HARNESSSQL: HARNESS-NATIVE TRAINING FOR SQL AGENTS IN REALISTIC DATABASE ENVIRONMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Text-to-SQL models are commonly trained to map questions directly to static queries, whereas real-world database agents operate through stateful, multi-turn interaction with live databases—inspecting schemas, executing probe queries, diagnosing errors, and revising hypotheses. This creates a critical train–deploy mismatch, as the execution harness that mediates this interaction is introduced only at inference time. To bridge this gap, we propose HARNESSSQL, a harness-native post-training framework that preserves the full interaction structure throughout both supervised fine-tuning and reinforcement learning. HARNESSSQL builds isolated, executable database environments paired with hidden execution oracles, rolls out teachers directly inside the target SQL harness, and retains only verified trajectories for full-sequence SFT, followed by execution-reward RL. Across Spider 2.0-SQLite, HARNESSSQL dramatically boosts the execution accuracy of compact models, raising Qwen3-8B from 15.5% to 45.2% and Qwen3-14B from 22.2% to 54.8%, while transferring effectively to out-of-distribution interactive benchmarks such as BIRD-Interact and LiveSQLBench. Our findings demonstrate that training database agents directly within their execution harness is essential for mastering complex, long-horizon database workflows.

1 INTRODUCTION

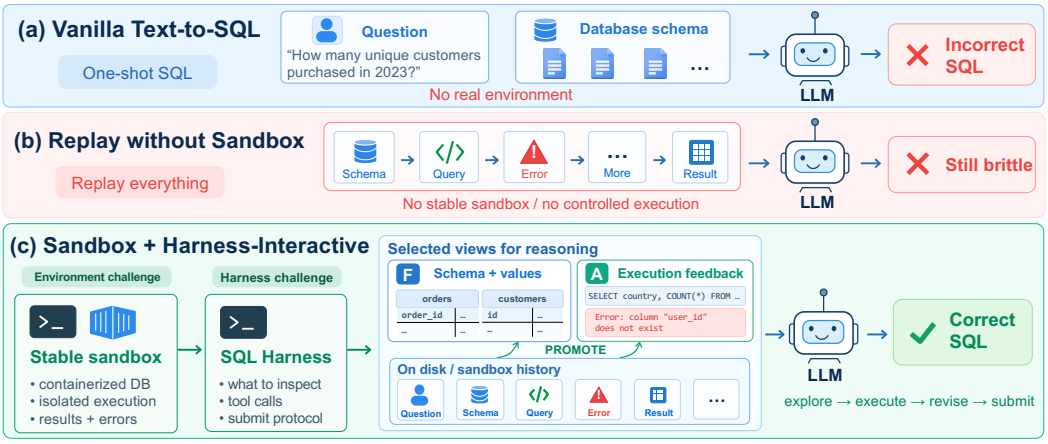

Figure 1: From one-shot Text-to-SQL to harness-interactive database agents. A stable sandbox and structured SQL harness enable iterative exploration, execution feedback, revision, and submission.

Text-to-SQL has traditionally been formulated as a direct mapping from a natural-language question and database schema to an executable SQL query (Yu et al., 2018; Li et al., 2023). This abstraction has enabled substantial progress, but it increasingly diverges from how modern systems solve realistic database tasks. Benchmarks such as Spider2.0(Lei et al., 2025) place agents in large, unfamiliar database environments where the information required to solve a task cannot always be captured in a static prompt. An agent may need to inspect tables, examine schemas and values, execute intermediate queries, diagnose errors, and revise its solution over multiple turns. Recent systems have consequently begun to treat Text-to-SQL as an interactive problem, using tools and execution

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

feedback to reason over databases rather than generating a query in a single pass (Deng et al., 2025; Li et al., 2026). This shift raises a corresponding question: *how should such SQL agents be trained?*

Figure 1 illustrates why interactive SQL agents require more than simply adding tools at inference time. Even for a seemingly simple request such as "How many unique customers purchased in 2023?", the agent may not know in advance which tables, keys, or values are relevant(Pham et al., 2026). It may first inspect the schema, issue a plausible query, encounter an execution error such as referencing a nonexistent `user_id` column, and then use the returned database state to revise its hypothesis before submitting a correct query. This process exposes three coupled challenges. First, the environment must be executable and reproducible: the agent needs access to authentic schemas, values, query results, and runtime errors rather than a static textual approximation. Second, the harness itself shapes the policy: which inspection and execution tools are available, what observations are exposed, and how submission and termination are defined determine the sequence of decisions the model can make. Third, the supervision must capture the interaction process rather than only the terminal SQL: a correct final query does not teach the model when to inspect the schema, how to interpret an execution failure, or how to recover from an incorrect intermediate hypothesis(Guo et al., 2026). Recent interactive SQL benchmarks and harness studies increasingly expose these requirements at evaluation time(Xu et al., 2025; Yao et al., 2026a), yet most SQL post-training still focuses on static question–SQL supervision or terminal execution correctness. This creates a fundamental train–deploy mismatch: models are trained to produce answers, but deployed as policies that must explore, execute, revise, and terminate through a concrete database harness.

To resolve these challenges, we introduce HARNESSSQL, a harness-native post-training framework that aligns the supervision of database agents directly with their runtime execution interface. HARNESSSQL consists of three core components: **1. Executable SQL Environment Construction.** We instantiate isolated database sandboxes that provide authentic schemas, query execution, runtime feedback, and hidden executable verification, enabling agent trajectories to be generated and evaluated against real database states. **2. SQL-Specific Harness Adaptation.** Rather than directly adopting a generic agent harness, we adapt its interaction interface and execution protocol to the characteristics of SQL tasks. The resulting harness exposes task-relevant observations and actions while reducing unnecessary interaction complexity, making teacher rollouts more reliable and the resulting trajectories easier for compact models to learn from. **3. Harness-Native Post-Training.** We collect oracle-verified full interaction trajectories directly inside the adapted harness and use them for supervised fine-tuning, followed by execution-reward reinforcement learning in the same environment. This two-stage training process teaches the model not only to produce correct SQL, but also to acquire the complete interaction policy for exploration, execution, recovery, and submission.

Empirical results demonstrate that harness-native training substantially elevates compact models. On Spider 2.0-SQLite, HARNESSSQL improves Qwen3-8B execution accuracy from 15.5% to 45.2% and Qwen3-14B from 22.2% to 54.8% after SFT and RL. Furthermore, the learned policies transfer effectively to out-of-distribution interactive environments, reaching 9.0%/11.3% on BIRD-Interact Mini and 24.8%/28.5% on LiveSQLBench Base-Lite for 8B/14B models. Controlled comparisons confirm that harness-native post-training significantly outperforms conventional non-harness fine-tuning, highlighting the importance of aligning training with the deployment interaction interface.

Our contributions are threefold: **(1) Problem Formulation:** We identify a train–deploy mismatch in database agents and formulate interactive Text-to-SQL as long-horizon policy learning grounded in an executable harness, highlighting the coupled roles of environment, interaction interface, and trajectory-level supervision. **(2) HARNESSSQL Framework:** We develop a harness-native pipeline spanning sandbox environment construction, dedicated SQL harness design, and end-to-end full-trajectory post-training via SFT and execution-reward RL. **(3) State-of-the-Art Compact Agents:** We show that HARNESSSQLsubstantially improves 8B/14B open-weight models on Spider 2.0 and transfers to out-of-distribution interactive benchmarks, demonstrating the effectiveness of training database agents natively within their deployment environment.

2 PROBLEM SETTING

2.1 SQL AGENTS AS PARTIALLY OBSERVED POLICIES

We formalize interactive Text-to-SQL as a partially observed Markov decision process (POMDP) mediated by an execution harness. A task instance is defined as a tuple $x = (q, \mathcal{D}, h, v)$, where q is

the natural-language user instruction, $\mathcal{D}$ denotes the underlying database instance, h specifies the execution harness, and v is a hidden executable oracle.

At each turn $t \in [1, T]$, the agent receives an observation $o_t \in \mathcal{O}$ emitted by the harness and generates an action $a_t \in \mathcal{A}$ according to its policy $\pi_\theta(a_t \mid s_t)$, where the visible state $s_t = (q, a_1, o_1, \ldots, a_{t-1}, o_{t-1})$ comprises the dialogue history. The action a_t consists of natural-language reasoning accompanied by either a tool call (e.g., schema lookup or query execution) or a terminal submission. An entire interaction episode is represented as:

$$\tau = (q, a_1, o_1, a_2, o_2, \ldots, a_T, o_T, y), \tag{1}$$

where y is the final submitted SQL artifact extracted from a_T. Crucially, the agent never observes the full ground-truth state of $\mathcal{D}$ directly; instead, $o_t = h(\mathcal{D}, a_t)$ is a harness-dependent projection bounded by system protocols (such as row truncations, query timeouts, and error message formatting).

Task success is evaluated over the complete trajectory by a joint reward function:

$$R(\tau; x) = R_{\text{result}}(y, v) \cdot R_{\text{protocol}}(\tau, h). \tag{2}$$

Here, $R_{\text{result}}(y, v) \in \{0, 1\}$ evaluates the denotational execution equivalence of the submitted artifact y against the hidden verifier v over $\mathcal{D}$, rather than relying on surface-form string matching. $R_{\text{protocol}}(\tau, h) \in \{0, 1\}$ verifies compliance with harness invariants, including read-only query constraints, adherence to action schemas, absence of repeated redundant queries, and termination within the step budget T.

2.2 The Train–Deploy Mismatch

Existing SQL post-training frameworks predominantly conceptualize Text-to-SQL as a one-shot conditional generation task. Formally, conventional supervision maximizes the likelihood of the golden query y^* conditioned on the prompt and a static schema description $\mathcal{S}$:

$$\mathcal{L}_{\text{static}}(\theta) = -\log \pi_\theta(y^* \mid q, \mathcal{S}). \tag{3}$$

Even recent reasoning-oriented methods that incorporate chain-of-thought traces c still formulate the objective as $\pi_\theta(c, y^* \mid q, \mathcal{S})$, terminating statically without environment interaction.

When deploying such a model as an interactive agent, a severe distributional and structural mismatch emerges across three orthogonal dimensions:

1. Marginalizing out intermediate states. By treating the final query y as the sole optimization target, static training marginalizes over the interaction trajectory:

$$p(y \mid q, S) = \sum_{\tau \setminus \{y\}} p(\tau \mid q, S). \tag{4}$$

Yet in interactive environments, transitions $p(o_t \mid s_t, a_t)$ are determined by live database execution. Discarding these intermediate states removes the procedural policy for exploration, execution, and termination.

2. Policy blindness to execution feedback. A deployed SQL agent operates under closed-loop control:

$$a_t \sim \pi_\theta(\cdot \mid q, a_1, o_1, \ldots, a_{t-1}, o_{t-1}). \tag{5}$$

Static supervision never exposes the model to runtime observations such as execution errors or empty results, creating exposure bias when such feedback must be handled at inference time.

3. Harness-oblivious action spaces. Bridging this mismatch requires *harness-native training*: explicitly optimizing $\pi_\theta(a_t \mid s_t)$ over trajectories τ collected directly within the deployment harness h, driven by execution-grounded rewards $R(\tau; x)$.

3 Methodology

Figure 2 illustrates the HarnessSQL pipeline. To resolve the environment, harness, and supervision mismatches identified in Section 2, HarnessSQL couples hermetic sandbox construction with an SQL-specific interaction contract, synthesizing verified trajectories for full-sequence SFT and execution-reward RL.

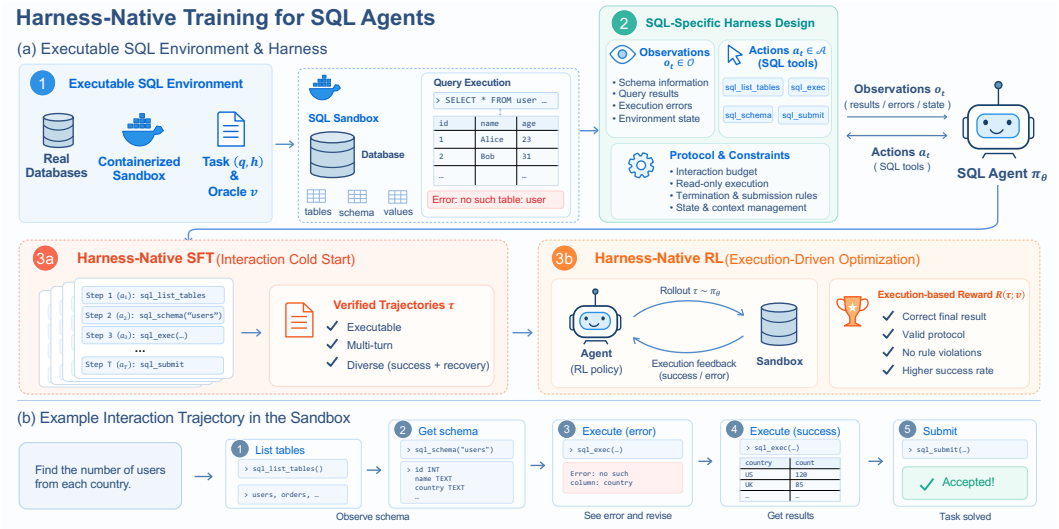

Figure 2: Overview of HARNESSSQL, our harness-native post-training framework for interactive SQL agents. (a) Databases and hidden execution oracles are encapsulated into isolated, containerized sandboxes, where a SQL-specific harness mediates agent–environment interaction through four actions: table discovery (`sql_list_tables`), schema inspection (`sql_schema`), query execution (`sql_exec`), and submission (`sql_submit`). The harness exposes schemas, query results, and execution errors while enforcing interaction and execution constraints. Harness-native SFT provides an interaction cold start from verified multi-turn trajectories, after which harness-native RL further optimizes the policy through online sandbox interaction and execution-based rewards. (b) An example trajectory illustrates the resulting workflow: the agent discovers tables, inspects the schema, executes a query, observes an execution error, revises its solution, and submits the verified result.

3.1 EXECUTABLE SQL ENVIRONMENT CONSTRUCTION

Catalog grounding and task synthesis. For each target database, we extract an execution-grounded catalog directly from the live engine, capturing runtime schemas, sample column values, and valid join paths. Rather than pairing questions with post-hoc queries, task construction follows an *SQL-first* blueprinting approach: the generator samples an executable SQL query adhering to difficulty controls (Appendix I) and derives both the natural-language prompt and the execution oracle from the ground-truth relation.

Hermetic sandboxing and deterministic gating. Each generated task packages public instruction fields (q, h) and a hidden oracle verifier v. Tasks are instantiated inside isolated container sandboxes enforcing strict read-only execution, single-statement dialect constraints, and child-process deadlines (60s). Candidate blueprints pass through automated deterministic gates that prune syntax errors, empty relations, degenerate joins, and uninformative edge cases. Mutation tests further perturb queries to ensure semantic discriminability before rollout.

3.2 SQL-SPECIFIC HARNESS ADAPTATION

Rather than exposing an unbounded coding environment, we specialize the DSH substrate (DeepSeek AI, 2026) to define a minimal, database-aligned state–action contract:

Action space ($\mathcal{A}$): The model-visible tool interface is restricted to four primitives: `sql_list_tables` (table and row enumeration), `sql_schema` (DDL inspection and sample values), `sql_exec` (bounded read-only query trial), and `sql_submit` (terminal artifact submission). Unrelated shell, filesystem, and code-runtime tools are disabled. **Observation and protocol middleware ($\mathcal{O}$):** The environment plane injects runtime guards around every turn, including deterministic row truncation to bound observation length, repeated-call reminders to mitigate loop deadlocks, and pressure-triggered context compaction for long-horizon episodes. Detailed lifecycle hooks are elaborated in Appendix A.

3.3 HARNESS-NATIVE POST-TRAINING

Supervised fine-tuning (cold start). We collect demonstration experience by deploying an expert teacher directly within the target harness. For each task, the teacher interacts with the live database solely through the four-tool contract without access to the hidden oracle. Trajectories that pass denotational verification and protocol checks are preserved, yielding 2,512 verified episodes across 79 databases. We train the student policy via full-trajectory SFT:

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\sum_{t=1}^{T} \log \pi_\theta(a_t \mid s_t), \quad \text{where } s_t = (q, a_1, o_1, \ldots, a_{t-1}, o_{t-1}). \tag{6}$$

Crucially, tokens originating from the harness observations o_t are masked out of the loss, training the model to orchestrate reasoning, tool invocation, and error recovery conditioned on runtime feedback.

Reinforcement learning (execution optimization). To encourage the agent to discover autonomous exploration and self-correction paths beyond teacher demonstrations, we continue training via reinforcement learning in the identical harness environment. We optimize the policy using Decoupled Clip and Dynamic Sampling Policy Optimization (DAPO) (Yu et al., 2025). For each task rollout, the hidden verifier evaluates the submitted artifact against the oracle, providing a sparse, execution-based terminal reward $R(\tau; x) \in \{0, 1\}$. Token-level policy gradients are applied across all generated assistant segments. By preserving identical tool semantics, observation boundaries, and termination protocols across both SFT and RL, the policy natively masters the closed-loop decision process required at deployment. Complete training objectives and hyperparameters are detailed in Appendix C.

4 EXPERIMENTAL SETUP

Corpus and Data Synthesis. We construct our training corpora from 79 databases originating from Spider 2.0-Lite (30 SQLite) and Spider 2.0-DBT (49 DuckDB ported to SQLite with full schema and column parity). For SFT, collecting verified teacher rollouts yields 2,512 multi-turn trajectories. For reinforcement learning, our training pool comprises 2,700 synthesized question–SQL prompts and 100 validation instances. Detailed database mappings, difficulty stratification, and corpus statistics are provided in Appendix D.

Harness Deployment. Evaluations and rollouts strictly follow the 4-tool DSH-SQL contract defined in Section 3.2. Each task runs in a fresh, isolated child process with read-only database bindings, a 60-second execution deadline, and bounded observation buffers. Tasks terminating without an explicit `sql_submit` call are marked as failures. Full runtime architecture, middleware hooks, and activation logs are detailed in Appendix A.

Model Training. We evaluate HARNESSSQL across two compact base models: Qwen3-8B and Qwen3-14B. Models are trained sequentially via harness-native full-trajectory SFT and execution-reward RL using DAPO. Optimization is conducted over isolated DSH environments. Specific training hyperparameters, sequence length caps, and compute configurations are deferred to Appendix C.

Evaluation Benchmarks. We evaluate execution accuracy under both in-domain and out-of-distribution (OOD) settings: (i) **Spider 2.0-SQLite** (135 complex tasks from Spider 2.0-Lite) as the in-domain benchmark; (ii) **BIRD-Interact Mini** (Huo et al., 2025) (under the A-Interact interactive protocol with a GPT-4o user simulator) for multi-turn user/environment interaction transfer; and (iii) **LiveSQLBench Base-Lite SQLite** (BIRD Team, 2026) for cross-database and complex reasoning generalization. Comprehensive benchmark specifications are given in Appendix G.

Baseline Models and Evaluation Protocols. We compare against three groups of baselines in Table 1. **General-purpose models** include Qwen2.5-Coder-32B, providing coding-oriented and reasoning-capable references. **SQL-specialized models** include OmniSQL(Li et al., 2025) and Arctic-Text2SQL(Yao et al., 2026b) at both 7B and 32B scales. **Multi-agent systems** are represented by MARS-SQL(Yang et al., 2025), which coordinates three 7B models for SQL generation. To ensure fair comparison, all baselines are evaluated using their native inference protocols, avoiding performance degradation from forced harness interactions (see Appendix F for cross-harness ablations).

Table 1: Execution accuracy across in-domain (S2-SQLite) and out-of-distribution (BI-mini, LSB-SQLite) benchmarks (S2-SQLite: Spider 2.0-SQLite; BI-mini: BIRD-Interact Mini; LSB-SQLite: LiveSQLBench Base-Lite SQLite). **Bold** indicates the best, and underline indicates the second best.

Method	Model Size	Agentic?	Reasoning?	Harness-trained?	S2-SQLite	BI-mini	LSB-SQLite
General-purpose models							
Qwen2.5-Coder	32B	✗	✓	✗	16.3%	3.0%	7.4%
SQL-specialized models							
OmniSQL	7B	✗	✗	✗	13.3%	1.7%	7.0%
OmniSQL	32B	✗	✗	✗	14.8%	2.7%	8.9%
Arctic-Text2SQL	7B	✗	✓	✗	15.6%	2.3%	7.0%
Arctic-Text2SQL	32B	✗	✓	✗	16.3%	–	–
Multi-agent systems							
MARS-SQL	3×7B	✓	✓	✗	12.6%	3.0%	7.4%
HARNESSSQL	8B	✓	✓	✓	45.2%	9.0%	24.8%
	14B	✓	✓	✓	**54.8%**	**11.3%**	**28.5%**

5 EXPERIMENTAL RESULTS

5.1 MAIN RESULTS

Table 1 compares our final harness-native agents against existing SQL baselines across in-domain (ID) and out-of-distribution (OOD) benchmarks. Along with execution accuracy, we report model size, agentic interaction, explicit reasoning, and whether each model is trained within an interactive harness, making the differences in model scale and interaction paradigm explicit.

In-Domain Performance. On Spider 2.0-SQLite, HARNESSSQL establishes a substantial advantage over both traditional non-agentic baselines and prior multi-agent setups. Our 8B variant achieves 45.2% execution accuracy, nearly tripling the performance of 32B-scale specialized models such as OmniSQL (14.8%), Arctic-Text2SQL (16.3%), and Qwen2.5-Coder-32B (16.3%). Scaling the policy to 14B yields further consistent gains, reaching 54.8%. This highlights that compact models, when natively grounded in the execution harness, can vastly outperform far larger open-loop models that lack trajectory-level interaction capabilities.

Out-of-Distribution Generalization. The performance gains transfer robustly to unseen environments and distinct interaction protocols. On BI-mini, which requires multi-turn dialogue with a user simulator, HARNESSSQL reaches 9.0% (8B) and 11.3% (14B), well above the 1.7%–3.0% range achieved by existing 7B–32B baselines and the MARS-SQL multi-agent pipeline (3.0%). Crucially, on LSB-SQLite, our 8B and 14B models achieve 24.8% and 28.5% execution accuracy, respectively, outperforming all evaluated 32B baselines by a large margin (7.0%–8.9%). These trends confirm that harness-native post-training instills resilient procedural reasoning and closed-loop error-recovery capabilities rather than simply memorizing in-domain schema patterns.

5.2 EFFECT OF POST-TRAINING

Table 2 reports the performance progression across different post-training stages on Spider 2.0-SQLite, starting from the original Qwen3 base models.

Trajectory Supervision vs. Direct RL. Supervised fine-tuning (SFT) substantially elevates execution accuracy, rising from 15.5% to 30.4% for the 8B model and from 22.2% to 37.8% for the 14B model (+14.9 and +15.6 percentage points, respectively). In contrast, applying execution-reward RL directly to the 8B base model achieves only 20.0% (+4.5 points). This demonstrates that while uninitialized RL provides modest exploration gains, cold-starting via verified harness trajectories is critical for establishing effective task grounding.

Table 2: Effect of harness-native post-training on Spider 2.0-SQLite execution accuracy.

Training Stage	8B	14B
Base	15.5%	22.2%
Direct RL	20.0%	-
SFT	30.4%	37.8%
SFT + RL	**45.2%**	**54.8%**

Execution-Reward RL after SFT. Continuing from the SFT checkpoints, RL yields a further substantial leap to 45.2% (+14.8 points) at 8B and 54.8% (+17.0 points) at 14B. The complete two-stage pipeline delivers total gains of 29.7 and 32.6 points over the respective base models. Notably, RL post-SFT yields over $3\times$ the accuracy increment compared to Direct RL on the 8B model, highlighting that trajectory supervision provides the structural foundation necessary for reinforcement learning to efficiently optimize complex execution-guided reasoning.

5.3 Ablation Studies

We next examine the design choices most closely tied to our formulation: alignment between post-training and the deployment harness, preservation of complete interaction trajectories, and sufficient context for long-horizon interaction. We additionally test the robustness of our results to the RL optimization algorithm. Further analyses of trajectory synthesis and learned interaction behavior are provided in Appendix B.

Table 3: Comparison of non-interactive vs. harness-native training and inference across ID and OOD benchmarks.

Post-training	Inference	S2-SQLite (ID)	BI-mini (OOD)	LSB-SQLite (OOD)
None	Non-harness	2.2%	1.0%	3.3%
None	DSH-SQL	15.5%	3.3%	12.2%
Non-harness	Non-harness	14.1%	1.3%	3.7%
Non-harness	DSH-SQL	9.6%	3.0%	8.2%
Harness-native	DSH-SQL	30.4%	5.3%	16.6%

Harness-native training and inference. We first ablate the core premise of our framework: aligning post-training with the interactive execution harness. Table 3 separates the effects of interactive inference and harness-native training.

First, an executable interactive environment is already valuable without post-training: applying DSH-SQL to the base model improves S2-SQLite from 2.2% to 15.5%, with consistent gains on both OOD benchmarks. Second, with inference fixed to DSH-SQL, harness-native post-training substantially outperforms non-harness post-training across all three benchmarks, including 30.4% vs. 9.6% on S2-SQLite. Finally, non-harness post-training improves non-harness inference (2.2% to 14.1%) but transfers poorly to DSH-SQL (9.6%), demonstrating the importance of training–inference alignment. Additional comparisons between the generic Full DSH interface and our SQL-specific harness are reported in Appendix B.1.

SFT trajectory organization. We next investigate whether multi-turn interactions should be trained as unified long sequences or decomposed into turn-level instances. In our default *full-trajectory* setting, an entire episode is trained as a single sequence where all assistant actions are jointly supervised while environment observations are masked from the loss. In contrast, the *turn-split* variant decomposes each episode into separate training samples, optimizing the loss solely on each individual assistant turn conditioned on its preceding history prefix.

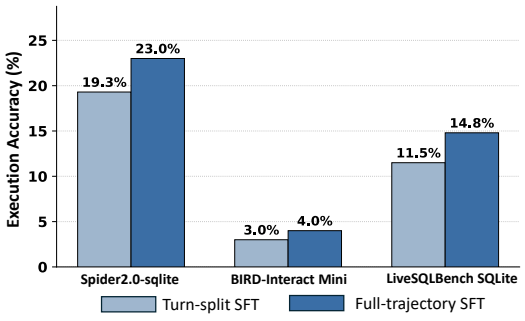

Figure 3: Effect of multi-turn trajectory organization for SFT.

As shown in Figure 3, full-trajectory SFT consistently outperforms turn-split training across all three benchmarks. This result supports our central supervision hypothesis: preserving complete interaction structure provides a stronger learning signal for long-horizon database agents than decomposing trajectories into isolated decisions.

Trajectory context length. Long-horizon SQL agents impose substantially greater context demands than conventional Text-to-SQL systems, as a single trajectory must accommodate reasoning traces, tool calls, execution outputs, intermediate observations, and iterative revisions. We therefore vary the maximum sequence length while keeping the context limit consistent across SFT and RL. As shown in Figure 4, performance improves consistently as the context window increases from 8K to 32K across both ID and OOD benchmarks. The improvement is particularly pronounced on S2-SQLite, where execution accuracy increases from 27.4% at 8K to 45.2% at 32K, while BI-mini and LSB-SQLite exhibit similar upward trends. These results suggest that sufficient context capacity is important for learning long-horizon agentic trajectories, where truncating earlier interactions can remove execution evidence and intermediate reasoning required for subsequent decisions.

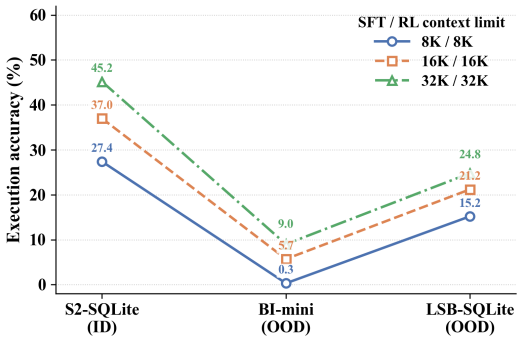

Figure 4: Effect of the maximum trajectory context length. The same context limit is used during both SFT and RL.

RL algorithm. Our main results use a fixed RL recipe. To determine whether the gains depend strongly on the specific optimization algorithm, we compare GRPO (Shao et al., 2024), GSPO (Zheng et al., 2025), and DAPO (Yu et al., 2025) starting from the same SFT checkpoint and using identical rollout data, reward signals, context limits, and training budgets.

As shown in Table 4, both DAPO and GSPO consistently outperform GRPO across both in-domain and out-of-distribution suites, with DAPO achieving the highest accuracy on Spider2.0-SQLite (45.2% vs. 40.7%). This improvement is consistent with the more stable policy updates of DAPO and GSPO, which may be particularly beneficial for long-horizon trajectories with sparse execution rewards. The relatively small difference between DAPO and GSPO further suggests that the gains of HARNESSSQL are not tied to a single RL algorithm.

Table 4: Comparison of RL algorithms under matched training and rollout budgets.

RL Algorithm	S2-SQLite	BI-mini	LSB-SQLite
GRPO	40.7%	8.7%	23.3%
GSPO	43.7%	9.0%	**25.2%**
DAPO	**45.2%**	**9.0%**	24.8%

Beyond execution accuracy, trajectory-level analysis shows that harness-native post-training substantially improves protocol adherence, error recovery, and valid termination, providing behavioral evidence that the model learns the intended interaction policy rather than only improving terminal accuracy (Appendix B.2).

6 RELATED WORK

LLMs for Text-to-SQL. Text-to-SQL has progressed from cross-domain semantic parsing benchmarks such as Spider (Yu et al., 2018) and BIRD (Li et al., 2023) to more realistic settings involving large schemas, external knowledge, and enterprise database workflows, as exemplified by Spider2.0(Lei et al., 2025) and BIRD-Interact (Huo et al., 2025). Recent work improves LLM-based text-to-SQL through large-scale synthetic supervision and execution-based optimization, including OmniSQL and Arctic-Text2SQL-R1 (Li et al., 2025; Yao et al., 2026b), while systems such as CHESS (Talaei et al., 2024), ReFoRCE (Deng et al., 2025), and MARS-SQL (Yang et al., 2025) increasingly rely on retrieval, database exploration, execution feedback, and iterative refinement at inference time. Together, these developments suggest that performance on realistic SQL tasks depends not only on SQL generation ability, but also on effective interaction with the surrounding execution environment. Rather than proposing another SQL-specific inference strategy, HARNESSSQL uses SQL as a challenging domain for studying how training data can be synthesized to teach models to operate effectively through an agent harness.

Harnesses and tool-using agents. Modern language-model agents operate through harnesses that define their tools, observations, execution loops, and context management. ReAct (Yao et al., 2023) established the paradigm of interleaving reasoning and actions, while subsequent systems study richer interfaces for API use, executable code, and software environments (Qin et al., 2023; Wang et al., 2024; Yang et al., 2024). Interactive benchmarks further expose the execution environment as part of the problem (Liu et al., 2023; Yao et al., 2024); most directly, Harness-Bench shows substantial performance variation across model–harness pairings even under controlled tasks and validators (Yao et al., 2026a). These findings motivate treating agent behavior as a property of the model–harness system rather than the backbone alone. HARNESSSQL extends this perspective from evaluation to learning by making the deployment harness part of task generation, teacher rollout, post-training, and evaluation.

Synthetic instruction and agent trajectory data. Synthetic supervision has evolved from bootstrapped instruction–response pairs (Wang et al., 2023) and self-supervised tool-use annotations (Schick et al., 2023) to multi-step tool-use data (Tang et al., 2023; Qin et al., 2023) and complete agent trajectories (Chen et al., 2023; Zeng et al., 2023; Mitra et al., 2024). Recent work also emphasizes executable and semantic verification to improve the reliability of synthetic tool-use data (Liu et al., 2024b;a). HARNESSSQL combines these directions through *harness-native* synthesis: tasks are grounded in executable environments, solved by teachers through the target harness, and retained only after outcome verification. Task construction, trajectory generation, and verification therefore share the deployment protocol, yielding supervision for the complete harness-conditioned interaction policy rather than isolated answers or tool calls.

7 LIMITATIONS AND ONGOING WORK

Our current study focuses primarily on SQLite, while several realistic SQL benchmarks additionally involve dialects such as PostgreSQL, Snowflake, and BigQuery. This limitation is partly inherited from the underlying small models, whose initial capabilities are substantially weaker and less consistent across non-SQLite dialects. Prior work has similarly identified SQL dialect generalization as a major bottleneck for smaller text-to-SQL models, and has shown that execution-verified adaptation can substantially improve performance on previously underrepresented dialects (Zhang et al., 2025). In our setting, this challenge is further compounded by long-horizon interaction, where dialect-specific syntax, database clients, metadata commands, and execution behavior all become part of the agent trajectory. Our ongoing work therefore extends harness-native synthesis to dialect-specific execution environments, with the goal of closing the current performance gap between SQLite and other practically important SQL dialects.

8 CONCLUSION

We presented HARNESSSQL, a harness-native framework for training long-horizon SQL agents. Our central finding is that closing the train–deploy gap requires alignment at three levels: an executable environment that exposes authentic database states and runtime feedback, a task-specific harness that defines the agent's interaction protocol, and trajectory-level supervision that teaches the resulting closed-loop policy. HARNESSSQL realizes these principles through isolated SQL sandboxes with hidden executable oracles, a minimal four-tool SQL harness, and harness-native full-trajectory SFT followed by execution-reward RL.

Across Spider 2.0-SQLite, BIRD-Interact Mini, and LiveSQLBench Base-Lite SQLite, harness-native post-training consistently improves both tested model scales and transfers to out-of-distribution interactive settings. On Spider 2.0-SQLite, Qwen3-8B improves from 15.5% to 45.2% and Qwen3-14B from 22.2% to 54.8%. Controlled analyses further show that these gains depend not only on access to an interactive execution environment, but also on aligning training with the deployment harness and preserving complete interaction trajectories during supervision. These results support a broader view of agent learning: the harness is not merely an inference-time wrapper, but part of the behavior distribution that training should reproduce. While broader dialect and harness coverage remains an important direction, our results provide a concrete path toward compact database agents that learn verified interaction policies rather than isolated terminal answers.

Reproducibility Statement

Appendix A specifies the harness composition and runtime hooks, Appendix C specifies the post-training objective and context budgets, Appendix H specifies the task and trajectory records, and Appendix J lists the verifier gates used in the current implementation. We record generation prompts, database identifiers, random seeds, raw rollouts, executed SQL, normalized result fingerprints, per-seed outcomes, selected-trajectory provenance, timing, and rejection reasons. The final submission will include an anonymized code snapshot, data manifests, training hyperparameters, environment versions, and evaluation commands; proprietary database contents and model endpoints will be replaced by documented public equivalents where redistribution is not permitted.

AI use statement

We used an AI assistant for grammar checking and translation during manuscript preparation. For grammar checking, we used the assistant to identify grammatical errors and suggest corrections to sentence structure, tense, and word usage. For translation, we used it to translate text, with attention to fluency, consistent technical terminology, and preservation of the original meaning.

References

BIRD Team. LiveSQLBench-CLI: Benchmarking command-line agents on postgresql-backed text-to-SQL tasks. `https://github.com/bird-bench/livesqlbench/tree/main/LiveSQLBench-CLI`, 2026. Open-source benchmark repository; accessed 2026-09-02.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. FireAct: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*, 2023.

DeepSeek AI. Deepseek harness: Everything is a plugin. `https://github.com/deepseek-ai/deepseek-harness`, 2026. Open-source software repository; accessed 2026-09-02.

Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. ReFoRCE: A text-to-SQL agent with self-refinement, format restriction, and column exploration. *arXiv preprint arXiv:2502.00675*, 2025.

Taicheng Guo, Hai Wang, Chaochun Liu, Mohsen Golalikhani, Xin Chen, Xiangliang Zhang, and Chandan K. Reddy. MTSQL-r1: Towards long-horizon multi-turn text-to-SQL via agentic training. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 33905–33938, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.1563. URL `https://aclanthology.org/2026.acl-long.1563/`.

Nan Huo, Xiaohan Xu, Jinyang Li, Per Jacobsson, Shipei Lin, Bowen Qin, Binyuan Hui, Xiaolong Li, Ge Qu, Shuzheng Si, et al. Bird-interact: Re-imagining text-to-sql evaluation for large language models via lens of dynamic interactions. *arXiv preprint arXiv:2510.05318*, 2025.

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

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

Jinyang Li, Ziyong Wang, Xuanhe Zhou, Zujie Huang, Jun Zhao, Jiaxi Jiang, Bing Wang, Xuan Zhou, Shimin Chen, Jianwen Su, Bin Cui, and Guoliang Li. Omnisql: Synthesizing high-quality text-to-sql data at scale. *Proceedings of the VLDB Endowment*, 18(1):4695–4708, 2025. doi: 10.14778/3749646.3749723.

Jinyang Li, Xiaolong Li, Ge Qu, Per Jacobsson, Bowen Qin, Binyuan Hui, Shuzheng Si, Nan Huo, Xiaohan Xu, Yue Zhang, Ziwei Tang, Yuanshuai Li, Florensia Widjaja, Xintong Zhu, Feige Zhou, Yongfeng Huang, Yannis Papakonstantinou, Fatma Ozcan, Chenhao Ma, and Reynold Cheng. Swe-sql: Illuminating llm pathways to solve user sql issues in real-world applications, 2026. URL https://arxiv.org/abs/2506.18951.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, et al. ToolACE: Winning the points of LLM function calling. *arXiv preprint arXiv:2409.00920*, 2024a.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. AgentBench: Evaluating LLMs as agents. *arXiv preprint arXiv:2308.03688*, 2023.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, et al. APIGen: Automated pipeline for generating verifiable and diverse function-calling datasets. *arXiv preprint arXiv:2406.18518*, 2024b.

Arindam Mitra, Luciano Del Corro, Guoqing Zheng, Shweti Mahajan, Dany Rouhana, Andres Codas, Yadong Lu, Wei-ge Chen, Olga Vrousgos, Corby Rosset, et al. AgentInstruct: Toward generative teaching with agentic flows. *arXiv preprint arXiv:2407.03502*, 2024.

Quang Hieu Pham, Yang He, Ping Nie, Canwen Xu, Davood Rafiei, Yuepeng Wang, Xi Ye, and Jocelyn Qiaochu Chen. Flexsql: Flexible exploration and execution make better text-to-sql agents, 2026. URL https://arxiv.org/abs/2605.02815.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. *arXiv preprint arXiv:2307.16789*, 2023.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, 2023.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

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

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. ToolAlpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*, 2023.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. *arXiv preprint arXiv:2402.01030*, 2024.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, 2023.

Zekun Xu, Siyu Xia, Chuhuai Yue, Jiajun Chai, Mingxue Tian, Xiaohan Wang, Wei Lin, Haoxuan Li, and Guojun Yin. Mtir-sql: Multi-turn tool-integrated reasoning reinforcement learning for text-to-sql, 2025. URL https://arxiv.org/abs/2510.25510.

Haolin Yang, Jipeng Zhang, Zhitao He, and Yi R. Fung. Mars-sql: A multi-agent reinforcement learning framework for text-to-sql, 2025. URL https://arxiv.org/abs/2511.01008.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool–agent–user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Yilun Yao, Xinyu Tan, Chao-Hsuan Liu, Yaoming Li, Zhengyang Wang, Wenhan Yu, Zhewen Tan, Yuxuan Tian, Guangxiang Zhao, Lin Sun, Xiangzheng Zhang, and Tong Yang. Harness-Bench: Measuring harness effects across models in realistic agent workflows. *arXiv preprint arXiv:2605.27922*, 2026a.

Zhewei Yao, Guoheng Sun, Łukasz Borchmann, Zheyu Shen, Minghang Deng, Bohan Zhai, Hao Zhang, Ang Li, and Yuxiong He. Arctic-text2sql-r1: Simple rewards, strong reasoning in text-to-SQL. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 26966–26995, San Diego, California, United States, July 2026b. Association for Computational Linguistics. URL `https://aclanthology.org/2026.findings-acl.1345/`.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, et al. DAPO: An open-source LLM reinforcement learning system at scale. In *Advances in Neural Information Processing Systems*, 2025.

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. 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*, 2018.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. AgentTuning: Enabling generalized agent abilities for LLMs. *arXiv preprint arXiv:2310.12823*, 2023.

Jipeng Zhang, Haolin Yang, Kehao Miao, Ruiyuan Zhang, Renjie Pi, Jiahui Gao, and Xiaofang Zhou. Exesql: Self-taught text-to-sql models with execution-driven bootstrapping for sql dialects. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 24305–24326, 2025.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025.

A DSH-SQL Harness Composition and Lifecycle

System boundary. DSH-SQL is a profile-level composition, not a replacement agent loop. Table 5 separates inherited DSH services, our SQL-specific additions, and orchestration that runs outside DSH. This distinction is important for attribution: our contribution is the SQL profile, environment adapter, and harness-aligned data and evaluation pipeline; the generic plugin lifecycle, agent loop, and session substrate come from DSH.

Table 5: Layers of the evaluation harness. "Visible" indicates whether the model can invoke or directly observe the component, rather than whether its effects can influence the trajectory.

Layer	Ownership	Responsibility	Visible
Experiment controller	Ours, outside DSH	Loads tasks; creates a process, workspace, and DSH home per task; sets environment bindings; enforces episode deadlines; schedules workers; recovers submissions; records all outcomes.	No
Profile composition	DSH + ours	Composes `dsh-base`, `dsh-headless`, and `dsh-bundle-sql`; disables unrelated coding-agent capabilities; mounts the selected model and database backend.	No
Control plane	DSH	Prompt and model routing, agent/session state, durable checkpoints, cancellation, retry, compaction, result retention, and headless termination.	Indirect
ReAct-style loop	DSH	Alternates model requests with validated tool dispatch until a terminal response, maximum-token stop, error, or cancellation.	Indirect
SQL environment plane	Ours	Exposes backend-specific tools; validates SQL; executes queries; formats bounded observations; implements budget, phase, and submission semantics.	Yes
Executable verifier	Ours, outside policy context	Re-executes the submitted artifact, compares normalized results with the hidden oracle, and checks protocol predicates.	No

Lifecycle and middleware. Cordis provides scoped plugin lifecycle effects and ordered event waterfalls. The former install and remove services and listeners safely; the latter act as middleware around an agent step or tool execution. In simplified order, one step is

$$\texttt{system-prompt/assemble} \to \texttt{agent/pre-step} \to \texttt{agent/request} \downarrow \texttt{llm/stream} \to \texttt{tools/pre-execute} \to \texttt{tools/execute} \downarrow \text{SQL tool body} \to \texttt{tools/post-execute} \to \texttt{tools/result}.$$

The pre-step, model-request, and top-level tool boundaries flush the session before the next external action. The request-error branch supports provider retry and context-overflow recovery. Pre-execution middleware can deny or request approval; execution middleware propagates cancellation and optional tool deadlines; post-execution middleware can bound large outputs and inject corrective context. Every durable event—including prompt headers, token usage, calls, results, compactions, and turn termination—is written to the task's session log.

Composition does not imply activation. The inspected SQLite profile contains 82 registrations: 40 are mounted and 42 are disabled. The disabled set includes model-facing shell, filesystem, web, editor, code-runtime, skill, goal, plan, subagent, and workflow capabilities. Among the mounted set, some services are conditional middleware rather than actions that fire in every episode. Table 6 reports an implementation audit over one complete 135-task evaluation directory. The directory contains 138 session logs because three tasks were attempted more than once; counts below are over logs rather than unique benchmark items.

Table 6: Runtime activation audit for the SQLite profile. A mounted hook may be traversed without its conditional action firing.

Mechanism	Observed	Interpretation
Session policy initialization	138	Permission preset, sandbox mode, and approval policy were recorded once per session.
`sql_list_tables` / `sql_schema`	153 / 199	Both tools appeared in every audited session.
`sql_exec` / `sql_submit`	3,230 / 129	All sessions explored with execution; 129 reached explicit submission.
Automatic context compaction	22	The pressure-triggered summarization branch fired only on long trajectories.
Tool-result pruning	1	One over-threshold observation was deterministically shortened.
Repeated-call reminders	19	Middleware injected feedback after repeated identical calls.
SQLite query deadline	35	The adapter killed child queries exceeding 60 seconds.
Multi-statement guard	2	The SQL adapter rejected calls containing more than one statement.
LLM retry / output spill	0 / 0	Both plugins were mounted, but their trigger conditions did not occur.
Generic DSH tool deadline	0	SQL tools use the adapter's child-process deadline rather than a tool-definition deadline.

Other mounted base services—for example settings, credentials, attachments, jobs, session querying, and generic shell/sandbox infrastructure—satisfy composition dependencies or initialize ambient state but are not model-callable and are not on the semantic SQL path. We therefore do not count the 40 mounted registrations as 40 agent capabilities.

Backend-specific environment planes. The SQL plugin selects one backend per process. SQLite exposes the four tools used for synthesis and Spider evaluation. It checks the single-statement read-only grammar, opens the database read-only, and executes each query in a separately killable child process. PostgreSQL and BigQuery adapters share the four-tool abstraction while implementing dialect-specific execution and scan constraints; they are not used by the reported SQLite results. The BIRD-Interact profile instead exposes the benchmark's nine operations: query execution, schema access, column semantics, three external-knowledge lookups, user clarification, and submission. A wrapper charges the configured bird-coin cost, persists actions and remaining budget, blocks further exploration after exhaustion, and transitions a successful phase-1 submission to the phase-2 user follow-up. The `ask_user` operation forwards one question to the GPT-4o user-simulator service. Thus the BIRD experiment preserves DSH's control plane while intentionally changing both the backend and the model-visible capability contract.

B Additional Ablations and Analysis

B.1 Trajectory Synthesis Configuration

We ablate two core design choices in trajectory synthesis: the strength of the teacher model and the specialization of the execution harness. Using Qwen3-8B fine-tuned on a fixed 1000-example synthesized subset, Table 7 reveals two critical insights: First, **teacher capability directly dictates**

Table 7: Ablation on trajectory synthesis configurations evaluated on Spider 2.0-SQLite, using Qwen3-8B fine-tuned on 1,000 synthesized trajectories.

Teacher Model	Harness	S2-SQLite
Base Model	Full DSH	13.3%
Qwen3.5-9B	DSH-SQL	7.4%
Qwen3.6-27B	Full DSH	15.5%
Qwen3.6-27B	DSH-SQL	23.0%

downstream policy quality. Weak supervision from Qwen3.5-9B degrades performance below even the un-finetuned base model, confirming that suboptimal trajectories and noisy exploration heavily mislead student learning. High-capacity teachers are therefore indispensable for viable trajectory synthesis.

Second, **specialized harnesses facilitate both inference and learning for small size models.** Compared to the general-purpose Full DSH, our streamlined DSH-SQL yields superior performance across the board—not only boosting zero-shot base inference (15.5% vs. 13.3%, as shown previously) but also maximizing post-training gains. Narrowing the action space and reducing interface noise make demonstrations significantly easier for compact models to absorb effectively.

B.2 Interaction Behavior Analysis

Execution accuracy alone does not reveal whether harness-native post-training actually changes how the model interacts with the database environment. We therefore analyze the complete interaction trajectories of the base, SFT, and SFT+RL models. For each checkpoint, we perform four independent rollouts on all 135 Spider 2.0-SQLite tasks, yielding 540 trajectories per model. Harness-native post-training substantially changes the model's interaction behavior, not only its final execution accuracy. SFT produces the largest improvement in basic protocol adherence: the fraction of trajectories containing invalid calls falls from 67.4% to 17.4%, strict protocol violations decrease from 83.2% to 26.7%, and duplicate calls drop from 58.8% to 16.3%. At the same time, the model becomes substantially more likely to execute its final SQL before submission and to recover immediately from execution errors.

DAPO further consolidates these behaviors. Invalid calls become almost eliminated (0.6%), exactly-one-valid-submit trajectories increase from 75.7% after SFT to 93.5%, and strict protocol violations fall to 11.5%. In contrast, the reduction in SQL execution errors is more modest (9.0% to 8.1%). This suggests a complementary role for the two stages: trajectory SFT teaches the interaction protocol and common recovery patterns, while execution-reward RL primarily improves episode-level behavioral consistency and successful termination.

C Post-training Configuration and Objectives

C.1 Training configuration

Tables 9 and 10 report the launch configurations of the checkpoints used in our main results. The two model chains are not identical: Qwen3-8B and Qwen3-14B RL are initialized from the 32K SFT checkpoint. Both RL runs allow a 32,768-token DSH session, with at most 4,096 newly generated tokens in any single model call. The trainer's token cap controls dynamic batching for memory use and is not an additional model context-window limit.

C.2 Policy optimization objectives

GRPO objective. For prompt q, let $\{\tau_i\}_{i=1}^{G}$ be $G = 8$ trajectories with scalar rewards r_i. The implementation uses the group-standardized advantage

$$\hat{A}_i = \frac{r_i - \mathrm{mean}_{j\in[G]}\, r_j}{\mathrm{std}_{j\in[G]}\, r_j + 10^{-6}}, \tag{7}$$

and broadcasts $\hat{A}_i$ to the loss-bearing assistant tokens in trajectory i. With token-level importance ratio $\rho_{i,t} = \pi_\theta(a_{i,t} \mid s_{i,t})/\pi_{\mathrm{old}}(a_{i,t} \mid s_{i,t})$, we minimize the clipped policy loss

$$\mathcal{L}_{\mathrm{GRPO}}(\theta) = -\mathbb{E}_{i,t}\Big[\min\Big(\rho_{i,t}\hat{A}_i, \mathrm{clip}(\rho_{i,t}, 1-\epsilon, 1+\epsilon)\hat{A}_i\Big)\Big], \qquad \epsilon = 0.2. \tag{8}$$

The number of optimizer steps and effective global batch size for each checkpoint are reported in Table 10. Both runs use no entropy bonus. The launch configuration sets the auxiliary KL-loss coefficient to zero. It also passes a reference-KL bookkeeping coefficient of 0.01; in the inspected Slime revision, the standard GRPO return broadcaster does not subtract that computed KL tensor. Consequently, the effective objective for these checkpoints is the clipped, group-normalized execution-reward objective above, without a KL term. Failed harness processes are removed groupwise before optimization rather than treated as negative demonstrations.

Table 8: Interaction behavior across post-training stages on Spider 2.0-SQLite.

Metric	Base	SFT	SFT+DAPO
Trajectories with invalid calls ↓	67.41%	17.41%	**0.56%**
SQL execution-error rate ↓	17.93%	9.03%	**8.08%**
Immediate recovery ↑	44.09%	67.80%	**72.72%**
Final SQL executed before submit ↑	47.78%	70.56%	**72.04%**
Exactly one valid successful submit ↑	18.70%	75.74%	**93.52%**
Strict protocol violation ↓	83.15%	26.67%	**11.48%**
Duplicate-call rate ↓	58.82%	16.29%	**6.61%**

Table 9: SFT configurations used for both Qwen3-8B and Qwen3-14B checkpoints. Both models share identical training settings and are initialized from their respective base checkpoints. Each example is one complete verified harness episode.

Hyperparameter / Setting	**Value / Specification**
Base models	Qwen3-8B / Qwen3-14B
Training examples	2,512 verified trajectories
Maximum serialized length	32,768 tokens
Loss mask	Assistant reasoning and actions only; system, user, and environment tokens masked
Optimization horizon	3 epochs (471 optimizer updates)
Batching	Per-GPU batch size 1; gradient accumulation steps 2; effective batch size 16
Optimizer	AdamW; $\mathrm{lr} = 10^{-5}$; cosine decay; 5% warmup; weight decay 0.01
Hardware & parallelization	8× NVIDIA H800 GPUs; bfloat16; gradient checkpointing; DeepSpeed ZeRO-3

GSPO objective. Group Sequence Policy Optimization (GSPO) replaces GRPO's token-level importance weighting and clipping with a single sequence-level decision (Zheng et al., 2025). Let $\mathcal{T}_i$ be the loss-bearing assistant-token positions in trajectory τ_i and $N_i = |\mathcal{T}_i|$. Environment observations remain part of each conditioning state $s_{i,t}$ but do not bear loss. From the token-level ratios above, GSPO constructs the length-normalized trajectory ratio

$$\sigma_i(\theta) = \left(\prod_{t \in \mathcal{T}_i} \rho_{i,t} \right)^{1/N_i} = \exp\left(\frac{1}{N_i} \sum_{t \in \mathcal{T}_i} \log \rho_{i,t} \right). \tag{9}$$

Using the same group-standardized trajectory advantage $\hat{A}_i$, its clipped loss is

$$\mathcal{L}_{\mathrm{GSPO}}(\theta) = -\mathbb{E}_i \Big[\min\Big(\sigma_i(\theta)\hat{A}_i, \mathrm{clip}(\sigma_i(\theta), 1-\epsilon_s, 1+\epsilon_s)\, \hat{A}_i \Big) \Big], \tag{10}$$

where ϵ_s is the sequence-level clipping radius. Thus every assistant token in a trajectory is weighted by the same trajectory ratio, and the entire trajectory is clipped together. The length normalization keeps ratios from trajectories of different lengths on a comparable numerical scale. This makes the unit of importance correction and clipping coincide with the unit of our terminal execution reward, whereas GRPO clips each token ratio separately.

DAPO objective. Decoupled Clip and Dynamic sAmpling Policy Optimization (DAPO) retains token-level importance ratios but modifies group policy optimization in three relevant ways: it uses different lower and upper clipping radii, removes zero-variance reward groups through dynamic sampling, and normalizes the loss across all loss-bearing tokens in the batch rather than first assigning equal weight to every trajectory (Yu et al., 2025). For binary execution rewards, define the effective prompt set

$$\mathcal{D}_{\mathrm{eff}} = \left\{ q : \; 0 < \sum_{i=1}^{G} \mathbb{I}[r_i = 1] < G \right\}. \tag{11}$$

Table 10: DAPO configurations used for the reported Qwen3-8B and Qwen3-14B results. "Prompt pool" is the number of records in the corresponding launch input.

Setting	Configuration (Qwen3-8B / Qwen3-14B)
Initialization	32K Qwen3-8B / 14B SFT checkpoint
Rollout horizon	150 rollout iterations
Rollout group	4 prompts per iteration; 8 trajectories per prompt
Updates and batching	2 optimizer steps per rollout; global batch 16; micro-batch 1
Context limits	32,768 tokens per DSH session; at most 4,096 new tokens per model call
Optimizer	Adam; $\mathrm{lr} = 10^{-6}$; constant schedule; $\beta = (0.9, 0.98)$; weight decay 0
Policy objective	DAPO; asymmetric clipping $\epsilon \in [0.20, 0.28]$; no entropy bonus; no effective KL penalty
Trainer implementation	32,768-token dynamic-batch cap; FP32 main gradients
Systems	4 H800 actor GPUs and 4 H800 rollout GPUs; rollout tensor parallelism 4

Groups outside this set have $\hat{A}_i = 0$ for every trajectory and hence provide no policy-gradient signal. With $N_i = |\mathcal{T}_i|$, the DAPO loss is

$$\mathcal{L}_{\mathrm{DAPO}}(\theta) = -\mathbb{E}_{q\in\mathcal{D}_{\mathrm{eff}}}\left[\frac{1}{\sum_{i=1}^{G} N_i}\sum_{i=1}^{G}\sum_{t\in\mathcal{T}_i}\min\Big(\rho_{i,t}\hat{A}_i,\ \mathrm{clip}(\rho_{i,t}, 1-\epsilon_{\mathrm{low}}, 1+\epsilon_{\mathrm{high}})\,\hat{A}_i\Big)\right], \tag{12}$$

where DAPO typically chooses $\epsilon_{\mathrm{high}} > \epsilon_{\mathrm{low}}$ to permit larger probability increases for positively advantaged, initially unlikely tokens. In the matched ablation, the algorithms receive the same rollout pool before their algorithm-specific reduction or filtering, and DAPO uses the same terminal execution reward and context cutoff as GRPO and GSPO; we do not add a separate overlength shaping reward. This isolates the policy-optimization rule while preserving the common harness interaction budget.

Execution reward. Let $E(y)$ execute the submitted SQL y with a 30-second timeout and return its normalized relation. For tasks whose output ordering is semantically required, normalization preserves row order; otherwise, it compares an order-invariant result fingerprint. The terminal reward is

$$R(\tau) = \begin{cases} 1, & E(y) = E(y^*) \text{ under the task's ordered/unordered comparison,} \\ 1, & \text{values match and the expected aliases are not specified publicly,} \\ 0, & \text{otherwise.} \end{cases} \tag{13}$$

The zero-reward cases include no parsable final SQL, execution errors or timeouts, wrong values, and wrong public column names. There is no partial credit for syntactic validity, number of tool calls, or intermediate SQL. This keeps the RL signal aligned with executable task success while leaving exploration behavior to be learned through credit assignment over the complete DSH trajectory.

D SFT Corpus Statistics

Table 11 summarizes the final SFT corpus after teacher rollout and oracle-based selection. The corpus contains 2,512 successful trajectories collected from 79 database instances across Spider2.0-Lite and Spider2.0-DBT. Each example retains the complete harness interaction rather than only the final SQL query. The resulting trajectories are relatively long, averaging 11.8K tokens and a median of 16 agent turns, with 17.0 tool calls and 13.9 SQL executions per trajectory on average.

The interaction statistics show that successful demonstrations frequently contain non-trivial intermediate behavior. In particular, 77.83% of trajectories contain at least one failed intermediate SQL execution, while 99.92% execute at least two distinct SQL queries. Under the stricter criterion of revising the query after an execution failure, 77.71% of trajectories exhibit such behavior. Thus,

the retained demonstrations capture not only successful final queries but also iterative execution, feedback, and revision within the harness.

Across the retained trajectories, we record 42,704 tool calls and 34,936 SQL executions, of which 3,989 return execution failures. These failures should not be interpreted directly as SQL reasoning errors. Of the 3,989 failed executions, 3,230 arise from queries beginning with SQL comments (`--`), which are rejected by the harness read-only guard because the first parsed token is not `SELECT`, `WITH`, or `EXPLAIN`. We retain these interactions as collected rather than sanitizing them from otherwise successful trajectories, since they reflect the authentic execution feedback observed by the teacher during data collection.

Ablation on Sanitizing Comment-Induced Execution Failures. To verify whether exposing the model to such syntactically benign execution rejections during training introduces harmful artifacts, we perform a controlled ablation study. Specifically, we construct a sanitized training set by pruning all failed interaction turns caused solely by leading comment rejections from the original trajectories, and train an identical model checkpoint using the same training configuration.

When evaluated on the Spider 2.0-Lite SQLite benchmark:

- The model trained on the original, unpruned trajectories achieves an execution accuracy of **30.4%**.
- The model trained on the sanitized trajectories achieves an execution accuracy of **28.2%** (a drop of 2.2 percentage points).

This performance gap demonstrates that retaining benign execution failures along with the subsequent recovery turns provides valuable supervised signal for error recovery and environment self-correction. Sanitizing these interaction cycles reduces the diversity of feedback trajectories, making the model more brittle when encountering real-world runtime rejections from the evaluation harness.

E Data Integrity and Leakage Audit for Shared-Database Evaluation

In our main experiments, the agent is fine-tuned on synthetic tasks derived from the 30 SQLite database instances within Spider 2.0-Lite and evaluated on the official 135 Spider 2.0-Lite SQLite test tasks. This paradigm reflects a realistic enterprise setting where database schemas remain stable while queries vary over time. To ensure that performance gains stem from procedural task execution rather than rote task memorization, we conduct a rigorous multi-tier audit confirming that the evaluation tasks were not present in or leaked into the synthetic training corpus.

Because the Spider 2.0 evaluation benchmark assesses execution outcomes via accepted ground-truth CSV outputs rather than canonical Gold SQL queries, we audit orthogonality across three complementary dimensions: lexical similarity, chunk-level semantic embedding, and database execution results.

E.1 Exact Matching and Near-Duplicate Analysis

We first perform exact matching and lexical overlap checks between the 135 SQLite test queries and both the full synthetic generation pool and the final SFT subset. As summarized in Table 12, across all 30 byte-identical SQLite database instances (verified via SHA-256), there is strictly zero overlap in Task IDs, exact question strings, normalized question strings, and substring containment.

To rule out subtle surface paraphrasing, we measure nearest-neighbor distance over normalized word tokens across all pairs within the same database:

- **Token Sequence Ratio:** Across all 135 test queries, the median sequence ratio to its nearest same-DB synthetic counterpart is 0.0913 (P95: 0.1352; Maximum: **0.1689**).
- **Token-set Jaccard Similarity:** The median Jaccard index is 0.0971 (P95: 0.1600; Maximum: **0.1869**).

Table 11: Statistics of the SFT corpus after teacher rollout and oracle-based selection.

Statistic	Value
Tasks / trajectories	2,512
Database instances	79
Average trajectory tokens	11,793.89
Median agent turns	16
Average tool calls	17.00
Average SQL executions	13.91
Trajectories with ≥ 1 failed intermediate query	77.83%
Trajectories with query revision	99.92%
Trajectories with post-failure query revision	77.71%
Teacher pass@1	71.20%
Teacher pass@3	83.61%

Audit Check	**Full Synthetic Pool vs. Spider2.0**	**Final SFT Subset vs. Spider2.0**
Database Names	30 / 30 Identical	30 / 30 Identical
Database File SHA-256	30 / 30 Byte-identical	30 / 30 Byte-identical
Task ID Intersection	0	0
Exact Question Matches	0	0
Normalized Exact Matches	0	0
Same-DB Question Containment	0	0
Same-DB Execution Matches (Unordered)	0	0
Semantic Embedding Cosine ≥ 0.75	—	0 / 135

Table 12: Comprehensive leakage and overlap audit between synthetic training data and Spider 2.0-Lite SQLite-135 evaluation tasks.

- **Longest Common Contiguous Span:** The median contiguous token span is only 2 tokens (P95: 4; Maximum: **5 tokens**).

None of the test queries remotely approach standard duplicate thresholds (e.g., 0.5 Jaccard or sequence ratio). The maximum contiguous span of 5 tokens reflects shared database entities (e.g., column names and domain terminology) rather than replicated phrasing. Manual inspection of the top nearest pairs (detailed in Table 13) confirms that high-overlap token pairs define fundamentally distinct business objectives and analytical cohorts.

Test ID	**Nearest Synthetic ID**	**Database**	**Seq. Ratio**	**Task Divergence (Manual Verification)**
`local007`	`local_syn_000323`	Baseball	0.1689	**Test:** Lifetime career span across all players. **Train:** Career home-run metrics for the 1990s debut cohort.
`local310`	`local_syn_000250`	F1	0.1688	**Test:** Annual minimal combinations of driver and constructor. **Train:** Constructor points share over 2010–2020.
`local259`	`local_syn_000331`	IPL	0.1425	**Test:** Lifetime batting and bowling profiles per player. **Train:** Venue-level toss-to-field match aggregations.
`local055`	`local_syn_000045`	Chinook	0.1419	**Test:** Customer spending by top vs. bottom sales artists. **Train:** Invoice revenue aggregations by billing country.
`local114`	`local_syn_000369`	Education	0.1412	**Test:** Regional sales-representative performance report. **Train:** FY2021 product-segment revenue breakdown.

Table 13: Manual inspection of top-5 nearest train-test pairs by lexical sequence ratio within shared databases.

E.2 Semantic Overlap Analysis

To capture semantic alignment beyond surface-level n-grams, we compute dense representations using `sentence-transformers/all-MiniLM-L6-v2` (384-dimensional). To avoid truncation artifacts on longer synthetic task instructions, each text is chunked into overlapping windows of at most 254 tokens with a 48-token stride. We evaluate both the normalized chunk-mean document cosine similarity and the maximum pairwise chunk cosine similarity against all 1,797 same-database synthetic training candidates.

As shown in Table 14, the distribution of semantic similarities remains low to moderate:

- The median document cosine similarity across the 135 test queries is **0.4965** (P90: 0.6229; Maximum: **0.7122**).
- Crucially, **zero** test queries have a document cosine similarity or max-chunk cosine similarity ≥ 0.75 against any same-database synthetic training candidate.

Metric (over 135 test tasks)	Median	P90	P95	Maximum	$\geq$ 0.75	$\geq$ 0.80
Document Cosine Similarity	0.4965	0.6229	0.6487	**0.7122**	**0**	**0**
Max-Chunk Cosine Similarity	0.5241	0.6306	0.6639	**0.7075**	**0**	**0**

Table 14: Semantic similarity distributions between evaluation queries and candidate training queries within the same database.

Only a single pair reaches a cosine similarity above 0.70: `local259` vs. `local_syn_000531` (0.7122, on the IPL database). Manual inspection reveals that while both queries involve cricket player terminology, `local259` requires calculating complete career batting and bowling profiles, whereas `local_syn_000531` constructs a filtered cohort of bowlers who conceded zero sixes in Season 5 and computes country-level runs-conceded metrics in Season ≥ 6. The observed similarity stems naturally from domain ontology (teams, bowlers, matches) rather than task objective alignment.

E.3 Execution Result Fingerprinting

Because lexical and semantic distances alone cannot completely rule out deep semantic paraphrasing, we implement an execution result fingerprinting audit. Since the underlying SQLite databases are identical bit-for-bit, any synthesized query that unintentionally mirrors a test task's analytical intent would produce identical execution outputs on that database.

We compare the reference execution results of all synthetic tasks against the complete set of 328 official accepted ground-truth CSV outputs across the 135 test tasks:

- We cross-check every candidate pair within the same database ($N = 14{,}266$ pairs for the full generation pool; $N = 14{,}260$ pairs for the final SFT corpus).
- We evaluate across four fingerprint criteria: ordered vs. unordered row sets, and full schema (with column headers) vs. value-only sets.
- Cell values undergo standard normalization (handling of NULLs, floating-point precision, whitespace, and case insensitivity).

Across all evaluated pairs under the most permissive criterion (**ignoring column names, ignoring row order, and matching raw result values only**), the number of matches is **strictly zero**.

In summary, the 135 test tasks feature zero intersection in task identifiers, query phrasing, semantic embeddings, and execution outputs. These results demonstrate that the observed downstream execution performance cannot be attributed to training data leakage or target memorization.

F Cross-Harness Evaluation

The main comparison in Table 1 evaluates existing baselines under their native or author-recommended inference protocols rather than forcing all models into DSH-SQL. This is important because these

systems are trained for different interaction paradigms: conventional text-to-SQL models directly generate SQL from the input context, whereas agentic models interact with the database through multi-turn tool use and execution feedback. Evaluating a model under an unfamiliar harness therefore introduces an additional inference-protocol mismatch beyond differences in model capability.

To quantify this effect, we evaluate representative baselines under both their native protocols and DSH-SQL, while keeping the benchmark instances and execution-based evaluation unchanged. Under DSH-SQL, the models use the same interactive environment as our agents but receive no additional harness-specific training or adaptation.

As shown in Table 15, directly transferring these baselines to DSH-SQL consistently degrades performance. The effect is particularly pronounced for SQL-specialized models: OmniSQL-32B drops from 14.8% to 1.5% on S2-SQLite, while Arctic-Text2SQL-7B drops from 15.6% to 0%. Even the general-purpose Qwen2.5-Coder-32B declines substantially under the unfamiliar interaction protocol. These results indicate that access to an interactive harness alone does not guarantee better performance; models must also be able to operate effectively under its interaction protocol.

This finding motivates our use of native-protocol results for existing baselines in Table 1: forcing non-harness-trained models into DSH-SQL would conflate model capability with harness compatibility and systematically disadvantage methods designed for direct SQL generation. Conversely, our controlled ablations in Table 3 evaluate training strategies under the same DSH-SQL inference protocol, allowing the contribution of harness-native post-training to be isolated.

G Evaluation Benchmarks

To comprehensively evaluate reasoning capabilities, interactive workflow compliance, and cross-distribution generalization, we evaluate all models across three challenging SQLite-based benchmarks.

Spider 2.0-SQLite (S2-SQLite). Derived from the SQLite subset of Spider 2.0-Lite (Lei et al., 2025), S2-SQLite serves as our primary in-domain (ID) evaluation benchmark containing 135 complex database instances. Unlike traditional single-turn Text-to-SQL tasks with simplified schemas, S2-SQLite reflects authentic software engineering environments: databases feature complex schemas, inter-table constraints, messy multi-table joins, and enterprise data representations. Models must explore tables iteratively, formulate hypotheses, verify intermediate results, and generate the correct query without human intervention.

BIRD-Interact Mini (BI-mini). To assess interactive task completion and dialogue-driven requirement disambiguation, we evaluate on the *Mini-Interact* split of BIRD-Interact (Huo et al., 2025). We adopt the **A-Interact** protocol, wherein the model acts as an interactive SQL agent communicating with a simulated human user (powered by GPT-4o). The user simulator provides ambiguous or partial initial requests, answers clarify questions, and gives iterative feedback based on intermediate execution failures. Models are assessed on their ability to minimize unnecessary turns while successfully converging on the gold execution output.

LiveSQLBench Base-Lite SQLite (LSB-SQLite). To evaluate zero-shot transfer and robustness against contamination, we evaluate on the SQLite subset of LiveSQLBench (BIRD Team, 2026). LiveSQLBench features continually updated real-world database tasks sourced from open-source repositories and production data pipelines. The Base-Lite partition reflects realistic developer interactions, testing whether models can generalize their multi-turn exploration and schema navigation strategies to unseen database topologies, dirty schemas, and dialect-specific edge cases.

Evaluation Protocol and Tool Contract. All models—including baselines and HarnessSynth—are integrated and evaluated uniformly via the DSH-SQL interface. Across all benchmarks, execution accuracy (EX) serves as the primary metric:

$$\mathrm{EX} = \frac{1}{|D|} \sum_{i=1}^{|D|} \mathbb{I}\left(\mathrm{Exec}(Y_i, \mathcal{D}_i) = \mathrm{Exec}(Y_i^*, \mathcal{D}_i)\right), \tag{14}$$

Table 15: Cross-harness evaluation of representative baselines. *Native* denotes the original or author-recommended inference protocol; *DSH-SQL* evaluates the same model in our interactive harness without additional harness-specific training.

Model	Protocol	S2-SQLite	BI-mini	LSB-SQLite
Qwen2.5-Coder-32B	Native	16.3%	3.0%	7.4%
	DSH-SQL	3.0%	0%	5.6%
OmniSQL-32B	Native	14.8%	2.7%	8.9%
	DSH-SQL	1.5%	0%	0%
Arctic-Text2SQL-7B	Native	15.6%	2.3%	7.0%
	DSH-SQL	0%	0%	1.1%

where $\mathrm{Exec}(Y, \mathcal{D})$ denotes the result table returned by executing query Y on database $\mathcal{D}$. Missing answers, unparseable outputs, syntax errors, or queries exceeding the execution deadline are strictly materialized as non-matching sentinel errors, counting directly against the denominator.

H Data Record and Export Format

Each task blueprint contains at least `sample_id`, `database_id`, `template_id`, instruction, oracle SQL, expected columns, row bounds, operators, semantic roles, joins, and mutations. The environment record adds dialect, connection metadata, tool definitions, permissions, and budgets. The trajectory record stores the full message stream, tool arguments, observations, final SQL, execution result, verifier outcome, and selection method. Training exports remove hidden oracle fields from the model-visible context and mask loss on system, user, and tool-observation tokens.

I Structural Difficulty Control and Synthesized Task Examples

I.1 Controllable structural difficulty

For a synthesized candidate query z, let n_{tok} be its lexical token count; let n_{cte}, n_{join}, n_{sel}, n_{win}, n_{set}, n_{exists}, n_{cagg}, n_{case}, n_{time}, and n_{json} count the corresponding SQL constructs; and let d_{sel} be the maximum nesting depth at a `SELECT`. We compute

$$\begin{aligned} D_{\mathrm{str}}(z) = {} & 0.012 n_{\mathrm{tok}} + 1.2 n_{\mathrm{cte}} + 0.8 n_{\mathrm{join}} + 1.4[n_{\mathrm{sel}} - 1]_+ \\ & + 2.2 n_{\mathrm{win}} + 2.0 n_{\mathrm{set}} + 1.5 n_{\mathrm{exists}} + 1.2 n_{\mathrm{cagg}} \\ & + 0.8 n_{\mathrm{case}} + 1.0 n_{\mathrm{time}} + 1.0 n_{\mathrm{json}} + 1.2 d_{\mathrm{sel}}. \end{aligned} \tag{15}$$

This score is a transparent structural control variable, not a claim that SQL syntax alone perfectly predicts model success. A generation run specifies a mixture π_b over four bounded bands $b \in \{\mathrm{foundation}, \mathrm{core}, \mathrm{growth}, \mathrm{stretch}\}$. Each band provides lower and upper score and token bounds together with a minimum number of advanced structure families:

$$L_b^D \le D_{\mathrm{str}}(z) \le U_b^D, \qquad L_b^T \le n_{\mathrm{tok}}(z) \le U_b^T, \qquad |\mathcal{F}(z)| \ge m_b, \tag{16}$$

where $\mathcal{F}(z)$ includes measurable families such as multi-stage CTEs, windows, set or anti-joins, nested subqueries, temporal sequences, and nontrivial derived classifications. Changing π_b or the band bounds controls the synthesized curriculum without changing the underlying databases. Candidates outside the requested interval receive feature-level feedback and are simplified or enriched through bounded repair; upper bounds prevent uncontrolled complexity escalation. The score and gate are computed only on candidate SQL produced by the synthesis pipeline. No benchmark reference-query text is supplied to either component.

I.2 Representative synthesized task records

We show two tasks retained in the 2,512-trajectory SFT corpus. The instruction is visible to the rollout policy, whereas the oracle SQL, result fingerprint, and mutations are hidden during rollout. We expose the oracle SQL here only to make the construction auditable. The stored records additionally

contain the complete catalog-grounding trace, execution plan, result preview, hashes, and selected teacher trajectory.

Example 1: core task. **Record.** `sqlite_candidate_000002` uses the `Airlines` database. Its synthesized SQL has $D_{\text{str}} = 15.732$ and belongs to the core band. It composes two CTEs, two joins, JSON extraction, conditional aggregation, distinct counting, and a ranking window. The verifier records six output columns, eight result rows, and five result-changing mutations.

Public instruction (abridged for presentation). Profile August 2017 ticket revenue by aircraft model. Keep flights whose status is exactly `Arrived` and whose scheduled-departure date prefix lies between August 1 and 31, then identify each model by the English value in the aircraft-model JSON. For each model, report the distinct arrived-flight count, total revenue, revenue per flight, business-fare revenue percentage, and rank by raw total revenue. Compute decisions from raw values, round only displayed revenue and ratio fields to two decimals, and order by raw revenue descending with model name as the tie-breaker.

Hidden oracle SQL.

```
WITH arrived_legs AS (
  SELECT f.flight_id,
         json_extract(a.model, '$.en') AS model_en
  FROM flights AS f
  JOIN aircrafts_data AS a
    ON a.aircraft_code = f.aircraft_code
  WHERE f.status = 'Arrived'
    AND substr(f.scheduled_departure, 1, 10)
        BETWEEN '2017-08-01' AND '2017-08-31'
),
model_revenue AS (
  SELECT al.model_en,
         COUNT(DISTINCT al.flight_id) AS flight_ct,
         SUM(tf.amount) AS total_amt,
         SUM(CASE WHEN tf.fare_conditions = 'Business'
                  THEN tf.amount ELSE 0 END) AS business_amt
  FROM arrived_legs AS al
  JOIN ticket_flights AS tf ON tf.flight_id = al.flight_id
  GROUP BY al.model_en
)
SELECT model_en AS aircraft_model,
       flight_ct AS arrived_flights,
       ROUND(total_amt, 2) AS total_revenue,
       ROUND(total_amt * 1.0 / flight_ct, 2) AS revenue_per_flight,
       ROUND(100.0 * business_amt / total_amt, 2)
         AS business_revenue_pct,
       RANK() OVER (ORDER BY total_amt DESC) AS revenue_rank
FROM model_revenue
ORDER BY total_amt DESC, aircraft_model ASC;
```

The first verified row is `(Sukhoi Superjet-100, 1748, 1603407500.0, 917281.18, 29.58, 1)`.

Example 2: stretch task. **Record.** `dbt_score20_30_000001` uses the `activity` database. It was generated under the bounded contract $20 \leq D_{\text{str}} \leq 30$ and has $D_{\text{str}} = 20.824$, 302 SQL tokens, and two measured advanced families: window computation and nontrivial derived classification. The verifier records eight output columns, eight result rows, and five result-changing mutations.

Public instruction (abridged for presentation). Audit how completely an expected event set covers the raw `input__aggregate_in_between` stream. Keep all stream rows and match coverage by `activity_id` with a left join. For each entity, anchor the sequence at its earliest `visit page`; then aggregate by entity and activity, reporting total events, post-anchor events, covered events, covered revenue, within-group coverage, and share of the entity's events. Use raw values for all decisions and round only the final percentages.

Hidden oracle SQL.

```
WITH stream AS (
  SELECT s.activity_id, s.entity_uuid, s.activity, s.ts,
    MIN(CASE WHEN s.activity = 'visit page' THEN s.ts END)
      OVER (PARTITION BY s.entity_uuid) AS first_visit_ts,
    COUNT(*) OVER (PARTITION BY s.entity_uuid) AS entity_event_total
  FROM input__aggregate_in_between AS s
```

```
),
matched AS (
  SELECT st.entity_uuid, st.activity, st.ts, st.first_visit_ts,
    st.entity_event_total,
    CASE WHEN d.activity_id IS NULL THEN 0 ELSE 1 END AS is_covered,
    CASE WHEN d.activity_id IS NULL THEN 0
         ELSE COALESCE(
           d.aggregate_in_between_bought_something_revenue_impact, 0)
    END AS covered_revenue
  FROM stream AS st
  LEFT JOIN dataset__aggregate_in_between_1 AS d
    ON d.activity_id = st.activity_id
)
SELECT m.entity_uuid, m.activity,
  COUNT(*) AS events_total,
  SUM(CASE WHEN m.first_visit_ts IS NOT NULL
           AND m.ts >= m.first_visit_ts THEN 1 ELSE 0 END)
    AS events_on_or_after_first_visit,
  SUM(m.is_covered) AS events_in_expected_output,
  SUM(m.covered_revenue) AS expected_bought_revenue,
  ROUND(100.0 * SUM(m.is_covered) / COUNT(*), 2)
    AS pct_events_in_expected_output,
  ROUND(100.0 * COUNT(*) / MAX(m.entity_event_total), 2)
    AS pct_of_entity_events
FROM matched AS m
GROUP BY m.entity_uuid, m.activity
ORDER BY m.entity_uuid ASC, COUNT(*) DESC, m.activity ASC;
```

The first verified row is `(1, visit page, 4, 4, 4, 100, 100.0, 50.0)`. Together, the core and stretch examples illustrate how changing the bounded difficulty contract alters the required composition depth while preserving the same catalog-grounded, execute-and-verify construction protocol.

J Verifier Checklist

1. **Static SQL safety:** a single read-only `SELECT` or `WITH` statement; no write, DDL, attachment, or multi-statement escape.
2. **Schema validity:** referenced tables and columns exist; joins contain predicates; dialect-sensitive constructs are supported.
3. **Live execution:** query planning and bounded execution succeed on the designated database.
4. **Result quality:** output columns and row bounds match the blueprint; the relation is non-empty and non-degenerate where required.
5. **Consistency:** instruction, program, and expected result agree; targeted mutations should not preserve the verified result.
6. **Trajectory protocol:** tool arguments are valid, access is read-only, the budget is respected, and the final full SQL was executed before submission.

K Generation Diagnostics

Table 16: Verified trajectories retained after three teacher rollouts per task.

Source	Verified trajectories	Share
Spider 2.0-Lite	1,797	71.5%
Spider 2.0-DBT	715	28.5%
Combined	2,512	100.0%