder-3B	✓	82.1±0.5	83.3±0.2	59.8±0.3	60.7±0.3
BASS (Ours)	Qwen2.5-Coder-7B	✓	85.9±0.2	87.1±0.3	68.1±0.2	69.3±0.2
BASS (Ours)	Qwen2.5-Coder-14B	✓	89.7±0.1	89.5±0.2	69.5±0.1	70.7±0.2

Table 1: Main results on Spider and BIRD. Methods are grouped into proprietary/frontier models, open-source baselines, and search-enhanced frameworks. We report execution accuracy (EX) on Spider Dev/Test and BIRD Dev, and Valid Efficiency Score (VES) on BIRD Dev. ✓ denotes zero-shot evaluation, and “–” indicates numbers not reported in prior work. Published baseline results are used for external performance positioning rather than fully matched-compute comparison. Shaded rows correspond to BASS; our results are reported as mean±std over 5 runs with per-instance maximum budget $B = 50$.

4 Experiments

4.1 Setup

We evaluate BASS on Spider and BIRD. Spider evaluates cross-domain SQL generalization over unseen database schemas (Yu et al., 2018), while BIRD contains larger schemas and more realistic query complexity (Li et al., 2023). We report execution accuracy (EX) on Spider Dev/Test and BIRD Dev, and Valid Efficiency Score (VES) on BIRD Dev.

We instantiate BASS with Qwen2.5-Coder backbones at 1.5B, 3B, 7B, and 14B scales (Hui et al., 2024) in the zero-shot setting, where no few-shot exemplars or in-context demonstration pairs are included. Unless otherwise specified, BASS uses a maximum rollout budget of $B = 50$ per instance, while the realized cost may be lower due to early stopping.

BASS uses UCT selection, execution-gated semantic reward, best-reward final selection, rule-based SBD with admissibility filtering, and stagnation-aware budget control. The verifier uses deterministic decoding with temperature 0.0, and we do not use PUCT because the generator inter-

face does not expose token-level log-probabilities. Main results and ablations use full evaluation splits; expensive controlled analyses use the BIRD Subsampled Development Set (SDS) following CHESS (Talaie et al., 2024). SDS is used only for diagnostic controlled analyses, not as a replacement for full BIRD Dev evaluation. Its construction, difficulty distribution, and usage scope are provided in Appendix D.2. Additional setup details, including generator decoding, computational setup, evaluator settings, and run artifacts, are provided in Appendices C and D.

We compare against proprietary/frontier models, open-source baselines, and prior search-enhanced Text-to-SQL systems. For published systems, we report numbers from the corresponding papers and treat them as external performance positioning rather than fully matched-compute comparisons.

4.2 Main Results

Table 1 summarizes the zero-shot performance of BASS on Spider and BIRD and positions it against representative reported systems. Overall, BASS achieves strong performance across Qwen2.5-Coder backbones, with particularly clear

gains for small and medium-sized models. These results suggest that budget-aware structured search can improve Text-to-SQL reliability without modifying the underlying backbone parameters.

Compared with MCTS-SQL using the same Qwen2.5-Coder backbone family, BASS consistently improves performance on both Spider and BIRD at the 1.5B and 3B scales. This indicates that semantic-unit search, execution-gated verification, and stagnation-aware budget control provide useful structure and reward signals for models with limited self-correction capacity.

At larger backbone scales, BASS remains competitive with representative search-enhanced systems. Since prior systems differ in backbone access, prompting strategy, verifier usage, rollout budgets, and cost accounting, we interpret these comparisons as external performance positioning rather than fully matched-compute evaluation. Controlled quality–cost behavior, component ablations, hyperparameter sensitivity, and verifier behavior are analyzed in Section 5 and Appendix D.

5 Analysis

We analyze how BASS uses additional test-time computation and which design choices contribute to its gains. Rather than treating search as black-box inference-time scaling, we examine three questions: whether additional computation is used more effectively than naive repeated sampling, which components are responsible for the final accuracy, and how sensitive BASS is to key search and budget-control hyperparameters.

5.1 Quality–Cost Behavior

BASS treats the rollout budget B as a maximum budget rather than a fixed realized cost. Under the default setting $B = 50$, BASS executes 27.92 rollouts per instance on average, substantially below the maximum budget. This shows that BASS does not uniformly consume all available rollouts: easy or saturated instances can stop early, while harder instances continue to expand as long as verified rewards remain informative.

To quantify the trade-off, we conduct a controlled internal comparison on BIRD-SDS with Qwen2.5-Coder-7B. Figure 2 compares Direct decoding, Best-of- N , and Full BASS under the same schema-pruning, evaluation, and logging protocol. Direct decoding is cheapest but has the lowest execution accuracy. Best-of- $N@10$ reaches 48.98

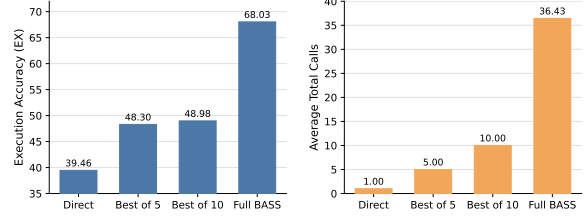


Figure 2: Controlled internal quality–cost comparison on BIRD-SDS with Qwen2.5-Coder-7B. BASS reaches a higher-accuracy operating point at higher token and call cost than Direct decoding and Best-of- N ; the points are not compute matched.

Additional rollouts	0	5	10	20
EX (%)	48.30	55.10	60.54	63.27
Additional rollouts	30	40	50	
EX (%)	67.35	67.35	68.71	

Table 2: Fixed-horizon BASS on BIRD-SDS. “0” denotes the five-proposal initialization pool with no additional rollout; all initialization cost is included.

EX with 17,387 tokens and 10.00 LLM calls per instance. Full BASS reaches 68.03 EX with 77,372 tokens, 36.43 total LLM calls, and 12.03 execution checks. BASS therefore occupies a higher-cost, higher-accuracy operating point; this comparison does not claim compute-matched dominance.

We also disable instance-level early stopping and evaluate fixed rollout horizons while keeping the seed and proposal stream unchanged. Table 2 shows that most gains arrive in the first 30 additional rollouts. Adaptive BASS reaches 68.03 EX with 27.92 realized rollouts on average, compared with 68.71 EX when every instance is forced to 50 rollouts. This reduces realized rollouts by 44.2% for a 0.68-point EX difference.

The practical choice depends on deployment constraints. BASS is most appropriate for offline analytics, high-value queries, and reliability-sensitive workflows where the accuracy gain justifies additional inference cost. Direct decoding or a small candidate set remains preferable for strict-latency or high-throughput interfaces. Detailed cost accounting is provided in Appendix D.5.

5.2 Verifier Reliability

We evaluate the verifier against correctness from the official BIRD evaluator, which is never used as the search reward. On the default BIRD-SDS candidate pools, at least one correct candidate is available for 112 of 147 questions, while reward-based Top-1 selects a correct candidate for 100.

Verifier meta-evaluation	Result
Reward-selected Top-1	68.03% (100/147)
Candidate-pool oracle	76.19% (112/147)
Oracle selection gap	8.16 points (12)
High-score false positives	36.4% (153/420)

Table 3: Verifier meta-evaluation on the default adaptive BIRD-SDS run.

The 47 final errors therefore comprise 35 candidate-generation failures and 12 selection errors, giving an 8.16-point gap to the candidate-pool oracle. Among oracle-solvable questions, the verifier selects a correct candidate in 100/112 cases (89.29%).

The candidate-level results show both enrichment and residual brittleness. In the highest verifier-score bin $[0.8, 1.0]$, 267 of 420 executable candidates are correct and 153 are false positives. High reward is therefore an informative ranking signal, not a calibrated probability or correctness guarantee. Table 3 summarizes the main statistics; the complete score-bin analysis and error taxonomy appear in Appendix E.

5.3 Component Ablations

Table 4 reports ablations on BIRD Dev using Qwen2.5-Coder-7B in the zero-shot setting with $B = 50$. The results show that BASS depends most strongly on execution-gated semantic verification and stagnation-aware budget control.

Removing the semantic audit causes the largest drop, from 68.1 to 57.1. This confirms that executability alone is not a sufficient reward signal for Text-to-SQL search: many candidates can execute successfully while missing key predicates, aggregation behavior, join paths, or question intent. The rubric-based semantic score provides a finer-grained signal for guiding search among executable candidates.

Removing the stagnation penalty also substantially reduces performance, from 68.1 to 60.9. Budget control therefore affects search quality as well as efficiency by discouraging repeated expansion of branches whose best verified rewards no longer improve. Removing warm-start yields a smaller drop, from 68.1 to 67.1, suggesting that initialization stabilizes early search but is not the primary source of BASS’s gains.

5.4 Hyperparameter Sensitivity

We further study how BASS responds to key search and budget-control hyperparameters. Figure 3 re-

Variant	Task Difficulty			Overall (Δ)
	Sm.	Md.	Ch.	
Full BASS	74.1	59.7	55.9	68.1
w/o Semantic Audit	64.6	46.3	41.0	57.1 ($\downarrow 11.0$)
w/o Stagnation Penalty	68.9	49.5	44.8	60.9 ($\downarrow 7.2$)
w/o Warm-start	74.0	56.7	55.2	67.1 ($\downarrow 1.0$)

Table 4: Ablation study on BIRD Dev with Qwen2.5-Coder-7B in the zero-shot setting ($B = 50$). We report execution accuracy (EX, %) across three difficulty levels and overall performance. Sm., Md., and Ch. denote simple, moderate, and challenging questions, respectively.

ports one-at-a-time sensitivity results on BIRD-SDS with Qwen2.5-Coder-7B, where each hyperparameter is varied while the remaining configuration is kept fixed.

The sensitivity results show that moderate search and budget-control settings work best. Very small rollout budgets limit exploration and reduce the chance of finding semantically stronger SQL candidates. Increasing the budget initially improves accuracy, but the gains saturate once additional rollouts increasingly revisit unproductive or already saturated regions. This supports treating B as a maximum budget rather than a fixed cost.

The exploration coefficient C_{exp} also affects the balance between exploiting high-reward branches and exploring alternative semantic revisions. Moderate exploration performs best, while either insufficient exploration or excessive exploration can reduce accuracy. Similarly, the bootstrap proposal count benefits from a moderate value: too few proposals limit early branching diversity, while too many proposals can introduce redundant or weak candidates.

Budget-control parameters show a similar pattern. Stagnation and saturation penalties help suppress branches whose verified rewards no longer improve, but overly aggressive suppression can prematurely downweight useful branches. The thresholds τ_{stag} , τ_{exh} , and $\text{max_unchanged_champion_steps}$ control how quickly BASS detects saturation and terminates search. The results indicate that the default configuration provides a stable trade-off between exploration, verification quality, and realized cost. Detailed hyperparameter values and additional configuration details are provided in Appendix D.6.

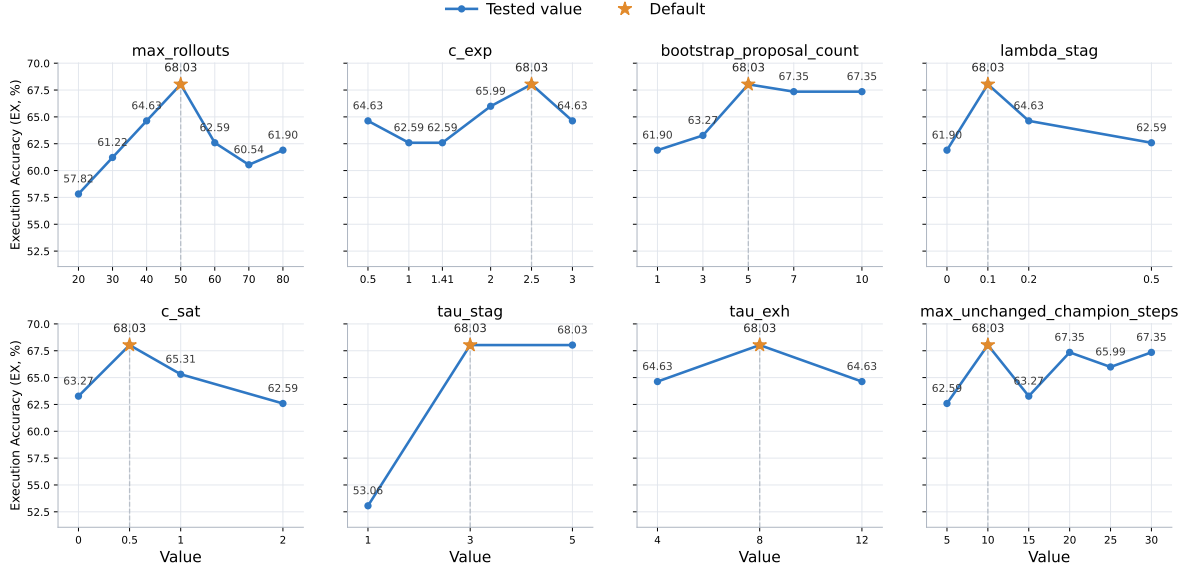


Figure 3: Hyperparameter sensitivity on BIRD-SDS with Qwen2.5-Coder-7B. Each subplot varies one hyperparameter while keeping the remaining configuration fixed. BASS is generally robust around the default configuration, while very small rollout budgets, weak exploration, or overly aggressive budget-control settings can reduce execution accuracy.

6 Conclusion

We presented BASS, a Budget-Aware Structured Search framework for Text-to-SQL generation. BASS improves inference-time search by combining semantic-unit search, execution-gated rubric verification, and stagnation-aware budget control. This design directs test-time computation toward executable, schema-grounded, and semantically promising SQL revisions rather than invalid, redundant, or saturated search regions. Experiments on Spider and BIRD show that BASS achieves strong zero-shot performance across Qwen2.5-Coder backbones, especially for small and medium-sized models. Since external systems differ in backbone, prompting strategy, verifier usage, and cost accounting, we treat reported comparisons as performance positioning and use controlled analyses to isolate the effects of BASS components. These analyses further show that BASS obtains better quality–cost behavior than naive repeated sampling and that its gains are mainly driven by semantic verification and budget control. Additional SDS experiments with diverse backbones also suggest that the benefit of structured search is not limited to a single model family. Overall, our results suggest that effective inference-time scaling for Text-to-SQL depends on allocating computation to structurally meaningful, task-grounded revisions rather than merely increasing its volume.

Limitations

BASS is evaluated only on Text-to-SQL benchmarks using SQLite. Generalization to other structured generation tasks, SQL dialects, and database engines remains unverified. Query-adaptive schema pruning may remove required schema elements, while rule-based Structural Boundary Detection and lightweight admissibility filtering do not provide the formal guarantees of token-level grammar-constrained decoding.

Our implementation uses UCT-style selection because the generator interface does not expose token-level log-probabilities for a PUCT proposal prior. BASS also relies on an LLM-based semantic verifier. Execution gating rejects non-executable SQL, but semantic auditing may over-score plausible near misses, under-score equivalent queries, or be sensitive to prompts and schema presentation. Verifier reliability and calibration therefore remain limitations.

BASS incurs additional inference overhead from repeated generation, execution checks, semantic audits, search bookkeeping, and cost logging. It is most suitable when improved SQL reliability justifies this cost. Complex schema linking, long-range compositional reasoning, and deeply nested queries remain challenging, especially when the candidate pool lacks a sufficiently good query.

Acknowledgments

This work was supported by the Sichuan Provincial Natural Science Foundation Project (No. 2025ZNSFSC0508).

References

- Elie Bakouch, Loubna Ben Allal, Anton Lozhkov, Noumane Tazi, Lewis Tunstall, Carlos Miguel Patiño, Edward Beeching, Aymeric Roucher, Aksel Joonas Reedi, Quentin Gallouédec, Kashif Rasul, Nathan Habib, Clémentine Fourrier, Hynek Kydlicek, Guilherme Penedo, Hugo Larcher, Mathieu Morlon, Vaibhav Srivastav, Joshua Lochner, and 4 others. 2025. SmolLM3: smol, multilingual, long-context reasoner. <https://huggingface.co/blog/smolLM3>.
- Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.
- Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, Wei Chen, and Xiang Bai. 2024. Rsl-sql: Robust schema linking in text-to-sql generation. *Preprint*, arXiv:2411.00073.
- Jianlyu Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2318–2335, Bangkok, Thailand. Association for Computational Linguistics.
- Xinyun Chen, Chang Liu, and Dawn Song. 2019. Execution-guided neural program synthesis. In *International Conference on Learning Representations*.
- Ruomeng Ding, Chaoyun Zhang, Lu Wang, Yong Xu, Minghua Ma, Wei Zhang, Si Qin, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. 2024. Everything of thoughts: Defying the law of penrose triangle for thought generation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 1638–1662, Bangkok, Thailand. Association for Computational Linguistics.
- Yifu Ding, Wentao Jiang, Shunyu Liu, Yongcheng Jing, Jinyang Guo, Yingjie Wang, Jing Zhang, Zengmao Wang, Ziwei Liu, Bo Du, Xianglong Liu, and Dacheng Tao. 2025. Dynamic parallel tree search for efficient LLM reasoning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11233–11252, Vienna, Austria. Association for Computational Linguistics.
- Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Lu Chen, Jinshu Lin, and Dongfang Lou. 2023. C3: Zero-shot text-to-sql with chatgpt. *Preprint*, arXiv:2307.07306.
- Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-sql empowered by large language models: A benchmark evaluation. *Proc. VLDB Endow.*, 17(5):1132–1145.
- Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. 2025. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 19034–19055. PMLR.
- Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-SQL in cross-domain database with intermediate representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535, Florence, Italy. Association for Computational Linguistics.
- Hongbin Hao, Xin Zheng, and Xuhong Yu. 2026. Dynamic-SQL: An adaptive NL2SQL framework with multi-path fusion reasoning and execution feedback correction. *Scientific Reports*, 16(1):17565.
- Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 8154–8173, Singapore. Association for Computational Linguistics.
- Zhen Huang, Haoyang Zou, Xuefeng Li, Yixiu Liu, Yuxiang Zheng, Ethan Chern, Shijie Xia, Yiwei Qin, Weizhe Yuan, and Pengfei Liu. 2024. O1 replication journey – part 2: Surpassing o1-preview through simple distillation, big progress or bitter lesson? *Preprint*, arXiv:2411.16489.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shangaoran Qian, and 5 others. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*.
- Yuichi Inoue, Kou Misaki, Yuki Imajuku, So Kuroki, Taishi Nakamura, and Takuya Akiba. 2025. Wider or deeper? scaling llm inference-time compute with adaptive branching tree search. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pages 35448–35484. Curran Associates, Inc.
- Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg. Springer Berlin Heidelberg.

- Lawrence T. Kou, George Markowsky, and Leonard Berman. 1981. [A fast algorithm for Steiner trees](#). *Acta Informatica*, 15(2):141–145.
- Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2025. [MCS-SQL: Leveraging multiple prompts and multiple-choice selection for text-to-SQL generation](#). In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 337–353, Abu Dhabi, UAE. Association for Computational Linguistics.
- Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025a. [Alpha-SQL: Zero-shot text-to-SQL using monte carlo tree search](#). In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 36810–36830. PMLR.
- Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C.C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. [Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls](#). In *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS ’23*. Curran Associates Inc.
- Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. 2025b. [CodePRM: Execution feedback-enhanced process reward model for code generation](#). In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 8169–8182, Vienna, Austria. Association for Computational Linguistics.
- Wendi Li and Yixuan Li. 2025. [Process reward model with q-value rankings](#). In *International Conference on Learning Representations*, volume 2025, pages 14708–14726.
- Tao Liu, Jiafan Lu, Bohan Yu, Pengcheng Wu, Haixin Liu, Guoyu Xu, Xiangheng Li, Lixiao Li, Jiaming Hou, Shijun Zhao, Xinglin Lyu, Kunli Zhang, Yuxiang Jia, and Hongyin Zan. 2026. [IESR: Efficient MCTS-based modular reasoning for text-to-SQL with large language models](#). *Preprint*, arXiv:2602.05385.
- Shuai Lyu, Haoran Luo, Ripeng Li, Zhonghong Ou, Jiangfeng Sun, Yang Qin, Xiaoran Shang, Meina Song, and Yifan Zhu. 2025. [Sql-o1: A self-reward heuristic dynamic search method for text-to-sql](#). *Preprint*, arXiv:2502.11741.
- Arindam Mitra, Luciano Del Corro, Shweti Mahajan, Andres Codas, Clarisse Simoes, Sahaj Agarwal, Xuxi Chen, Anastasia Razdaibiedina, Erik Jones, Kriti Agarwal, Hamid Palangi, Guoqing Zheng, Corby Rosset, Hamed Khanpour, and Ahmed Awadallah. 2023. [Orca 2: Teaching small language models how to reason](#). *Preprint*, arXiv:2311.11045.
- OpenAI. 2025. [gpt-oss-120b & gpt-oss-20b model card](#). <https://openai.com/index/gpt-oss-model-card/>. Model card.
- Pouya Pezeshkpour and Estevam Hruschka. 2026. [AutoPyVerifier: Learning compact executable verifiers for large language model outputs](#). *Preprint*, arXiv:2604.22937.
- Minh Tam Pham, Trinh Pham, Tong Chen, Hongzhi Yin, Quoc Viet Hung Nguyen, and Thanh Tam Nguyen. 2026. [AV-SQL: Decomposing complex text-to-SQL queries with agentic views](#). *Preprint*, arXiv:2604.07041.
- Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaie, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan Arik. 2025. [Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql](#). In *International Conference on Learning Representations*, volume 2025, pages 60385–60415.
- Mohammadreza Pourreza and Davood Rafiei. 2023. [Din-sql: decomposed in-context learning of text-to-sql with self-correction](#). In *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS ’23*. Curran Associates Inc.
- Pranav Putta, Edmund Mills, Naman Garg, Sumeet Motwani, Chelsea Finn, Divyansh Garg, and Rafael Rafailov. 2024. [Agent q: Advanced reasoning and learning for autonomous ai agents](#). *Preprint*, arXiv:2408.07199.
- Qwen Team. 2024. [Qwen2.5 technical report](#). *arXiv preprint arXiv:2412.15115*.
- Corrado Rainone, Tim Bakker, and Roland Memisevic. 2025. [Replacing thinking with tool usage enables reasoning in small language models](#). In *ICML 2025 Workshop on Computer Use Agents*.
- Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. 1994. [Okapi at TREC-3](#). In *Proceedings of The Third Text REtrieval Conference, TREC 1994, Gaithersburg, Maryland, USA, November 2-4, 1994*, volume 500-225 of *NIST Special Publication*, pages 109–126. National Institute of Standards and Technology (NIST).
- Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. [PICARD: Parsing incrementally for constrained auto-regressive decoding from language models](#). In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.
- David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui,

- Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. [Mastering the game of go without human knowledge](#). *Nature*, 550:354–359.
- Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2025. [Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning](#). In *The Thirteenth International Conference on Learning Representations*.
- Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. [Chess: Contextual harnessing for efficient sql synthesis](#). *CoRR*, abs/2405.16755.
- Mattia Tritto, Giuseppe Farano, Dario Di Palma, Gaetano Rossiello, Dharmashankar Subramanian, Fedelucio Narducci, and Tommaso Di Noia. 2026. [Gradesql: Outcome reward models for intelligent text-to-sql generation from llms](#). *Journal of Intelligent Information Systems*.
- Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025a. [MAC-SQL: A multi-agent collaborative framework for text-to-SQL](#). In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 540–557, Abu Dhabi, UAE. Association for Computational Linguistics.
- Pengfei Wang, Baolin Sun, Xuemei Dong, Yaxun Dai, Hongwei Yuan, Mengdie Chu, Yingqi Gao, Xiang Qi, Peng Zhang, and Ying Yan. 2025b. [Agentar-scale-sql: Advancing text-to-sql through orchestrated test-time scaling](#). *Preprint*, arXiv:2509.24403.
- Teng Wang, Jiang Zhangyi, Zhenqi He, Hailei Gong, Shenyang Tong, Wenhan Yang, Zeyu Li, Yanan Zheng, Zifan He, Zewen Ye, Shengjie Ma, and Jianping Zhang. 2026. [Towards hierarchical multi-step reward models for enhanced reasoning in large language models](#). In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 565–576, San Diego, California, United States. Association for Computational Linguistics.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. [Chain-of-thought prompting elicits reasoning in large language models](#). In *Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS ’22*. Curran Associates Inc.
- Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2025. [Inference scaling laws: An empirical analysis of compute-optimal inference for llm problem-solving](#). In *International Conference on Learning Representations*, volume 2025, pages 55824–55845.
- Violet Xiang, Charlie Snell, Kanishk Gandhi, Alon Albalak, Anikait Singh, Chase Blagden, Duy Phung, Rafael Rafailov, Nathan Lile, Dakota Mahan, Louis Castricato, Jan-Philipp Franken, Nick Haber, and Chelsea Finn. 2025. [Towards system 2 reasoning in llms: Learning how to think with meta chain-of-thought](#). *Preprint*, arXiv:2501.04682.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025a. [Qwen3 technical report](#). *Preprint*, arXiv:2505.09388.
- Cheng Yang, Chufan Shi, Siheng Li, Bo Shui, Yujiu Yang, and Wai Lam. 2025b. [LLM2: Let large language models harness system 2 reasoning](#). In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*, pages 168–177, Albuquerque, New Mexico. Association for Computational Linguistics.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. [Tree of thoughts: deliberate problem solving with large language models](#). In *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS ’23*. Curran Associates Inc.
- Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. [Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task](#). In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921. Association for Computational Linguistics.
- Lifan Yuan, Wendi Li, Huayu Chen, Ganqu Cui, Ning Ding, Kaiyan Zhang, Bowen Zhou, Zhiyuan Liu, and Hao Peng. 2025. [Free process rewards without process labels](#). In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 73511–73525. PMLR.
- Shuozhi Yuan, Liming Chen, Miaomiao Yuan, and Zhao Jin. 2026. [MCTS-SQL: Light-weight LLMs can master the text-to-SQL through monte carlo tree search](#). In *Proceedings of the Fortieth AAAI Conference on Artificial Intelligence*, volume 40, pages 34521–34529.
- Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. 2025a. [Generative verifiers: Reward modeling as next-token prediction](#). In *The Thirteenth International Conference on Learning Representations*.
- Qinggang Zhang, Hao Chen, Junnan Dong, Shengyuan Chen, Feiran Huang, and Xiao Huang. 2025b.

Structure-guided large language models for text-to-SQL generation. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 74671–74691. PMLR.

Yao Zhang, Zijian Ma, Yunpu Ma, Zhen Han, Yu Wu, and Volker Tresp. 2025c. [Webpilot: a versatile and autonomous multi-agent system for web task execution with strategic exploration](#). In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*, volume 39 of AAAI’25/AAAI’25/EAAI’25, pages 23378–23386. AAAI Press.

Yunxiang Zhang, Muhammad Khalifa, Lajanugen Logeswaran, Jaekyeom Kim, Moontae Lee, Honglak Lee, and Lu Wang. 2024. [Small language models need strong verifiers to self-correct reasoning](#). In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 15637–15653, Bangkok, Thailand. Association for Computational Linguistics.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. [Language agent tree search unifies reasoning, acting, and planning in language models](#). In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 62138–62160. PMLR.

Yuxuan Zhu, Tengjun Jin, Yoojin Choi, and Daniel Kang. 2026. [Human-level text-to-SQL via reinforcement learning on verified data, without pipeline engineering](#). *Preprint*, arXiv:2603.20004.

A Schema Pruning

This appendix describes the schema pruning module used to construct the compact query-adapted database context C from the raw database context D_{raw} . The module is executed before BASS search and is used only for context construction. It does not access gold SQL, execution labels, or oracle schema links during inference. The core BASS procedure—semantic-unit search, execution-gated verification, and stagnation-aware budget control—operates on the resulting pruned context C .

A.1 Motivation

Large database schemas introduce both efficiency and grounding challenges for Text-to-SQL models. When the full schema is included in every generation and verification prompt, irrelevant tables and columns increase prompt length and may distract the generator or verifier. This issue is especially important for search-based inference methods, where multiple candidates may be generated, executed,

and verified for each question. Since Spider and BIRD contain cross-domain databases with diverse table structures, foreign-key relations, and domain-specific column names, constructing a compact but sufficiently complete schema context is a necessary preprocessing step (Yu et al., 2018; Li et al., 2023).

Our schema pruning module is designed to reduce schema noise while preserving the information needed to generate valid SQL. Instead of making an aggressive hard decision about the exact gold schema, the module follows a recall-oriented strategy: it first retrieves potentially relevant business columns through both semantic and lexical retrieval, then expands the corresponding tables through graph-based routing, and finally assembles a compact set of columns that preserves business columns, primary keys, and topology-valid foreign keys. This design aims to avoid common pruning failures such as removing required columns, dropping bridge tables, or retaining foreign keys that point outside the pruned schema context.

A.2 Offline Schema Indexing

Before inference, we serialize database schema information into column-level records and store them in a vector database. Each record corresponds to a candidate business column and contains available schema information, including the database identifier, table name, column name, table description, column description, foreign-key metadata, and optional value or evidence hints when available. These records are embedded with BGE-M3 and stored in Chroma for dense retrieval (Chen et al., 2024).

In addition to the vector index, we build a BM25 index over a pure literal schema representation (Robertson et al., 1994). The literal representation contains only table names and column names, without long descriptions or additional semantic evidence. This branch is used as a lexical safety net for exact schema mentions, abbreviations, or surface-form matches that may be under-ranked by dense retrieval or reranking.

A.3 Stage 1: Dual-Path Column Retrieval

Given an input question Q and optional evidence E , the pruning module retrieves potentially relevant business columns through two independent paths: a dense retrieval–reranking path and a BM25 lexical path.

Dense retrieval and cross-encoder reranking.

The first path uses BGE-M3 to retrieve a coarse set of candidate columns from the Chroma vector database. We retrieve the top COARSE_TOP_K=60 column records:

$$\mathcal{C}_{\text{coarse}} = \text{DenseRetrieve}_{60}(Q, E, D_{\text{raw}}). \quad (9)$$

These candidates are then reranked by a BGE-Reranker cross-encoder, which scores each candidate column conditioned on the input question and evidence. Let the reranked columns be

$$(c_1, c_2, \dots, c_m), \quad (10)$$

with reranker scores

$$r_1 \geq r_2 \geq \dots \geq r_m. \quad (11)$$

Instead of using a fixed final K , we apply a relative cliff truncation rule. The intuition is that when the reranker score distribution shows a sharp relative drop, columns before the drop are likely to be substantially more relevant than those after it. For adjacent ranked columns, we compute

$$\delta_i = \frac{r_i - r_{i+1}}{\max(|r_i|, \epsilon)}, \quad (12)$$

where ϵ is a small constant for numerical stability. The final reranked prefix size is selected as

$$K = \min\{i : \delta_i \geq \tau_{\text{cliff}}\}. \quad (13)$$

If no clear cliff is detected, the system uses a configured maximum prefix size. This adaptive truncation keeps more columns for ambiguous questions and fewer columns when the reranker produces a clear relevance boundary. The dense retrieval branch therefore returns

$$\mathcal{C}_{\text{rerank}} = \{c_1, \dots, c_K\}. \quad (14)$$

BM25 lexical retrieval. The second path performs BM25 retrieval over the literal table-column index. This path uses only schema surface forms, i.e., table names and column names, and returns the top 15 columns:

$$\mathcal{C}_{\text{bm25}} = \text{BM25Top15}(Q, E, D_{\text{raw}}). \quad (15)$$

The purpose of this path is to recover exact or near-exact schema-name matches that may not receive high dense-retrieval or reranker scores.

Initial business-column pool. The two retrieval paths are merged to form the initial business-column candidate pool:

$$\text{pred_cols} = \mathcal{C}_{\text{rerank}} \cup \mathcal{C}_{\text{bm25}}. \quad (16)$$

Duplicate columns are removed by database, table, and column identifiers. Importantly, no strict table-level filtering is applied at this stage. Any column retrieved by either branch is preserved as a potential business column.

A.4 Stage 2: Lenient Table Selection

After obtaining pred_cols , the module determines an initial set of core tables. This step is intentionally lenient: the goal is not to identify the exact minimal gold table set, but to avoid pruning away tables that may be needed by the final SQL.

We construct the initial table set from two sources.

Static high-score table backup. First, we collect the top five tables according to the highest reranker scores of their associated columns:

$$\text{TopTables}_5 = \text{Top5TablesByRerankScore}. \quad (17)$$

This provides a high-confidence table prior from the dense retrieval-reranking path.

Lineage-based table preservation. Second, we apply an unconditional lineage rule. For every column in pred_cols , its parent table is added to the selected table set:

$$\text{LineageTables} = \{\text{table}(c) : c \in \text{pred_cols}\}. \quad (18)$$

Thus, as long as a column is retrieved by either the reranker branch or the BM25 branch, its table is retained.

The resulting initial table set is

$$\text{selected_tables} = \text{TopTables}_5 \cup \text{LineageTables}. \quad (19)$$

This design deliberately favors recall over aggressive pruning. Although noisy retrieved columns may introduce extra tables, the subsequent graph-routing and topology-pruning stages ensure that the final context remains structurally meaningful.

A.5 Stage 3: Steiner-Tree Routing over the Schema Graph

The selected tables may be scattered across the database schema. To ensure that the final context

supports valid joins, we model the database schema as an undirected graph:

$$G = (V, E), \quad (20)$$

where each node $v \in V$ is a table and each edge $e \in E$ corresponds to a foreign-key relation between two tables. The selected tables are treated as terminal nodes:

$$T = \text{selected_tables}. \quad (21)$$

We then run Steiner-tree routing over the schema graph to connect these terminal tables through short foreign-key paths (Kou et al., 1981). The resulting tree may introduce intermediate bridge tables that were not directly retrieved in Stage 1 but are necessary to join the selected tables. The expanded table set is defined as

$$\text{expanded_tables} = V(\text{SteinerTree}(G, T)). \quad (22)$$

This graph-routing stage is crucial for Text-to-SQL because correct SQL queries often require bridge tables that are not explicitly mentioned in the question. For example, a question may mention two business entities, while the database connects them through an intermediate relation table. If the selected tables can be connected through the schema graph, Steiner-tree routing automatically adds the required intermediate tables. If the selected tables cannot be connected under the available foreign-key graph, the system falls back to a larger schema context rather than forcing an invalid connected subgraph.

A.6 Stage 4: Assembly and Topology-Aware Pruning

After obtaining `expanded_tables`, the module assembles the final column set `final_cols`. This stage preserves all retrieved business columns, adds primary keys for retained tables, and applies topology-aware pruning to foreign keys.

Business-column preservation. All columns in `pred_cols` are included in the final context:

$$\text{pred_cols} \subseteq \text{final_cols}. \quad (23)$$

Because no strict table-level filtering is applied after retrieval, the business columns selected by either the dense reranking branch or the BM25 branch are passed into the final schema context without loss.

Primary-key preservation. For every table in `expanded_tables`, all primary-key columns are added:

$$\text{PK}(t) \subseteq \text{final_cols}, \quad \forall t \in \text{expanded_tables}. \quad (24)$$

This ensures that every retained table exposes its key columns and can participate in join construction.

Topology-aware foreign-key pruning. Foreign keys are not added blindly. For each foreign-key relation involving a table in `expanded_tables`, we check whether the target table of the relation is also included in `expanded_tables`. If the target table is retained, the foreign key is useful for joining two tables inside the pruned context and is therefore kept. If the target table is outside `expanded_tables`, the foreign key points to a table that is not present in the final context and is removed as a dangling relation.

For compact notation, let $\mathcal{T}_{\text{exp}} = \text{expanded_tables}$. Formally, for a foreign-key relation

$$(t_s, c_s) \rightarrow (t_t, c_t), \quad (25)$$

where t_s is the source table and t_t is the target table, the source foreign-key column c_s is retained if and only if

$$t_s \in \mathcal{T}_{\text{exp}} \quad \wedge \quad t_t \in \mathcal{T}_{\text{exp}}. \quad (26)$$

We denote the retained topology-valid foreign keys as

$$\text{FK}_{\text{topo}} = \left\{ c_s \mid \begin{array}{l} (t_s, c_s) \rightarrow (t_t, c_t), \\ t_s \in \mathcal{T}_{\text{exp}}, t_t \in \mathcal{T}_{\text{exp}} \end{array} \right\}. \quad (27)$$

For compact notation, let $\mathcal{T}_{\text{exp}} = \text{expanded_tables}$. The final column set is therefore

$$\begin{aligned} \text{final_cols} = & \text{pred_cols} \\ & \cup \bigcup_{t \in \mathcal{T}_{\text{exp}}} \text{PK}(t) \\ & \cup \text{FK}_{\text{topo}}. \end{aligned} \quad (28)$$

This topology-aware pruning avoids adding foreign keys that point outside the retained schema. As a result, the final context is compact while preserving business columns and the key structure needed for join connectivity.

Stage	Operation
Retrieval	Retrieve candidate business columns through two independent paths: BGE-M3 dense retrieval followed by BGE-Reranker scoring, and BM25 lexical retrieval over table-column names. Merge both paths into <code>pred_cols</code> .
Lenient table selection	Select high-confidence tables from reranker scores and unconditionally include the parent table of every retrieved column, yielding <code>selected_tables</code> .
Steiner routing	Treat tables and foreign keys as an undirected schema graph, then connect <code>selected_tables</code> with Steiner-tree routing to obtain <code>expanded_tables</code> .
Assembly and topology pruning	Construct <code>final_cols</code> by preserving all business columns, adding primary keys for <code>expanded_tables</code> , and retaining only topology-valid foreign keys whose target tables are also retained.

Table 5: Overview of the query-adaptive schema pruning pipeline.

A.7 Pipeline Summary

Table 5 summarizes the four-stage schema pruning pipeline.

A.8 Context Serialization

The final schema context C is serialized from `expanded_tables` and `final_cols`. The serialization includes retained table names, retained column names, primary-key annotations, topology-valid foreign-key annotations, and available table or column descriptions. Original table and column names are preserved to avoid introducing aliasing errors.

A simplified serialized context has the following form:

```
Database: <db_id>

Table: singer
Columns:
- singer_id [primary key]
- name
- age

Table: concert
Columns:
- concert_id [primary key]
- singer_id [foreign key]
- year

Foreign keys:
concert.singer_id -> singer.singer_id
```

The same serialized context C is provided to warm-start generation, semantic-unit expansion, execution checking, and semantic auditing. This

avoids inconsistencies between the generator and verifier during search.

A.9 Evaluation Metrics for Schema Pruning

We evaluate schema pruning using two metrics: instance-level schema recall and schema reduction. These metrics measure whether pruning preserves all schema elements required by the gold SQL and how much irrelevant schema content is removed.

For each instance i , let G_i^T and G_i^C denote the gold tables and gold columns required by the target SQL. Let P_i^T and P_i^C denote the tables and columns retained after pruning. We define instance-level schema recall as

$$\text{Recall} = \frac{1}{N} \sum_{i=1}^N \mathbb{I} [G_i^T \subseteq P_i^T \wedge G_i^C \subseteq P_i^C]. \quad (29)$$

An instance is counted as successfully recalled only if all gold tables and all gold columns are retained after pruning. If any required table or column is removed, the instance is counted as a pruning failure. Thus, this metric is stricter than element-level recall.

For both Spider Dev and BIRD Dev, gold tables and columns are automatically extracted from the gold SQL queries. We parse each gold SQL query, normalize table aliases to canonical schema names, and collect the referenced tables and columns as the gold schema items. These gold schema items are used only for evaluating schema-pruning recall and are never used during inference.

We also report schema reduction, which measures the proportion of tables and columns removed from the original schema. Let S_i^T and S_i^C denote the tables and columns in the original schema before pruning. We compute

$$\text{Reduction} = \frac{1}{N} \sum_{i=1}^N \left(1 - \frac{|P_i^T| + |P_i^C|}{|S_i^T| + |S_i^C|} \right). \quad (30)$$

A higher reduction indicates that the pruning module removes more original schema elements and produces a more compact context.

A.10 Schema Pruning Effectiveness

Table 6 reports the pruning effectiveness on Spider Dev and BIRD Dev. The pruning module achieves high instance-level schema recall while removing approximately 60% of the original schema elements.

Dataset	Recall	Reduction
Spider Dev	98.74	61.19
BIRD Dev	97.46	59.92

Table 6: Schema pruning effectiveness. Recall is measured at the instance level: all gold tables and columns must be retained. Reduction measures the proportion of original schema elements removed. All numbers are percentages.

Logged field	Description
coarse_top_k	Number of columns retrieved by dense retrieval before reranking.
rerank_k	Number of columns retained after relative cliff truncation.
bm25_top_k	Number of literal BM25 columns retained.
num_pred_cols	Number of merged business-column candidates.
num_selected_tables	Number of tables after lenient table selection.
num_expanded_tables	Number of tables after Steiner-tree routing.
num_final_cols	Number of columns in the final serialized context.
num_pk_added	Number of primary-key columns added for expanded tables.
num_fk_kept	Number of topology-valid foreign-key columns retained.
num_fk_dropped	Number of dangling foreign-key columns removed.
fallback_triggered	Whether the module used a larger context due to low confidence or disconnected routing.

Table 7: Schema-pruning diagnostics logged for reproducibility and analysis.

These results show that the pruning module substantially compresses the schema context while preserving nearly all gold table-column requirements at the instance level. On Spider Dev, the module preserves all gold tables and columns for 98.74% of instances while removing 61.19% of the original tables and columns. On BIRD Dev, it preserves all gold schema elements for 97.46% of instances while removing 59.92% of the original schema elements. This indicates that the pruning module provides a compact and high-recall context for subsequent BASS search.

A.11 Diagnostics

For reproducibility and error analysis, we log pruning diagnostics for each instance. These logs separate schema-construction behavior from BASS search-time cost.

These diagnostics allow us to analyze whether pruning errors arise from retrieval failure, reranker

truncation, missing bridge tables, disconnected schema graphs, or topology-aware foreign-key pruning.

A.12 Failure Modes

Although the pruning module is designed to be recall-oriented and topology-aware, it may still fail in several cases.

Retrieval miss. If both dense retrieval and BM25 retrieval fail to retrieve a required business column, the column may be absent from `pred_cols`. This can prevent the generator from producing a correct candidate even if subsequent search and verification are effective.

Concrete column-level miss. One BIRD Dev question asks for the ruling description of the card *Condemn*. The reference SQL joins cards and rulings through `uuid` and projects `rulings.text`. The pruned context retains the `rulings` table and its `uuid`, `date`, and `id` columns, but omits `text`. This case shows that schema pruning can preserve the relevant table and join path while removing a non-key column required by the correct SQL.

Reranker cliff error. The relative cliff truncation rule may select too few columns when the reranker score distribution contains an early but misleading drop. The BM25 branch and lineage-based table preservation mitigate this risk but do not eliminate it.

Disconnected schema graph. If the selected tables lie in disconnected components of the foreign-key graph, Steiner-tree routing cannot produce a single connected subgraph. In this case, the system falls back to a larger schema context.

Over-retention from lenient table selection. Because every retrieved column pulls in its parent table, noisy retrieved columns may introduce extra tables. This behavior is intentional: the module prioritizes recall at the pruning stage and relies on later generation, execution checking, and semantic verification to select among candidates.

Topology-pruning mismatch. A foreign key may be dropped when its target table is outside `expanded_tables`. This is usually desirable because such a key would be dangling in the final context, but it can be harmful if the target table should have been included and was missed by retrieval or routing.

Overall, schema pruning is designed as a conservative context-construction module rather than a definitive schema-linking oracle. It reduces schema noise, preserves join connectivity, and provides a compact context for BASS to perform verifier-guided search over SQL semantic units.

B Structural Boundary Detection and Admissibility Filtering

This appendix provides implementation details for Structural Boundary Detection (SBD) and lightweight admissibility filtering. SBD defines the semantic-unit action space used by BASS, while admissibility filtering removes obviously malformed units before tree expansion. BASS does not use PICARD or token-level grammar-constrained decoding (Scholak et al., 2021); instead, it applies deterministic segmentation and post hoc rule/parser-based filtering.

B.1 Overview

BASS searches over SQL semantic units rather than raw tokens. A semantic unit is a locally closed SQL fragment aligned with clause-level or operator-level structure, such as a projection list, source table, join fragment, predicate group, grouping clause, ordering clause, nested subquery, or set-operation segment. Given a generated SQL candidate or continuation, SBD segments it into an ordered sequence of semantic units:

$$\text{SBD}(y) = (a_0, a_1, \dots, a_{m-1}), \quad (31)$$

where each a_i is a candidate semantic-unit action.

SBD is implemented as a deterministic rule-based segmentation layer with parser-assisted normalization when parser support is available. In our implementation, sqlglot is used when available; otherwise, the system falls back to deterministic rule-based segmentation. The active backend is recorded in run artifacts through sqlglot_available and sbd_parser_backend.

SBD follows three design principles:

- **Determinism:** identical generated SQL strings yield identical segmentations under the same boundary rules.
- **Local closure:** each segment should be locally meaningful and avoid splitting inside incomplete expressions.

Pattern	Boundary condition	Example unit
SELECT list	Projection expression or projection list is complete	SELECT COUNT(*)
FROM source	Table name, alias, or source expression is complete	FROM singer
JOIN fragment	Joined table, alias, and join condition are complete	JOIN album ON ...
WHERE predicate	Predicate or predicate group is complete	WHERE age > 30
GROUP BY	Grouping column list is complete	GROUP BY dept
HAVING	Aggregate condition is complete	HAVING COUNT(*) > 2
ORDER BY/LIMIT	Ordering expression and optional limit are complete	ORDER BY x DESC LIMIT 1
Nested query	Parentheses and subquery scope are balanced	WHERE id IN (SELECT ...)
Set operation	Top-level set-operation segment is complete	... UNION ...
WITH clause	CTE definition is complete and balanced	WITH t AS (...)
Window function	OVER(...) scope is balanced	RANK() OVER (...)

Table 8: Representative boundary rules used by Structural Boundary Detection. SBD aims to split SQL into locally closed semantic units while avoiding splits inside incomplete expressions or nested scopes.

- **SQL alignment:** boundaries should reflect SQL clause, predicate, join, subquery, or set-operation structure rather than arbitrary token positions.

B.2 Boundary Rules

Table 8 summarizes the main boundary rules used by SBD. These rules are designed to produce clause-level semantic units while avoiding splits inside nested expressions, function calls, or incomplete predicates.

B.3 Edge-Case Handling

SBD handles common SQL edge cases through scope tracking and conservative boundary decisions.

Nested subqueries. SBD tracks parenthesis depth and subquery scope. It does not split inside a nested subquery until the subquery’s parentheses are balanced. For example, a predicate such as

WHERE age > (SELECT AVG(age) FROM singer)

is treated as one predicate-level semantic unit rather than being split at the inner SELECT or FROM.

JOIN chains. For join expressions, SBD treats the joined table and its corresponding ON or USING condition as one locally closed unit whenever possible. This avoids creating a partial action such as JOIN album without the associated join condition. For multi-hop joins, each complete join fragment may be represented as a separate unit:

```
JOIN    album    ON    singer.id    =
album.singer_id
JOIN    concert  ON    album.id     =
concert.album_id
```

Set operations. For UNION, INTERSECT, and EXCEPT, SBD only splits at top-level set-operation boundaries. It does not split at set-operation keywords that appear inside nested subqueries.

Window functions. For expressions involving OVER(...) clauses, SBD tracks the window-function scope and avoids splitting inside the PARTITION BY or ORDER BY portion of the window expression.

Aggregate and scalar functions. SBD does not split inside balanced function calls such as COUNT(*), AVG(age), MAX(score), or scalar expressions such as strftime(...). A function expression is considered complete only after its parentheses are balanced.

Common table expressions. For WITH clauses, each complete CTE definition is kept intact until its subquery is balanced. SBD avoids splitting inside the CTE body before the closing parenthesis.

ORDER BY and LIMIT. When LIMIT directly modifies an ORDER BY clause, SBD keeps them in the same terminal semantic unit. This preserves the common Text-to-SQL pattern where ordering and limit jointly encode superlative intent.

B.4 Segmentation Procedure

SBD is implemented as a deterministic segmentation procedure. Given a SQL candidate or continuation y , it first normalizes the SQL string when parser support is available and then scans the token sequence from left to right. During scanning, it tracks clause keywords, parenthesis depth, subquery scope, CTE scope, function scope, and window-function scope. A boundary is emitted only when the current unit satisfies the boundary rules and does not end with an incomplete clause, predicate, join, operator, or delimiter.

At a high level, SBD performs the following steps:

1. normalize the SQL string when parser support is available;
2. tokenize the normalized SQL string;
3. track clause state, parenthesis depth, subquery scope, and incomplete-clause patterns;
4. emit a semantic unit when a valid boundary is reached;
5. apply admissibility filtering to remove malformed or incomplete units;
6. return the ordered semantic units and segmentation metadata.

This procedure is deterministic and records segmentation metadata for each generated candidate, including segmentation status, parser backend, the number of generated units, the number of admissible units, and rejection reasons.

B.5 Segmentation Examples

For the query

```
SELECT name FROM singer WHERE age >
(SELECT AVG(age) FROM singer) ORDER BY
age DESC LIMIT 1
```

SBD produces:

```
[SELECT name]
[FROM singer]
[WHERE age > (SELECT AVG(age) FROM
singer)]
[ORDER BY age DESC LIMIT 1].
```

For a query with a join chain:

```
SELECT singer.name FROM singer JOIN
album ON singer.id = album.singer_id
JOIN concert ON album.id =
concert.album_id WHERE concert.year =
2014
```

SBD produces:

```
[SELECT singer.name]
[FROM singer]
[JOIN album ON singer.id =
album.singer_id]
[JOIN concert ON album.id =
concert.album_id]
[WHERE concert.year = 2014].
```

For a query with a set operation:

```
SELECT name FROM singer WHERE country =
'US' UNION SELECT name FROM singer WHERE
country = 'UK'
```

SBD produces:

Rejection type	Condition
Empty unit	The semantic unit is empty or contains only whitespace/punctuation.
Unbalanced parentheses	Parentheses, brackets, or function scopes are not balanced.
Incomplete predicate	A predicate lacks a left-hand side, operator, or right-hand side.
Incomplete join	A JOIN fragment lacks a joined table or join condition when required.
Dangling operator	The unit ends with an operator such as =, >, AND, or OR.
Invalid clause ordering	The unit creates an invalid top-level SQL clause order.
Malformed set operation	A set operator lacks a complete left or right query segment.
Parser failure	A terminal SQL candidate fails parsing when parser support is available.

Table 9: Representative rejection rules used by the lightweight admissibility filter. The filter removes obviously malformed units but does not provide formal grammar-constrained decoding guarantees.

```
[SELECT name FROM singer WHERE country
= 'US']
[UNION]
[SELECT name FROM singer WHERE country
= 'UK'].
```

These examples illustrate that SBD aims to preserve locally meaningful SQL fragments rather than splitting at arbitrary token boundaries.

B.6 Lightweight Admissibility Filtering

After SBD segmentation, BASS applies a lightweight admissibility filter before expansion. The filter rejects obviously malformed semantic units using rule- and parser-based checks, but it is not grammar-constrained decoding.

Table 9 summarizes the main rejection rules.

Terminal validation. For terminal SQL candidates, the admissibility filter can apply parser-level validation when parser support is available. If the terminal candidate fails parsing, it is marked as malformed and receives no usable semantic-unit expansion. Terminal candidates are still ultimately evaluated by the execution checker in the verifier stage.

Difference from grammar-constrained decoding. The admissibility filter is applied after candidate generation and segmentation. It does not constrain the language model’s next-token distribution and does not guarantee that every generated continuation is syntactically valid. This differs from PICARD-style grammar-constrained decoding, which restricts token-level continuations

Field group	Description
Semantic units	Ordered semantic units produced by SBD.
Unit counts	Number of generated, admissible, and rejected units.
Segmentation status	Whether segmentation succeeds, falls back, or fails due to incomplete SQL.
Parser metadata	Parser availability and backend used for normalization.
Rejection reasons	Rule-based reasons for filtering malformed or incomplete units.

Table 10: Segmentation and admissibility metadata logged by BASS.

during decoding (Scholak et al., 2021). BASS is therefore best understood as structured search with lightweight admissibility checking, not as strict grammar-constrained decoding.

B.7 Segmentation and Filtering Metadata

For reproducibility, BASS records segmentation and admissibility metadata for each generated candidate. Table 10 summarizes the main field groups.

These logs allow us to distinguish failures caused by generation, segmentation, admissibility filtering, execution errors, and semantic-verifier errors.

B.8 Limitations of SBD

SBD is designed to provide a deterministic and practical semantic-unit action space, but it has limitations. First, because it is not a full grammar-constrained decoder, it cannot guarantee that all generated continuations are syntactically valid. Second, complex SQL dialect features may require parser support or additional boundary rules. Third, the semantic-unit granularity is heuristic: overly coarse units may reduce search flexibility, while overly fine-grained units may reintroduce token-level instability.

These limitations motivate the use of execution-gated verification after search expansion. Even if a candidate passes SBD and admissibility filtering, it must still execute successfully before receiving any semantic reward.

C Verifier Prompt and Reward Implementation

This appendix provides the full implementation details of the execution-gated rubric verifier used by BASS. The verifier is used only to score completed SQL candidates during search. It does not generate SQL, does not access gold SQL, and does not

use execution labels beyond the candidate’s own executability result. The verifier reward is used for MCTS backpropagation and final candidate selection.

C.1 Overview

BASS uses an execution-gated verifier to provide task-grounded feedback for Text-to-SQL search. Given a terminal candidate SQL query y , the verifier computes a reward in two stages. First, an execution checker determines whether y can be executed in the target database environment. Second, only executable candidates are passed to a deterministic rubric-based semantic audit. Non-executable candidates receive zero reward and bypass semantic auditing.

The reward follows the execution-gated definition in Eq. (6), where Q is the natural-language question, C is the pruned schema context, $W_{\text{exe}}(y) \in \{0, 1\}$ is the execution indicator, and $\text{Sim}_{\text{sem}}(y; Q, C) \in [0, 1]$ is the normalized semantic-audit score.

This design reflects a key property of Text-to-SQL: executability is necessary but not sufficient. A SQL query may execute successfully while still using the wrong schema elements, missing required predicates, applying incorrect aggregation, or answering a related but different question. The semantic audit therefore provides a finer-grained reward signal among executable candidates.

C.2 Execution Gate

BASS first applies an execution gate before semantic auditing. A candidate receives $W_{\text{exe}} = 1$ only if it executes successfully within the configured timeout under the corresponding benchmark database. Candidates with syntax errors, invalid table or column references, runtime errors, malformed joins or subqueries, parser failures, or timeouts receive $W_{\text{exe}} = 0$. These candidates bypass semantic auditing and receive zero terminal reward:

$$R(y; Q, C) = 0 \quad \text{if} \quad W_{\text{exe}}(y) = 0. \quad (32)$$

This execution gate reduces verifier cost and prevents non-executable SQL from receiving positive semantic reward. It also separates syntactic or database-level failures from semantic near misses among executable candidates.

C.3 Stage 2: Rubric-Based Semantic Audit

If $W_{\text{exe}}(y) = 1$, BASS applies a rubric-based semantic audit. The audit evaluates whether the can-

didate SQL is semantically aligned with the question and schema. It assigns five dimension-level scores:

- **Schema grounding:** whether the SQL uses appropriate tables, columns, aliases, and schema relations.
- **Intent matching:** whether the SQL answers the main intent of the natural-language question.
- **Condition correctness:** whether filters, predicates, constants, and predicate scopes are correct.
- **Aggregation and sorting:** whether aggregation, grouping, ordering, and limit clauses are correct.
- **Join and compositionality:** whether joins, nested queries, set operations, and multi-clause structure are semantically appropriate.

Each dimension receives a score $s_j \in \{0, 1, 2\}$, where 0 indicates incorrect or missing behavior, 1 indicates partially correct behavior, and 2 indicates correct behavior. The normalized semantic score follows Eq. (5), so $\text{Sim}_{\text{sem}}(y; Q, C) \in [0, 1]$.

The final reward is computed as in Eq. 6. The multiplicative form ensures that non-executable SQL receives zero reward, while executable candidates are ranked by semantic alignment.

C.4 Verifier Output Schema

The semantic auditor is required to return a structured JSON object. The expected schema is:

```
{
  "schema_grounding": 0-2,
  "intent_matching": 0-2,
  "condition_correctness": 0-2,
  "aggregation_sorting": 0-2,
  "join_compositionality": 0-2,
  "total_score": 0-10,
  "sim_sem": 0.0-1.0,
  "is_consistent": true/false,
  "failure_reason": "...",
  "rationale": "brief"
}
```

During parsing, the implementation clamps each dimension score to $[0, 2]$, recomputes `total_score` as the sum of the five dimension-level scores, and recomputes `sim_sem` as `total_score/10`. The recomputed value is used as the reward component rather than blindly trusting the value returned by the verifier.

If the verifier output cannot be parsed as valid JSON, the candidate receives

$$\text{Sim}_{\text{sem}}(y; Q, C) = 0. \quad (33)$$

The system marks the candidate with `semantic_audit_parse_error=true` and records the failure reason in the run artifacts.

C.5 Verifier Prompt

The exact verifier prompt used for semantic auditing is shown below. The prompt asks the verifier to judge semantic correctness under the given question and schema, rather than stylistic similarity to any specific SQL formulation.

You are a SQL semantic auditor. Your task is to evaluate whether a candidate SQL query correctly answers the user question under the given database schema.

Return only a valid JSON object. Do not include markdown or extra prose.

Evaluate the SQL using five dimensions. Each dimension must be an integer or numeric score from 0 to 2.

1. `schema_grounding`: Are the selected tables, columns, aliases, and schema relations correct?
2. `intent_matching`: Does the SQL answer the main intent of the natural-language question?
3. `condition_correctness`: Are WHERE/HAVING predicates, filters, constants, and scopes correct?
4. `aggregation_sorting`: Are aggregation, GROUP BY, ORDER BY, and LIMIT clauses correct?
5. `join_compositionality`: Are joins, nested queries, set operations, and multi-clause structure correct?

Executability alone is not sufficient. Penalize executable SQL that misses key constraints, uses wrong schema elements, applies wrong aggregation, or answers a related but different question.

Do not penalize stylistically different SQL formulations if they are semantically equivalent and would return the correct answer under the given schema and question. Focus on denotational and intent equivalence.

Return JSON with the following fields:

```
{
  "schema_grounding": ...,
  "intent_matching": ...,
  "condition_correctness": ...,
  "aggregation_sorting": ...,
  "join_compositionality": ...,
  "total_score": ...,
  "sim_sem": ...,
  "is_consistent": ...,
  "failure_reason": "...",
  "rationale": "..."
}
```

Question: {Q}

Schema/Context: {C}

Candidate SQL: {y}

C.6 Verifier Decoding and Determinism

All semantic-audit calls use deterministic decoding with temperature 0.0. The verifier model name, prompt version, decoding configuration, parse status, dimension-level scores, and semantic-audit statistics are recorded in the run artifacts.

The verifier is instructed to evaluate semantic equivalence rather than string similarity. In particular, it should not penalize stylistically different SQL formulations if they are semantically equivalent under the given schema and question. This is important because many Text-to-SQL questions admit multiple valid SQL formulations that produce the same denotation.

C.7 Reward Computation and Backpropagation

After the execution gate and semantic audit, BASS uses the terminal reward in Eq. (6) for backpropagation.

This reward is propagated along the visited MCTS path. For each visited node v , BASS increments the visit count $N(v)$, adds the terminal reward to the accumulated reward $Q(v)$, updates the best subtree reward, and updates stagnation statistics. Under `final_selector=best_reward`, the final SQL returned by BASS is the executable candidate with the highest verified reward.

C.8 Mitigating Verifier Failure and Reward Hacking

The verifier is not assumed to be perfect. It may over-score plausible but incorrect SQL, under-score semantically equivalent formulations, or be sensitive to schema presentation. BASS uses several safeguards to reduce these risks.

Execution gate. The semantic audit is applied only to executable candidates. This prevents non-executable SQL from receiving positive semantic reward.

Dimension-level scoring. The verifier returns five explicit dimension-level scores instead of a single free-form preference judgment. This makes the reward more interpretable and allows failure cases to be inspected by category.

Structured output validation. The implementation recomputes the total score and normalized semantic score from dimension-level scores. Invalid or unparseable verifier outputs receive zero semantic score.

Field group	Description
Execution gate	Execution result, timeout status, and execution error type.
Audit invocation	Whether semantic auditing is invoked after execution succeeds.
Rubric scores	Dimension-level scores for schema grounding, intent matching, predicate correctness, aggregation/ordering, and join/compositionality.
Semantic score	Normalized Sim_{sem} score computed from rubric dimensions.
Terminal reward	Final reward $W_{\text{exe}} \cdot \text{Sim}_{\text{sem}}$.
Parsing status	Whether verifier JSON parsing succeeds or fails.
Verifier metadata	Verifier model identifier and prompt-template version.

Table 11: Verifier-related metadata logged by BASS for reproducibility and failure analysis.

Semantic-equivalence instruction. The prompt explicitly instructs the verifier not to penalize SQL formulations that are stylistically different but semantically equivalent under the question and schema.

Cost-aware gating. Since semantic auditing is invoked only after execution succeeds, BASS avoids spending verifier calls on clearly invalid candidates and reduces unnecessary audit cost.

Despite these safeguards, verifier errors remain possible. We therefore report verifier-related failure cases and additional qualitative analyses in Appendix E.

C.9 Logged Verifier Fields

For each evaluated terminal candidate, BASS records verifier-related metadata. Table 11 summarizes the main field groups.

D Experimental Details, Cost Accounting, and Sensitivity

This appendix reports the experimental protocol, artifact use, default configuration, computational setup, stagnation-penalty definition, additional cost statistics, hyperparameter sweep settings, and run artifacts used by BASS. We separate externally reported benchmark positioning from controlled internal analyses because prior systems differ in backbone access, prompting strategy, verifier usage, rollout budgets, and cost accounting. Results copied from prior work are used only for external performance positioning, while BASS runs are logged under a unified protocol that records both task performance and realized inference cost.

D.1 Benchmarks and Evaluation Protocol

Benchmarks. We evaluate BASS on Spider and BIRD. Spider emphasizes cross-domain SQL generalization over unseen database schemas (Yu et al., 2018), while BIRD contains larger schemas and more realistic query complexity (Li et al., 2023).

Evaluation splits and metrics. Main benchmark results are reported on Spider Dev, Spider Test, and full BIRD Dev. Following standard practice, we report execution accuracy (EX) on Spider and BIRD, and additionally report Valid Efficiency Score (VES) on BIRD.

Execution environment. For executability checks and official evaluation, we use the benchmark-provided database files and evaluator interfaces. A generated SQL query is treated as executable only if it runs successfully under the corresponding database environment within the configured timeout. Syntax errors, invalid table or column references, malformed joins, runtime errors, parser failures, and timeouts are treated as execution failures. These candidates receive $W_{\text{exe}} = 0$ and bypass semantic auditing.

Artifact use and terms. Our experiments use existing public scientific artifacts, including the Spider and BIRD Text-to-SQL benchmarks, benchmark-provided database files and official evaluation scripts, Qwen2.5-Coder backbone models (Hui et al., 2024), additional backbone models used in SDS analyses including SmoLLM3-3B (Bakouch et al., 2025), Qwen3-4B (Yang et al., 2025a), and gpt-oss-20b (OpenAI, 2025), retrieval models used for schema pruning such as BGE-M3 (Chen et al., 2024), and standard lexical retrieval components such as BM25 (Robertson et al., 1994). We cite the creators of the datasets, models, retrieval components, and evaluation resources where applicable. These artifacts are used only for research evaluation and method development under their intended benchmark or model-use settings.

We follow the licenses and access terms associated with the original datasets, models, and evaluation scripts, and do not redistribute third-party artifacts beyond what is permitted by their respective terms. We do not collect new human-subject data or introduce new personally identifying information. The experiments are conducted on existing benchmark questions, database schemas, and database contents released for Text-to-SQL research. Our outputs consist of generated SQL queries, execu-

tion results, verifier scores, and aggregate evaluation statistics. Any released run artifacts should exclude private credentials, API keys, and non-anonymized user information.

Evaluator and package settings. We use the official Spider and BIRD evaluation interfaces for execution-based evaluation. During search, generated SQL queries are executed against benchmark-provided SQLite databases under the configured timeout. Parser-assisted normalization and Structural Boundary Detection use the configured SQL parser backend when available, and otherwise fall back to deterministic rule-based segmentation. Schema pruning uses the dense retrieval, reranking, graph-routing, and BM25 settings described in Appendix A. The exact evaluator paths, parser backend, timeout settings, retrieval parameters, package versions, model identifiers, and model-serving backend are recorded in `run_config.json`.

D.2 Subsampled Development Set

For expensive controlled analyses, including controlled cost comparison, hyperparameter sensitivity, verifier analysis, and additional-backbone experiments, we use the BIRD Subsampled Development Set (SDS), following prior work. The SDS contains 147 examples, corresponding to approximately 10% of BIRD Dev, with 81 simple, 54 moderate, and 12 challenging questions. Unless otherwise stated, SDS analyses use Qwen2.5-Coder-7B in the zero-shot setting with maximum rollout budget $B = 50$.

The main benchmark results and component ablations in the main paper are reported on full evaluation splits. SDS is used only for controlled analyses where full-split sweeps would be computationally expensive.

D.3 Default Configuration

BASS is instantiated on the Qwen2.5-Coder family, including 1.5B, 3B, 7B, and 14B backbones. All BASS results are obtained in the zero-shot setting, where no few-shot exemplars or in-context demonstration pairs are included in the prompt.

Unless otherwise specified, BASS uses UCT selection, execution-gated semantic reward, best-reward final selection, and stagnation-aware budget control. We do not use PUCT because the deployed generator interface does not expose token-level log-probabilities, and we do not use PICARD or token-level grammar-constrained decoding.

Warm-start policy. The warm-start SQL is generated by the same backbone used for search under the same zero-shot schema context. No gold SQL or stronger teacher model is used. The warm-start generation cost is included in the reported generator calls, token usage, and wall-clock time.

Generator decoding. All generator calls use the same decoding configuration across Direct, Best-of- N , and BASS. For BASS, the generator is used for warm-start SQL generation and semantic-unit proposal generation. Direct decoding draws one candidate, while Best-of- N draws N independent candidates under the same generator configuration. BASS uses the same generator configuration for proposal generation, so controlled quality–cost comparisons do not rely on different generator sampling policies. The concrete generator decoding parameters and model identifiers are recorded in `run_config.json`. The semantic verifier is decoded deterministically with temperature 0.0.

Computational setup and reporting. We report computational experiments under a unified logging protocol. Main benchmark results use Qwen2.5-Coder backbones at 1.5B, 3B, 7B, and 14B scales and are reported as mean and standard deviation over five runs. Unless otherwise specified, BASS uses a maximum rollout budget of $B = 50$ per instance. Since BASS is an anytime search method, this budget is an upper bound rather than a fixed realized cost. We therefore additionally report realized budget statistics, including executed rollouts, generator and verifier calls, execution calls, token usage when available, wall-clock time when available, and early stopping behavior.

Experiments are run on NVIDIA A100 80GB PCIe GPUs. Controlled analyses on BIRD-SDS, including quality–cost comparison, hyperparameter sensitivity, verifier analysis, and additional-backbone experiments, are diagnostic experiments under fixed configurations. For these analyses, we report the evaluation split, model backbone, method variant, maximum rollout budget, and per-instance average cost statistics when applicable. The concrete backend, model identifiers, decoding parameters, timeout settings, package versions, evaluator paths, and run-level summaries are recorded in `run_config.json` and `run_summary.json`.

Parameter	Default	Description
MAX_ROLLOUTS	50	Maximum rollout budget per instance.
C_exp	2.5	Exploration coefficient in UCT.
BOOTSTRAP_PROPOSAL_COUNT	5	Number of bootstrap proposals.
lambda_stag	0.1	Scale of the stagnation penalty.
C_max	1.0	Cap for the stagnation penalty.
C_sat	0.5	Saturation penalty for exhausted branches.
tau_stag	3	Threshold for activating the stagnation penalty.
tau_exh	8	Threshold for marking a branch exhausted.
max_unchanged_champion_steps	10	Early-stop threshold when the best candidate does not change.
verifier_temperature	0.0	Deterministic verifier decoding.
COARSE_TOP_K	60	Dense-retrieval candidates before reranking.
BM25_TOP_K	15	Lexical BM25 backup columns.
TopTables	5	High-score tables retained as backup.

Table 12: Default BASS hyperparameters used in the main experiments.

D.4 Stagnation Penalty and Early Stopping

For each node v , BASS maintains a stagnation counter $\sigma(v)$, which records the number of consecutive rollouts during which the best verified reward in the subtree rooted at v does not improve. Let ΔR_v denote the recent improvement of the best subtree reward. We define

$$g(v) = \mathbb{I}[\Delta R_v \leq 0], \quad (34)$$

and

$$h(v) = \exp(\max(0, \sigma(v) - \tau_{\text{stag}})). \quad (35)$$

The bounded stagnation penalty is

$$\Psi(v) = \min(C_{\text{max}}, \lambda_{\text{stag}} h(v) g(v)). \quad (36)$$

Here, λ_{stag} controls the penalty scale, τ_{stag} determines when stagnation begins to affect selection, and C_{max} caps the maximum penalty. The cap prevents short-term verifier noise from permanently suppressing a branch.

The stagnation-aware selection score follows the main-text definition in Eq. (7). This appendix expands the implementation of the penalty term $\Psi(v)$ and the exhaustion-based virtual saturation rule.

BASS also uses a stricter exhaustion threshold for early stopping:

$$\text{Exhausted}(v) = \mathbb{I}[\sigma(v) > \tau_{\text{exh}}]. \quad (37)$$

In implementation, exhausted branches receive an additional virtual saturation penalty:

$$\begin{aligned} \text{Score}_{\text{sat}}(u, v) &= \text{Score}(u, v) \\ &\quad - C_{\text{sat}} \cdot \text{Exhausted}(v). \end{aligned} \quad (38)$$

Search terminates when all expanded frontier nodes are exhausted. Thus, the realized number of rollouts can be smaller than the nominal maximum budget B .

Method	EX	Tokens	LLM Calls	Exec Calls
Direct	39.46	1,742	1.00	0.00
Best-of- $N@5$	48.30	8,718	5.00	4.43
Best-of- $N@10$	48.98	17,387	10.00	8.18
Full BASS	68.03	77,372	36.43	12.03

Table 13: Controlled internal cost comparison on BIRD-SDS with Qwen2.5-Coder-7B. Full BASS uses $B = 50$.

D.5 Controlled Internal Cost Comparison

We conduct a controlled internal quality–cost comparison on BIRD-SDS with Qwen2.5-Coder-7B. All methods use the same schema-pruning pipeline, evaluator, generator decoding policy, and logging protocol. Full BASS uses a maximum rollout budget $B = 50$. This analysis is not intended as a fully matched-compute comparison with external systems; rather, it compares different inference strategies under our unified experimental protocol.

Tokens, LLM Calls, and Exec Calls denote per-instance averages. LLM Calls includes generator and verifier model calls, while Exec Calls denotes database execution checks. Figure 2 in Section 5.1 visualizes the main quality–cost trend, while Table 13 provides the corresponding numeric cost statistics.

Direct decoding is cheapest but achieves the lowest EX. Increasing repeated sampling from Best-of- $N@5$ to Best-of- $N@10$ nearly doubles total calls and token usage, but yields only a marginal EX improvement. In contrast, Full BASS incurs higher inference cost but reaches substantially higher EX. The corresponding figure in the main paper emphasizes that BASS obtains a larger accuracy gain than naive repeated sampling, rather than simply increasing the number of calls.

Parameter	Values tested
MAX_ROLLOUTS	20, 30, 40, 50, 60, 70, 80
C_exp	0.5, 1.0, 1.41, 1.5, 2.0, 2.5, 3.0
BOOTSTRAP_PROPOSAL_COUNT	1, 3, 5, 7, 10
lambda_stag	0.0, 0.1, 0.2, 0.5
C_sat	0.0, 0.5, 1.0, 2.0
tau_stag	1, 3, 5
tau_exh	4, 8, 12
max_unchanged_champion_steps	5, 10, 15, 20, 25, 30

Table 14: Hyperparameter values tested in the one-at-a-time sensitivity analysis.

D.6 Hyperparameter Sweep Settings

We conduct one-at-a-time hyperparameter sensitivity analysis on BIRD-SDS with Qwen2.5-Coder-7B. For each sweep, we vary one hyperparameter while keeping the others fixed at the default configuration in Table 12. Figure 3 in Section 5.4 reports the corresponding EX trends.

The sweep is intended to evaluate local robustness around the default configuration rather than to perform exhaustive hyperparameter optimization. The results show that moderate rollout and budget-control settings generally work best: very small rollout budgets limit exploration, while overly weak or overly aggressive stagnation control can reduce accuracy.

D.7 Additional Backbone Results on BIRD-SDS

To examine whether the benefit of BASS is specific to the Qwen2.5-Coder backbone family, we conduct additional experiments on BIRD-SDS with SmolLM3-3B, Qwen3-4B, and gpt-oss-20b. These experiments use the same SDS split and compare Direct decoding, Best-of- $N@10$, and BASS under the same evaluation protocol.

As shown in Figure 4, BASS consistently improves over both Direct decoding and Best-of- $N@10$ across all three additional backbones. These auxiliary SDS results suggest that the benefit of structured search is not limited to the Qwen2.5-Coder family, although the main full-split comparisons remain focused on Qwen2.5-Coder for controlled evaluation.

D.8 Cost Accounting and Run Artifacts

BASS is an anytime search method: the nominal budget B is the maximum number of rollouts, while the realized cost may be lower due

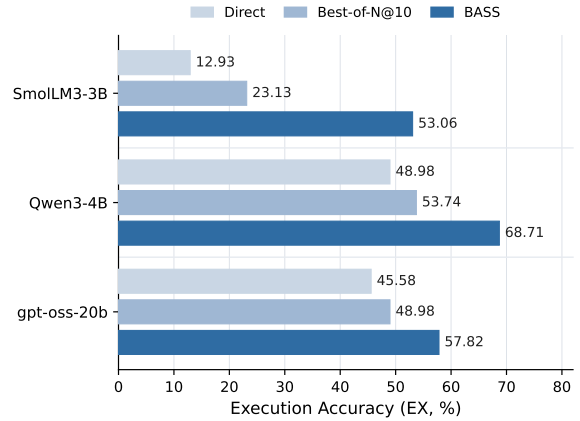


Figure 4: Additional backbone results on BIRD-SDS. BASS consistently outperforms Direct decoding and Best-of- $N@10$ across SmolLM3-3B, Qwen3-4B, and gpt-oss-20b.

to stagnation- and exhaustion-aware early stopping. We therefore report realized budget statistics in addition to task metrics.

Cost accounting. For each instance, BASS records executed rollouts, execution checks, semantic-audit calls, generator calls, verifier calls, token usage when available, wall-clock time when available, and the termination reason. Execution checks serve as a low-cost gate: non-executable candidates receive zero reward and bypass semantic auditing. Semantic-audit calls are made only for executable candidates under `reward_mode=execution_semantic`. Token fields are marked as unavailable when the backend does not return token accounting.

Run artifacts. Each run writes a structured run directory containing the configuration, per-instance predictions, run-level summaries, and evaluator outputs. Specifically, `run_config.json` records dataset, split, model, decoding settings, hardware configuration, package versions, and search configuration; `sample_table.jsonl` stores per-instance predictions, rewards, execution status, cost fields, and termination reasons; and `run_summary.json` aggregates accuracy, cost, and termination statistics. We also store evaluator input files and parsed Spider/BIRD evaluation outputs.

This logging protocol makes the distinction between nominal budget, realized cost, and final task performance explicit.

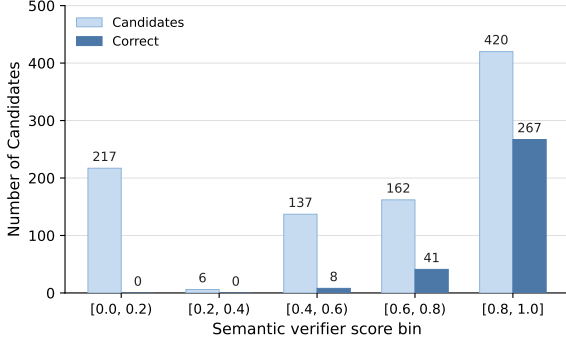


Figure 5: Distribution of executable audited candidates across semantic-verifier score bins on BIRD-SDS with Qwen2.5-Coder-7B. For each score bin, light bars show the total number of candidates, and dark bars show the number of correct candidates under the official evaluator. Higher-score bins contain substantially more correct candidates, supporting the usefulness of the semantic verifier for ranking executable SQL candidates.

E Verifier Analysis and Case Studies

This appendix provides additional analyses for understanding verifier behavior and remaining errors in BASS. We focus on three questions: whether the semantic verifier provides a useful ranking signal among executable candidates, what types of errors remain after structured search and verification, and how logged run artifacts can be used for qualitative inspection. Detailed verifier prompts and reward implementation are provided in Appendix C; cost logging and experimental details are provided in Appendix D.

E.1 Verifier Candidate Distribution

Execution checking is useful for filtering malformed SQL, invalid schema references, runtime errors, and timeouts. However, executability alone is not sufficient for Text-to-SQL correctness: an executable query may still omit required predicates, use the wrong aggregation scope, follow an incorrect join path, or answer a related but different question. BASS therefore applies semantic auditing only after the execution gate and uses the resulting Sim_{sem} score as a reward signal among executable candidates.

To examine whether this semantic score is informative, we group executable audited candidates from BIRD-SDS by semantic-verifier score bin and count how many candidates in each bin are correct under the official evaluator. Figure 5 shows the resulting distribution.

The distribution shows that low-score bins con-

tain few or no correct candidates, even when the number of executable audited candidates is non-trivial. In contrast, the highest score bin $[0.8, 1.0]$ contains both the largest number of candidates and the largest number of correct candidates, with 267 correct candidates out of 420. This pattern suggests that Sim_{sem} provides a useful ranking signal for prioritizing executable SQL candidates. At the same time, high-scoring candidates are not guaranteed to be correct, indicating that semantic auditing remains imperfect and can still over-score plausible near misses.

A representative false positive occurs on a challenging BIRD Dev question asking which users added a bounty of 50 to posts whose titles mention “variance.” The reference result contains two qualifying users, `chl` and `shabbychef`. The selected SQL adds an unsupported `LIMIT 1` and returns only `chl`, but the verifier assigns $\text{Sim}_{\text{sem}} = 1.0$, with the maximum score on all five rubric dimensions. The query is executable and its returned row is valid, yet the answer is incomplete. This example illustrates how a plausible partial result can receive a high semantic score.

E.2 Candidate-Pool Oracle and Selection Errors

We separately measure whether search generates a correct candidate and whether the verifier selects one. A question is oracle-solvable when at least one candidate in its completed BIRD-SDS pool is correct under the official evaluator. The pool oracle solves 112 of 147 questions (76.19%), whereas reward-based Top-1 solves 100 (68.03%). Ties in verifier reward are resolved by selecting the earliest-generated candidate. Thus, among the 47 Top-1 errors, 35 are generation failures for which the pool contains no correct candidate, and 12 are selection errors for which a correct candidate exists but is not selected. Conditional on oracle solvability, the selection accuracy is 100/112 (89.29%).

The 8.16-point oracle gap shows that improving candidate generation remains the larger opportunity, although verifier calibration is also material. The official evaluator is used only for this post-hoc meta-evaluation and never as a search reward.

E.3 Residual Error Taxonomy

Despite the gains from structured search and execution-gated verification, several recurring error types remain. Table 15 summarizes the main failure modes observed in BASS runs.

Failure mode	Description
Schema-linking error	The generated SQL uses a plausible but incorrect table or column, or follows an incorrect join path.
Schema-pruning error	A required table or column is absent from the compact schema context, preventing the correct SQL from being generated.
Predicate error	The SQL omits, weakens, or changes a required condition.
Aggregation or ordering error	The SQL uses incorrect aggregation, grouping, ordering, limit, or sorting behavior.
Compositional error	The query requires interacting operations, such as nested subqueries, GROUP BY/HAVING, or multi-hop joins, and the generated SQL fails to compose them correctly.
Executable near miss	The SQL executes successfully and is partially aligned with the question, but answers a related rather than exact query.
Candidate-generation failure	The search tree does not contain a sufficiently good candidate, often because the base model fails to propose the required structure.
Verifier ambiguity	The semantic audit over-scores a plausible but incorrect candidate, or under-scores a semantically equivalent formulation.

Table 15: Common remaining failure modes observed in BASS runs.

These failures clarify the limitations of using execution and semantic auditing as search rewards. Some errors arise before verification, such as schema-pruning or candidate-generation failures. Others arise after execution succeeds, such as near-miss candidates that differ from the target SQL by a small but decisive semantic constraint. The verifier reduces this problem by assigning graded semantic scores, but it does not eliminate all ambiguity.

E.4 Interaction Between Verifier Quality and Budget Allocation

The verifier and the budget-allocation policy are tightly coupled. The verifier determines which terminal candidates receive positive semantic reward, while the stagnation-aware controller determines how long a region of the search tree remains worth exploring. When a branch repeatedly produces weak or near-duplicate candidates, its verified reward saturates and stagnation-aware suppression reduces its selection probability. Conversely, branches that continue to yield executable and semantically stronger candidates remain competitive.

This interaction means that verifier quality affects more than final reranking. A useful veri-

fier can improve budget allocation by making reward improvements more informative. Conversely, noisy verifier scores can make semantically weak branches appear artificially promising, reducing the effectiveness of stagnation-aware control. The candidate distribution in Figure 5 supports the use of the semantic verifier as a search signal, while the remaining false positives motivate future work on stronger verifier calibration.

E.5 Search-Trace Inspection

BASS records search-level metadata that can be used to inspect how candidates are generated, verified, and selected. For each candidate or selected trajectory, the run artifacts may include:

- semantic units produced by SBD;
- segmentation status and error type;
- admissibility filtering counts and rejection reasons;
- execution success indicator W_{exe} ;
- semantic score Sim_{sem} ;
- verifier dimension scores;
- terminal reward;
- realized rollout count;
- selected SQL source;
- termination reason.

These logs make it possible to reconstruct why one candidate was preferred over alternatives. For example, a selected candidate may be executable and receive a high semantic-audit score, while another candidate may be rejected because it fails execution, omits a required predicate, or is removed by admissibility filtering.