TOWARDS SPECIALIZED TEXT-TO-SQL MODELS WITH SQALE

Cornelius Wolff[1,2], **Daniel Gomm**[1,2], **Madelon Hulsebos**[2]
[1]University of Amsterdam, [2]Centrum Wiskunde & Informatica
Amsterdam, Netherlands
{cornelius.wolff, madelon.hulsebos}@cwi.nl

ABSTRACT

The strong coding generation capabilities of large language models have accelerated progress in text-to-SQL, the task of converting natural language questions into executable SQL queries. The state-of-the-art in text-to-SQL is currently set by frontier language models embedded in composite agentic pipelines that incur substantial inference cost. Specialised small models would avoid this cost but training them requires data reflecting the scale, semantics, and structural diversity of real-world databases. Towards this vision, we introduce SQALE, a large-scale semi-synthetic text-to-SQL dataset built on 9,259 schemas from SchemaPile, a collection of real-world schemas extracted from GitHub. We present an informed data generation pipeline that combines schema extension, value synthesis, question generation, and agentic answering under execution-based validation at every stage, and produce 1,408,056 high-quality natural language (NL) questions pair with 176,761 corresponding SQL queries. Besides the large scale of SQaLe, our analysis shows that it offers significantly larger schemas than existing datasets and rich query distribution. We demonstrate how SQaLe empowers small specialised text-to-SQL models by training a small language model with GRPO directly on SQaLe that improves the untrained base model by nearly 33 points on BIRD dev, which is an unprecedented gain. We find that, among other factors, key to this performance is that training on SQaLe enables the model to learn exploring the database. It finds the relevant tables more often and looks up far more of the values it needs than when trained on other datasets. The dataset, generation pipeline, and training code are available at: https://trl-lab.github.io/sqale/.

1 INTRODUCTION

Text-to-SQL systems translate natural-language questions into executable SQL queries, enabling non-experts to query relational databases. Recent progress has been driven largely by general-purpose large language models (LLMs) and increasingly complex agentic pipelines that combine schema linking, tool use, self-correction, and majority voting. While these approaches achieve strong benchmark performance, they are computationally expensive and difficult to deploy at scale.

We propose a shift from LLMs to small specialised text-to-SQL models. Prior to the release of LLMs, text-to-SQL research progressed from template- and rule-based systems bounded by heavy manual engineering and limited domain flexibility, to statistical and neural encoder-decoder models that were trained end-to-end on annotated corpora (Liu et al., 2025a). To improve generalisation to unseen schemas, researchers developed increasingly complex architectures incorporating schema linking, graph neural networks, and execution-guided decoding (Wang et al., 2018; Katsogiannis-Meimarakis and Koutrika, 2023). The release of LLMs such as GPT-3 (Brown et al., 2020) and Codex (Chen et al., 2021) resulted in a shift towards prompting and agentic pipelines that leverage the few-shot SQL generation capabilities of these models, often combined with self-refinement, majority voting, and tool use to boost performance (Wang et al., 2025; Shkapenyuk et al.; Li et al., 2026). Despite achieving state-of-the-art performance on text-to-SQL benchmarks, such as Spider 2.0 (Lei et al., 2025) and BIRD (Li et al., 2023), these LLM-powered approaches are computationally expensive to run and difficult to deploy in practice. This motivates us a pivot to small-scale specialised models, conditioned on breaking the data barrier (Li et al., 2026).

Specialised text-to-SQL models require large high-quality datasets. While the pretraining and finetuning datasets for Large Language Models have been growing rapidly in size and diversity (Liu et al., 2025b), the same cannot be said for text-to-SQL datasets. The most widely used datasets, i.e., benchmarks such as Spider 2.0 (Lei et al., 2025) and BIRD (Li et al., 2023), contain at most around ten thousand (question, schema, SQL) pairs over fewer than a hundred schemas, with limited complexity and diversity. This is symptomatic of a broader gap recently diagnosed across the communities working on extracting insights from table corpora: prevailing benchmarks presuppose that the relevant tables are already known, and their validation mechanisms do not match the setting in which such systems are actually used (Gomm et al., 2026). Realistic training and evaluation data therefore has to confront models with schemas large enough that finding the relevant tables is itself part of the task. The large synthetic dataset SynSQL (Li et al., 2025), with 2.5 million samples, lacks the necessary schema complexity for training specialised models that truly reflect real-world database structures. This scarcity of high-quality training data is a major bottleneck for developing new text-to-SQL models, especially smaller ones that cannot rely on the few-shot capabilities of large LLMs.

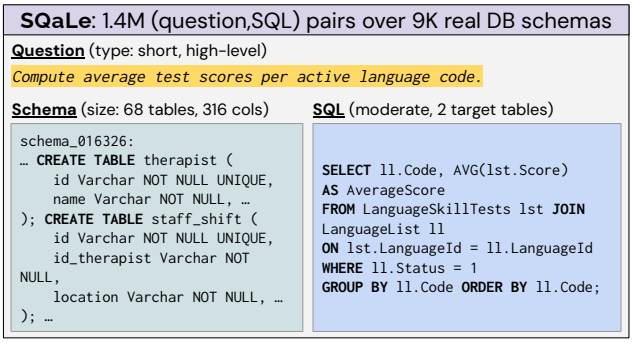


Figure 1: Excerpt of an entry of SQaLe showing a natural language question with corresponding SQL query, over a real-world target schema (table values not illustrated).

The SQaLe dataset fuels the next-generation of text-to-SQL models. To enable the next-generation of specialised text-to-SQL models, we introduce SQaLe, a large-scale, diverse, semi-synthetic text-to-SQL dataset grounded in real-world database schemas from SchemaPile (Döhmen et al., 2024), comprising 1,408,056 (question, schema, SQL) pairs. SQaLe is constructed via a structured generation pipeline that extends real-world database schemas to realistic sizes, populates them with coherent synthetic data, and generates (question, schema, SQL) pairs that are validated through execution-based and semantic filtering and iterative repair. Through this pipeline, the SQaLe dataset provides 176,761 queries over 9,259 real schemas, an example of which is presented in Figure 1. We demonstrate the utility of SQaLe by training a small model, M_{SQaLe}, with reinforcement learning on SQaLe and evaluate it on standard benchmarks: the resulting 2B model reaches 66.3% execution accuracy on the SQaLe test set, 27.7 points above the untrained base model it starts from, and outperforms most general-purpose models of 9–12B at a fraction of their inference compute, pushing the compute–accuracy Pareto curve. We also show that M_{SQaLe} learns to explore databases, finding relevant tables more often and reading far more of their values than models trained elsewhere.

Our contributions. We make the following contributions: 1) We propose a scalable, validated data generation pipeline for training specialized text-to-SQL models grounded in real-world schemas; 2) we introduce SQaLe, a large-scale and high-quality text-to-SQL dataset (1,408,056 pairs) with challenging questions with corresponding SQL queries over dense diverse real-world schemas; 3) we present an analysis of the SQaLe dataset in comparison with existing datasets and benchmarks; 4) we demonstrate that training on SQaLe results in small, specialised text-to-SQL models with the largest reinforcement-learning-only gain reported to date, which learn to explore the database on their own; and 5) we identify that agentic models trained on large and populated schemas learn to retrieve the relevant tables and explore values needed for the question rather than copying them from the question, while topical diversity is key for cross-domain generalization.

2 SQaLe: The First Large High-Quality Text-to-SQL Dataset

2.1 Dataset Design Criteria

For the design of SQaLe, we follow principles that ensure both realistic data distributions and diverse semantic coverage, for broad applicability and generalisation. We distinguish between schema-level,

data-level, and query-level considerations. These criteria are grounded in empirical analyses of production databases and informed by benchmarks such as Spider 2.0 (Lei et al., 2025), BIRD (Li et al., 2023), and SQLStorm (Schmidt et al., 2025), as well as user studies on how practitioners naturally formulate database queries (Dörschner et al., 2026).

Schema-level criteria. A realistic dataset should include schemas ranging from small, domain-specific databases to large, enterprise-scale systems containing hundreds of tables **(C1: schema size)** (van Renen et al., 2024; Vogelsgesang et al., 2018). This variety reflects the scale of modern data ecosystems observed in industry and academic corpora. Schemas should furthermore cover a wide range of domains, ensuring that trained models are exposed to heterogeneous subject matter rather than a narrow slice of application areas **(C2: domain diversity)** (Lei et al., 2025). Finally, schema naming should reflect authentic, heterogeneous practices, including inconsistent abbreviations, mixed casing, and domain-specific terms **(C3: naming conventions)** (Bodensohn et al., 2024). Preserving this variability enhances validity and prevents overfitting to unrealistically sanitised schemas.

Data-level criteria. Beyond the schemas themselves, training and evaluation, for example through query execution, require schema content. This data must satisfy primary key uniqueness, foreign key referential integrity, column types, and nullability constraints **(C4: structural compliance)** (Codd, 1970). Values must respect foreign key dependencies, with cyclic references handled explicitly **(C5: dependency-aware ordering)** (Kothapalli et al., 2026). Column values must be plausible and internally consistent, for example valid country codes, non-overlapping date intervals, and coherent categorical assignments **(C6: semantic coherence)** (Borisov et al., 2023). Rows should also contain varied names, dates, numeric magnitudes, and categorical distributions to be useful for downstream evaluation **(C7: data diversity)** (Sidorenko, 2025).

Query-level criteria. Queries should span complex SQL constructs such as nested subqueries, multi-way joins, and aggregations, and vary in natural language phrasing **(C8: query complexity)** (Vogelsgesang et al., 2018; Schmidt et al., 2025). They should also contain ambiguity that reflects a user's incomplete knowledge of the schema or its values **(C9: ambiguity and variability)** (Saparina and Lapata, 2024; Li et al., 2023; Dörschner et al., 2026) as over 60% of real-world NLQs exhibit at least one such semantic challenge, a share that benchmarks built by paraphrasing SQL do not reproduce (Dörschner et al., 2026). Furthermore, questions should use the entities, terminology, and values of their schema's domain rather than generic phrasing, since questions that fit any schema withhold the lexical evidence that schema linking relies on **(C10: domain grounding)** (Chen et al., 2025; Li et al., 2023). Finally, each SQL query must match its question and execute against its database **(C11: accuracy)** (Yu et al., 2018).

Dataset-level criteria. Beyond properties of individual samples, the dataset as a whole must exhibit statistical compatibility with established reference datasets (Li et al., 2023; Lei et al., 2025; Dörschner et al., 2026). Observable distributions (such as clause and operator frequencies, join depth, and the prevalence of linguistic variation and semantic challenge in natural language phrasing) should be broadly comparable, ensuring that models trained or evaluated on SQaLe generalise predictably to real-world evaluation conditions **(C12:generalizability)**.

2.2 Dataset Construction

The dataset construction pipeline follows a structured, multi-stage approach that integrates schema extension, synthetic data population, question synthesis, SQL generation, and linguistic reformulation, which we describe in detail here. Figure 2 presents the end-to-end dataset construction pipeline, and Appendix M lists the models used at each stage.

Schema collection and extension. SQaLe builds on SchemaPile (Döhmen et al., 2024; Dohmen et al., 2022), 23K real-world database schemas extracted from permissively licensed GitHub repositories **(C2)**. SchemaPile records the source repository of each schema but not the domain of the application behind it, which the value synthesis needs as semantic context **(C6)**. We therefore annotate each of the 14,597 source repositories with a tool-using LLM agent that explores a shallow

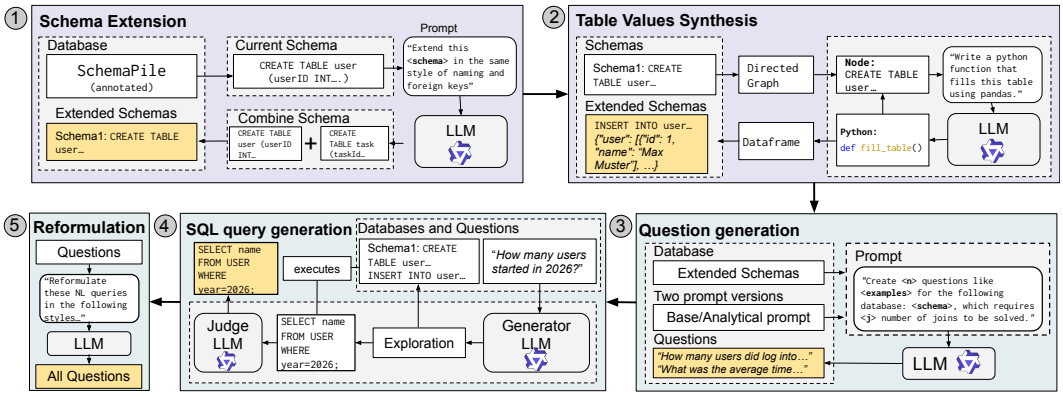


Figure 2: The data generation pipeline for the SQaLe dataset, with simplified prompts for clarity. The pipeline ① extends real-world schemas from SchemaPile, ② populates the empty tables with realistic values, ③ generates relevant questions per schema, ④ generates corresponding SQL queries, and ⑤ rewrites questions to reflect variance in formulation (e.g., verbosity level).

clone, prioritising READMEs, package manifests and schema files, and emits a short domain description, falling back to the table and column names when a repository is no longer available; the annotations are released as `trl-lab/schemapile_annotated`. To reach diverse schema sizes **(C1)**, each schema is then extended with LLM-generated tables that keep its naming conventions, level of normalisation and foreign-key style **(C3, C4)**, which brings the released schemas to a median of 113 tables (Section 3).

Table value synthesis. For training and evaluation, especially of agentic pipelines that interact with the data, the schemas in SQaLe are populated with values. Prior work models such data statistically (Patki et al., 2016) or with generative models that capture inter-table dependencies, such as graph-based VAEs (Mami et al., 2022), GANs (Gueye et al., 2022) and diffusion models (Hudovernik), but none of these draw on the semantics of the schema definitions or scale to the variety of schemas found in the wild. We therefore combine graph-based dependency analysis with LLM code generation and a validation and repair loop. Table contents are generated in foreign-key dependency order **(C5)**, each by a Python function the model writes from the table's DDL, its original SchemaPile rows, the allowed values of its foreign keys and the schema's domain description **(C6)**, so that a large fact table costs no more to generate than a small lookup table. Fact and junction tables receive several times more rows than the rest and a skewed foreign-key distribution (the most-used fifth of parent keys holds a median 67% of a fact table's rows, against 34% under uniform sampling), which keeps aggregations over the data non-trivial **(C7)**. Every generated table is checked for primary-key uniqueness and referential integrity before it is accepted **(C4)**; Appendix E gives the full procedure.

Question generation. Questions are generated over subgraphs of the populated schemas. Starting from a random seed table, sampling expands along foreign keys until it reaches a target table count drawn from a distribution spanning up to twenty tables, so that the join complexity of a question is fixed by that distribution rather than left to the generating model **(C8)**. Each subgraph is paired with one sample of the previously generated rows per table and used to prompt question generation, where a model is instructed to produce a natural language question answerable only by querying all provided tables, at one of three target difficulty levels (simple, moderate, hard). Malformed outputs are rejected and regenerated. Questions are generated with two versions of the generation prompt, each producing half of the corpus. Both require the question to state the information need in full, to make clear why each table of the subschema is required, and to ground every literal in the sampled rows, without ever referring to the schema **(C10)**. The first leaves the choice of question to the subschema, while the second asks for analytical shapes the model rarely writes unprompted, such as a per-group measure feeding the final answer, a ranking within each group, or a cohort defined by one action and measured by another.

SQL query generation and alignment. The gold SQL query for each question is generated through an exploratory, tool-augmented loop (prompt in Appendix B, tools in Appendix L). The model is given the natural-language question and may freely call database exploration tools (such as listing tables, inspecting schemas, sampling rows, or running test queries) before committing to a final answer via `submit_sql()`. Once a query is submitted it is executed against the same in-memory database; execution errors are fed back as a correction prompt (augmented with a summary of the exploration findings) and the model may retry up to a fixed budget. If execution succeeds, a second LLM call acting as a judge evaluates the result sample against the original question, checking for empty or duplicate results and semantic correctness; a rejected query triggers another rewrite using the judge's critique, until either the query is accepted or the retry budget is exhausted **(C11)**. Only questions whose query is accepted enter the corpus.

Question style variation. To satisfy the linguistic diversity criterion **(C9)**, an LLM rewrites every accepted question into seven styles. This lets us measure the effect of question style in isolation, instead of leaving it entangled with query content as in existing benchmarks (Wretblad et al., 2024). If the rewriting model fails twice on a question, the question is still admitted, with its verbose original only. A *casual* variant restates the request informally and a *structured* one as bulleted or numbered requirements, while a *requirements list* drops the sentence and keeps only fragments naming what is wanted. *Short and high level* is a short paraphrase of the top-level intent and *short and ambiguous* a short version that hints at the topic while leaving part of the specification implicit, which makes underspecification a controlled axis of the corpus rather than an accident of generation. A *spelling and grammar mistakes* variant carries typing and grammar errors, and an *evidence supported* variant pairs a compact question with a short evidence note stating the outside knowledge needed to map it onto the data, supplying what an agent would otherwise have to look up, so that the corpus can also train single-shot models that answer in one pass.

SQaLe dataset splits. The SQaLe dataset is split into a train-test partition on schema-level, with 95% of schemas and corresponding queries assigned to the training set and the remaining 5% to the SQaLe test set, so that no schema of the test set is seen during training.

2.3 Dataset Quality Validation

We validate the LLM judge on 225 judge calls, taken from a full run of the answer mechanism over 158 BIRD dev questions across nine databases. The two counts differ because the judge runs once per submitted query and a rejected query is rewritten and judged again, so 123 questions were completed on the first call while the remaining 35 took up to four, the retry budget of the answer mechanism. Execution accuracy (EX) against the BIRD gold query confirms the judge's verdict on 142 of those calls. A mismatch is, however, not by itself evidence of a wrong query, since BIRD's gold annotations contain errors (Jin et al., 2026) and a correct query can differ from the gold one in formulation alone. One of the authors therefore manually reviewed the remaining 83, labelling the predicted query as aligned or not aligned with the expressed information need, and agreed with the judge in 52 cases. Nearly all of these are queries that answer the question through a formulation the gold query does not match. Against the resulting labels the judge agrees on 86.2% of calls ($\kappa = 0.68$) against 76.9% for EX, and *94.5% of the queries it accepts are intent-aligned* (Table 1).

Table 1: Judge verdicts against human labels, on 225 judge calls over 158 BIRD dev questions.

	EX + Human label	
Judge	**Aligned**	**Not aligned**
Accept	**138**	8
Reject	23	56
Total	161	64

3 Analysis of the SQaLe Dataset

Corpus size and coverage. Using the outlined generation pipeline, SQaLe represents a large-scale, semi-synthetic text-to-SQL resource grounded in real-world database structures. The generation pipeline results in 1,408,056 validated (question, schema, SQL) pairs across 9,259 relational database schemas. Compared to existing datasets, SQaLe offers a substantially broader and more realistic schema distribution (Table 2).

With 9,259 schemas comprising 1,103,669 tables, a median of 113 tables and 538 columns per schema, and 1,196,078 foreign key relations in total, it far exceeds the structural diversity of Spider 2.0, SynSQL, and BIRD. While not as large as SynSQL in terms of total number of queries, our dataset features vastly more and more complex schemas, emphasising depth and structural realism over volume.

Synthetic data population quality. We validate the synthetically populated values with reference-free structural metrics, reported in the lower block of Table 2: nearly all tables are filled, their foreign keys resolve after repair, and the role-conditioned row budgets leave most foreign-key columns able to carry a non-trivial aggregation. The resulting tables hold fewer rows than the real database dumps behind BIRD, but two orders of magnitude more than SynSQL's, which is what lets a question depend on values a model has to read rather than guess **(C4, C7)**.

Comparison to other corpora. On sheer size SQaLe is second only to SynSQL: it contributes more natural-language questions than any other text-to-SQL corpus, and more distinct SQL statements than any other, in both cases with SynSQL as the single exception (Table 3). Where it separates from SynSQL as a training dataset is when it comes to the complexity and realism of the underlying schemas. Its schemas are the largest of any corpus compared here on both columns and tables per schema, and by a margin that is not close **(C1)**. Larger schemas drawn from real DDL carry correspondingly more foreign keys, so SQaLe also leads on foreign keys per schema, and its tables are less often isolated than either reference corpus's; schemas admitting nothing but single-table questions are rarer here **(C1)**. Its tables are also populated far more densely than SynSQL's, so its schemas can exercise selectivity and aggregate cardinality rather than being answered by string match, though they do not reach BIRD's absolute row counts, which remains the corpus's clearest current limitation (Table 2) **(C7)**. Query composition matches BIRD on operator diversity and goes further in nesting (23.4% against 7.7%) and in multi-join chains (40% of joining queries against 26%; Table 3).

Table 2: Schema statistics for datasets. M. stands for median. The lower block reports reference-free quality metrics of the SQaLe value population. A `GROUP BY` is non-trivial when at least three of its groups span more than one row.

Metric	BIRD	EHRSQL	SynSQL	SQaLe
#Schemas	80	2	16,575	**9,259**
M. #cols/schema	39	92	72	**538**
M. #tables/schema	5.0	13.5	10.0	**113**
#FKs	526	34	159,547	**1,196,078**
M. #rows/table	3,738	–	2	**69**
SQaLe value population (9,259 schemas, 1,103,669 tables)				
Tables populated (108,708,694 rows)				95.2%
Valid foreign-key cells, post-repair				95.1%
Foreign-key columns with a non-trivial `GROUP BY`				80.0%

Domain coverage of questions. SQaLe's questions carry the vocabulary of the schema they target **(C10)** and sit in the register of the reference datasets **(C12)**. We embed a 13,103-question subsample drawn from 6,617 schemas together with 500 BIRD (Li et al., 2023) dev questions, as well as 500 EHRSQL (Lee et al.) questions, and project them in one joint UMAP fit (McInnes et al., 2018) (Figure 3). While we can observe that depending on the schema and domain, the BIRD queries are clustered into specific groups, bar a few outliers those clusters fall within SQaLe's distribution rather than beside it: several lie inside its cloud and the rest on its boundary. Something similar is also observed with EHRSQL, which due to its single domain is more tightly clustered than BIRD. Yet, there are quite a few SQaLe questions that also cover the medical domain of EHRSQL.

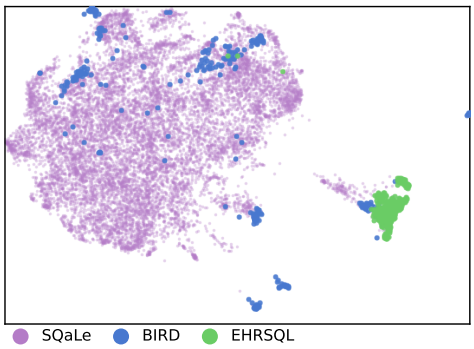


Figure 3: Samples from SQaLe (n=13,103), BIRD (n=500) and EHRSQL (n=500) embedded in a joint UMAP projection.

Table 3: SQL query statistics. All values except SQL and Natural Language Queries (NL) are in %.

Dataset	#SQL	#NL	Func.	Where	Join	Nested	Agg.
BIRD (train & dev)	10,962	10,962	13.8	88.1	76.2	7.7	47.0
EHRSQL	9,270	9,270	76.9	99.9	19.7	89.7	58.4
Spider2	250	250	45.2	94.4	72.0	95.2	84.4
SynSQL	2,544,390	2,544,390	12.1	75.6	89.4	49.4	74.6
SQaLe	**176,761**	**1,408,056**	**13.6**	**82.1**	**55.4**	**23.4**	**43.0**

Dataset content. Figure 4 compares SQaLe to prior text-to-SQL datasets on schema size, query length and tables per query. Its schemas are an order of magnitude larger than any comparator: a median of 538 columns per schema, against under 200 for BIRD, EHRSQL, Spider 2.0 and SynSQL **(C1)**. Its queries also span a wider range of lengths, from simple statements of around 100 characters to hard queries with a median of 379 and a much heavier spread **(C8)**. The number of tables a query touches has a longer tail: 3.1% of SQaLe queries touch five or more tables, up to 20, the largest subgraph the question generator samples, while no BIRD or EHRSQL query goes beyond four **(C8)**. SynSQL has a larger share of queries over three or more tables, but over schemas of under 200 columns. All of this goes to show that the SQaLe dataset is not only larger but also more structurally and semantically complex than existing datasets, making it a valuable resource for training and evaluating text-to-SQL models.

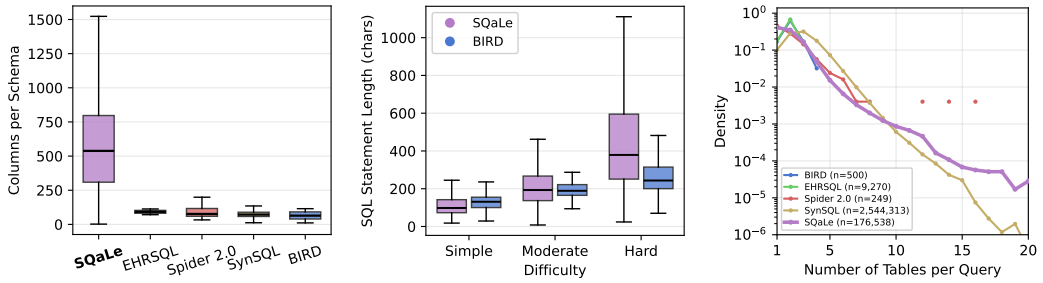


Figure 4: Characteristics of SQaLe showing that it contains more columns per schema (left), higher SQL query complexity in terms of query length (middle), with a longer tail of #tables used (right).

4 The Utility of SQaLe for Text-to-SQL Training and Evaluation

To validate the use of the SQaLe dataset we ask a single question: holding the model, the environment, the objective and the compute budget fixed, how much of a specialised text-to-SQL model's ability comes from the *data* it is trained on? We therefore train Qwen3.5-2B with reinforcement learning directly from the base checkpoint, three times, on three different corpora: SQaLe, SynSQL-2.5M (Li et al., 2025) and the BIRD training split (Li et al., 2023), and refer to the resulting models as M_{SQaLe}, M_{SynSQL} and M_{BIRD}. Apart from the training data and the strength of the length curriculum (Section 4.1), every choice is identical across the three models: base model, agentic environment, tool set, reward, GRPO objective, step count, batch size and evaluation protocol. The gap between their evaluation curves is therefore attributable to the data, and not to a training recipe tuned per corpus.

This design also separates our claim from recent small-model text-to-SQL systems such as SLM-SQL (Sheng and Shuai, 2025b), Struct-SQL (Thaker and Bresler, 2025) and FINER-SQL (Hoang et al., 2026), and from the rationalisation models of Rossiello et al. (2025). These reach strong BIRD accuracy by distilling reasoning traces out of much larger teachers into think-augmented corpora and fine-tuning on them, and several then add reinforcement learning or inference-time self-consistency on top. In contrast, we apply GRPO directly to the base checkpoint, with no distilled supervision and no test-time scaffolding at evaluation.

4.1 Training Procedure for Specialised Text-to-SQL Models with SQaLe

We optimise the base model directly with Group Relative Policy Optimisation (GRPO) (Shao et al.) against an execution-grounded reward, so that every behaviour the model acquires is learned from the reward on the training corpus itself. The detailed hyperparameters are given in Appendix I.

The environment. Each incoming sample of SQaLe is turned into a tool-use environment over a live SQLite instantiation of the schema and its data. Unlike other single-shot approaches, the DDL statement here is not handed over in full at the beginning, as this would be inefficient at the schema scale of the SQaLe dataset.

To ensure that the model works well on both small and large schemas, we apply a graph-aware schema context algorithm (outlined in detail in Appendix F), which starts at the needed gold tables and then propagates along the foreign key relations of the tables, until a randomly set number of total tables has been reached.

The agent has access to four tools to explore schemas in the database, two tools for content inspection, and the ability to run custom read-only queries. All tools and episode limits are described in Appendix L, and the prompt in Appendix D.

Question `Which supplier shipped the most back-ordered items last quarter?`

Turn 1 `{"tool": "list_tables"}`
→ *suppliers, shipments, order_items, … (113 tables)*

Turn 2 `{"tool": "join_path", "args": {"table_a": "suppliers", "table_b": "order_items"}}`
→ *suppliers → shipments → order_items*

Turn 7 (final) `{"tool": "submit_sql", "args": {"sql": "SELECT s.name ..."}}`
→ *result matches gold: $r = 3.0$*

Figure 5: One training episode with the schema withheld; turns 3–6 omitted. Multiple tool calls in one turn are possible.

The curriculum. To prevent the model from overoptimising on solving the simple queries, we bucket questions by gold-SQL length and anneal the sampling mixture from the default distribution towards that tail, while keeping short queries in the mix so the model does not learn to only write long SQL. SynSQL, whose SQL lengths span a much narrower range, gets a milder version of the curriculum, and BIRD, narrower still, gets none. Full details are given in Appendix H.

The reward function. The reward is leveled by outcome. A query whose result set matches the gold one scores highest, followed by queries that execute, then queries that only parse, and anything else scores zero. An executable but wrong query gets partial credit for how much of the gold result it recovers, plus small bonuses for joining along declared foreign keys and for agreeing with the most common result in its group. The levels are spaced so that no partial credit can rank a wrong query above a correct one. The exact reward terms are given in Appendix G.

4.2 SQaLe pushes the Pareto Curve and Generalises Beyond Its Corpus

We compare M_{SQaLe}, M_{BIRD} and M_{SynSQL} and two reference base models using the same agentic harness and settings (turn limits, no test-time scaffolding) that was introduced for training in Section 4.1 using 300 questions from SQaLe test, BIRD dev and EHRSQL, each evaluated on the last checkpoint of its run, with SQaLe evaluated over the full schema.

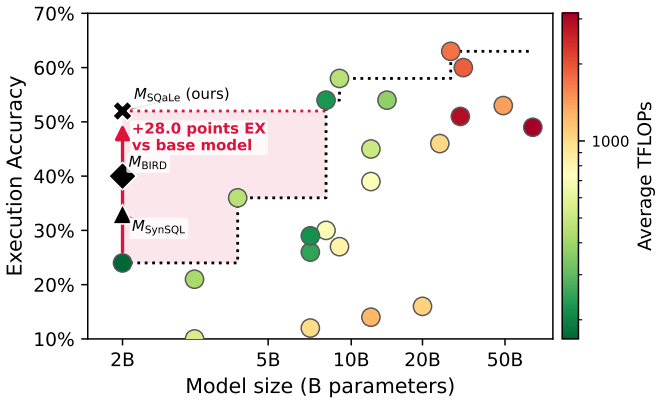


Figure 6: Accuracy against model size on 100 moderate questions from the SQaLe test set, with the average compute each model spends per question carried as colour.

SQaLe closes most of the gap to much larger models. Table 4 sets the three trained models against the untrained base models as a reference. On

SQaLe test, M_{SQaLe} reaches an execution accuracy of 66.3%, a gain of 27.7 points over the base model. With the same pipeline, M_{BIRD} gains only +15.3 points on SQaLe test and M_{SynSQL} +12.0, so the gain is based on the corpus rather than the model objective (details in Section 4.3).

On 100 moderate questions from the SQaLe test set (Figure 6), the 2B M_{SQaLe} reaches 52%, ahead of every general-purpose model in the 9–12B range except Qwen3.5-9B and 11 points behind Qwen3.5-27B at 224 rather than 1,710 TFLOPs per question, placing it above the Pareto curve of the general-purpose models (labelled version in Appendix J).

4.3 The Distinctive Value of SQaLe

Schema size matters. To understand the impact of the larger schemas in SQaLe, we evaluate the SQL generation performance for each model for various schema sizes of schemas in the SQaLe test set. We observe that accuracy falls for every model as the schema grows with more tables, and M_{SQaLe} is strongest across all schema sizes (gold, +32 tables, full schema). Table 5a shows that while only the gold tables are present, all three models succeed in exploring those tables, but when reaching full schema size, M_{SQaLe} leads gold table discovery with 89.7% of episodes against 85.0% for M_{BIRD} and 83.3% for M_{SynSQL}, and it keeps its accuracy lead at every schema size. The schemas in SynSQL and BIRD are small (5-10 tables), so M_{BIRD} *and* M_{SynSQL} *don't learn to search for relevant tables, which is a behavior that emerges in* M_{SQaLe}.

Table 4: Execution accuracy (%) on the SQaLe test set, BIRD dev and EHRSQL.

Model	SQaLe	BIRD	EHRSQL
M_{SQaLe}	**66.3**	52.3	**23.7**
M_{BIRD}	54.0	**54.7**	**23.7**
M_{SynSQL}	50.7	44.3	13.3
Qwen3.5-2B	38.7	19.3	8.2
Qwen3.6-27B	76.0	69.3	55.0

Populated tables matter. Reaching the gold tables is not enough on its own, since many questions turn on a string literal that the question does not state and that exists only in the table values. To understand whether the models search for these values, we check for each string literal in the gold SQL whether the agent found it in a tool result before submitting or wrote it into its SQL without ever looking it up (Figure 7). We observe that all three trained models account for a similar share of the literals in total, but only M_{SQaLe} consistently reads them out of the data (in 88% of the cases), while M_{BIRD} and M_{SynSQL} copy values more often from the question directly without checking it against values in the database.

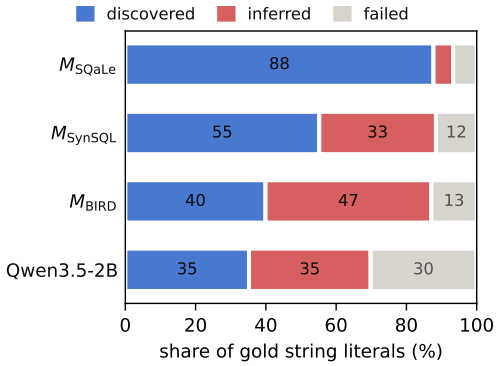


Figure 7: Methods to identify SQL literals across models: as a result from data exploration using tools, or copying values from the natural language question into the SQL without looking up. M_{SQaLe} learns to lookup the relevant values from the tables, and does so in 88% of the cases.

This difference matters once the question no longer supplies the value. Where the question states the literal, all three trained models put it into their SQL equally often (92%), but where it does not, M_{SQaLe} searches the data for 60.6% of the literals against 27.3% for M_{BIRD} and 37.4% for M_{SynSQL}, and submits the correct value in 54.5% of cases against 41.4% and 39.4%. The schemas in SynSQL hold a median of only two rows per table against 69 in SQaLe, and while M_{BIRD} gains accuracy over the base model, it does not learn to read the database before answering. *Large schemas make the agent find the table, and populated tables make it find the value inside.*

Table 5: Execution accuracy (%) on SQaLe test illustrating the effects of (a) schema size, and (b) domain coverage measured as embedding distance to BIRD dev schemas.

Number of tables	M_{SQaLe}	M_{BIRD}	M_{SynSQL}
Gold only	**72.7** (95.3)	64.7 (98.7)	59.3 (98.3)
+32 distractor tables	**68.0** (96.7)	61.0 (90.7)	52.3 (92.7)
Full schema	**66.3** (89.7)	54.0 (85.0)	50.7 (83.3)

(a) Accuracy (share of gold tables inspected in brackets)

Domain	M_{BIRD}	M_{SQaLe}	Diff.
In-domain	36.5	**39.3**	+2.8
Out-of-domain	36.5	**46.4**	+9.9

(b) Performance on in-domain vs out-of-domain databases.

Domain diversity matters. Even though the ability to discover the gold tables explains part of the performance difference, a gap of 8.0 points to M_{BIRD} persists even on the gold tables alone. Embedding each SQaLe test set schema's table and column names and taking its cosine similarity to the nearest BIRD dev database separates the third of the questions whose schemas BIRD dev covers from the two thirds it does not (Table 5b). M_{SQaLe} leads by 9.9 points on the questions outside the domains of BIRD train and by 2.8 on those they do, while M_{BIRD} scores the same in both groups (36.5). Covering SQaLe's domains is valuable, precisely because its schemas are SchemaPile's real databases mined from GitHub repositories (Döhmen et al., 2024). So, the semantics of the questions and databases outside BIRD train's domains are the ones relevant in practice and we show that SQaLe's diversity captures that.

5 Conclusion

We introduce a pipeline that turns SchemaPile's real-world schemas into populated databases with verified (question, schema, SQL) pairs, and ran it to produce SQaLe the first large-scale text-to-SQL training dataset that is grounded in real-world schemas. SQaLe comprises 1,408,056 questions and SQL queries over 9,259 schemas. The database schemas in SQaLe are larger than those of existing corpora, and M_{SQaLe}, a 2B model trained on it, outperforms most general-purpose models in the 9–12B range on the SQaLe test set while leading M_{BIRD} and M_{SynSQL} at every schema size. The gain comes from behaviour that SQaLe enforces: M_{SQaLe} finds the relevant tables in full-size schemas, reads the values it needs from the data, and generalises to domains BIRD train does not cover. The SQaLe dataset, generation pipeline and training code will be open-sourced.

Acknowledgements

This work was supported by the Dutch Research Council (NWO, AiNed Fellowship grant NGF.1607.22.045) and a grant from SAP. We also gratefully acknowledge support through compute credit grants from SURF Snellius EINF-19698 and EINF-17700.

References

Jan-Micha Bodensohn, Ulf Brackmann, Liane Vogel, Matthias Urban, Anupam Sanghi, and Carsten Binnig. Llms for data engineering on enterprise data. *Proceedings of the VLDB Endowment. ISSN*, 2150:8097, 2024.

Vadim Borisov, Kathrin Sessler, Tobias Leemann, Martin Pawelczyk, and Gjergji Kasneci. Language models are realistic tabular data generators. In *The Eleventh International Conference on Learning Representations*, 2023.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners, July 2020. URL http://arxiv.org/abs/2005.14165. arXiv:2005.14165 [cs].

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, July 2021. URL http://arxiv.org/abs/2107.03374. arXiv:2107.03374 [cs].

Peter Baile Chen, Fabian Wenz, Yi Zhang, Devin Yang, Justin Choi, Nesime Tatbul, Michael Cafarella, Çağatay Demiralp, and Michael Stonebraker. BEAVER: An Enterprise Benchmark for Text-to-SQL, January 2025. URL http://arxiv.org/abs/2409.02038. arXiv:2409.02038 [cs].

Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

Till Dohmen, Madelon Hulsebos, Christian Beecks, and Sebastian Schelter. GitSchemas: A Dataset for Automating Relational Data Preparation Tasks. In *2022 IEEE 38th International Conference on Data Engineering Workshops (ICDEW)*, pages 74–78, Kuala Lumpur, Malaysia, May 2022. IEEE. ISBN 978-1-6654-8104-5. doi: 10.1109/ICDEW55742.2022.00016. URL https://ieeexplore.ieee.org/document/9814699/.

Charlotte Dörschner, Frank Jäkel, and Carsten Binnig. Towards more realistic natural language queries in text-to-sql benchmarks. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, pages 1–8, 2026.

Till Döhmen, Radu Geacu, Madelon Hulsebos, and Sebastian Schelter. SchemaPile: A Large Collection of Relational Database Schemas. *Proceedings of the ACM on Management of Data*, 2(3): 1–25, May 2024. ISSN 2836-6573. doi: 10.1145/3654975. URL https://dl.acm.org/doi/10.1145/3654975.

Daniel Gomm, Maarten de Rijke, and Madelon Hulsebos. Open Tabular Insight Extraction: Where Do We Stand, and Where Should We Go?, September 2026. URL http://arxiv.org/abs/2609.07629. arXiv:2609.07629 [cs].

Mohamed Gueye, Yazid Attabi, and Maxime Dumas. Row Conditional-TGAN for generating synthetic relational databases, November 2022. URL http://arxiv.org/abs/2211.07588. arXiv:2211.07588 [cs].

Thanh Dat Hoang, Thanh Trung Huynh, Matthias Weidlich, Thanh Tam Nguyen, Tong Chen, Hongzhi Yin, and Quoc Viet Hung Nguyen. Finer-sql: Boosting small language models for text-to-sql, 2026.

Valter Hudovernik. Relational Data Generation with Graph Neural Networks and Latent Diffusion Models.

Tengjun Jin, Yoojin Choi, Yuxuan Zhu, and Daniel Kang. Pervasive annotation errors break text-to-sql benchmarks and leaderboards. *arXiv preprint arXiv:2601.08778*, 2026.

George Katsogiannis-Meimarakis and Georgia Koutrika. A survey on deep learning approaches for text-to-SQL. *The VLDB Journal*, 32(4):905–936, July 2023. ISSN 1066-8888, 0949-877X. doi: 10.1007/s00778-022-00776-8. URL https://link.springer.com/10.1007/s00778-022-00776-8.

Vignesh Kothapalli, Rishabh Ranjan, Valter Hudovernik, Vijay Prakash Dwivedi, Johannes Hoffart, Carlos Guestrin, and Jure Leskovec. Plurel: Synthetic data unlocks scaling laws for relational foundation models. *arXiv preprint arXiv:2602.04029*, 2026.

Gyubok Lee, Hyeonji Hwang, Seongsu Bae, Yeonsu Kwon, Woncheol Shin, Seongjun Yang, Minjoon Seo, Jongyeup Kim, and Edward Choi. EHRSQL: A Practical Text-to-SQL Benchmark for Electronic Health Records.

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

Boyan Li, Chong Chen, Zhujun Xue, Yinan Mei, and Yuyu Luo. DeepEye-SQL: A Software-Engineering-Inspired Text-to-SQL Framework, April 2026. URL http://arxiv.org/abs/2510.17586. arXiv:2510.17586 [cs].

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. OmniSQL: Synthesizing High-quality Text-to-SQL Data at Scale, July 2025. URL http://arxiv.org/abs/2503.02240. arXiv:2503.02240 [cs].

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs, November 2023. URL http://arxiv.org/abs/2305.03111. arXiv:2305.03111 [cs].

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. A Survey of Text-to-SQL in the Era of LLMs: Where are we, and where are we going?, June 2025a. URL http://arxiv.org/abs/2408.05109. arXiv:2408.05109 [cs].

Yang Liu, Jiahuan Cao, Chongyu Liu, Kai Ding, and Lianwen Jin. Datasets for large language models: A comprehensive survey. *Artificial Intelligence Review*, 58(12):403, October 2025b. ISSN 1573-7462. doi: 10.1007/s10462-025-11403-7.

Ciro Antonio Mami, Andrea Coser, Eric Medvet, Alexander T. P. Boudewijn, Marco Volpe, Michael Whitworth, Borut Svara, Gabriele Sgroi, Daniele Panfilo, and Sebastiano Saccani. Generating Realistic Synthetic Relational Data through Graph Variational Autoencoders, November 2022. URL http://arxiv.org/abs/2211.16889. arXiv:2211.16889 [cs].

Leland McInnes, John Healy, and James Melville. UMAP: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. The Synthetic Data Vault. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 399–410, Montreal, QC, Canada, October 2016. IEEE. ISBN 978-1-5090-5206-6. doi: 10.1109/DSAA.2016.49. URL http://ieeexplore.ieee.org/document/7796926/.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan Arik. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql, 2025.

Gaetano Rossiello, Nhan H Pham, Michael Glass, Junkyu Lee, and Dharmashankar Subramanian. Rationalization models for text-to-SQL. In *Workshop on Reasoning and Planning for Large Language Models*, 2025. URL https://openreview.net/forum?id=QgUVtHcHzI.

Irina Saparina and Mirella Lapata. Ambrosia: A benchmark for parsing ambiguous questions into database queries. *Advances in Neural Information Processing Systems*, 37:90600–90628, 2024.

Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proceedings of the VLDB Endowment*, 18(11):4144–4157, July 2025. ISSN 2150-8097. doi: 10.14778/3749646.3749683. URL https://dl.acm.org/doi/10.14778/3749646.3749683.

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.

Lei Sheng and Xu Shuai Shuai. Csc-sql: Corrective self-consistency in text-to-sql via reinforcement learning. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pages 1473–1496, 2025a.

Lei Sheng and Xu Shuai Shuai. Slm-sql: An exploration of small language models for text-to-sql. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pages 1497–1512, 2025b.

Vladislav Shkapenyuk, Divesh Srivastava, Parisa Ghane, and Theodore Johnson. Automatic Metadata Extraction for Text-to-SQL.

Andrey Sidorenko. A note on statistically accurate tabular data generation using large language models. *arXiv preprint arXiv:2505.02659*, 2025.

Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1 (2):146–160, June 1972. ISSN 0097-5397, 1095-7111. doi: 10.1137/0201010. URL http://epubs.siam.org/doi/10.1137/0201010.

Khushboo Thaker and Yony Bresler. Knowledge distillation with structured chain-of-thought for text-to-sql, 2025.

Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. Why tpc is not enough: An analysis of the amazon redshift fleet. *Proceedings of the VLDB Endowment*, 2024.

Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. Get real: How benchmarks fail to represent the real world. In *Proceedings of the Workshop on Testing Database Systems*, pages 1–6, 2018.

Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. Robust Text-to-SQL Generation with Execution-Guided Decoding, September 2018. URL http://arxiv.org/abs/1807.03100. arXiv:1807.03100 [cs].

Pengfei Wang, Baolin Sun, Xuemei Dong, Yaxun Dai, Hongwei Yuan, Mengdie Chu, Yingqi Gao, Xiang Qi, Peng Zhang, and Ying Yan. Agentar-Scale-SQL: Advancing Text-to-SQL through Orchestrated Test-Time Scaling, December 2025. URL http://arxiv.org/abs/2509.24403. arXiv:2509.24403 [cs].

Niklas Wretblad, Fredrik Riseby, Rahul Biswas, Amin Ahmadi, and Oskar Holmström. Understanding the effects of noise in text-to-sql: An examination of the bird-bench benchmark. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 356–369, 2024.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244, 2026.

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*, pages 3911–3921, Brussels, Belgium, 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1425. URL http://aclweb.org/anthology/D18-1425.

A Question Generation Prompt

Figure 8 shows the prompt used to generate NL questions from subschemas. The schema block interleaves `CREATE TABLE` statements with one sample of rows per table. Difficulty counts are split as evenly as possible: given total N and $r = N \bmod 3$, $n_{\text{simple}}=\lfloor N/3 \rfloor+\mathbf{1}[r>0]$, $n_{\text{moderate}}=\lfloor N/3 \rfloor+\mathbf{1}[r>1]$, $n_{\text{hard}}=\lfloor N/3 \rfloor$, with $N \geq 3$ enforced. On parse failure the retry suffix is appended and resubmitted up to three times.

Two versions of the prompt are used, each producing half of the corpus, and both fix the verbose register described in Section 2.2. Both require every question to be answerable only by joining all tables of the subschema, to vary the operation asked for across retrieval, aggregation, filtering, sorting and grouping, to ground any literal it mentions in the sampled rows, and to avoid both references to the schema itself and near-repetitions of earlier questions in the same batch. Questions are further required to be phrased so that a correct SQL query answering them cannot return an empty result, which keeps execution-based validation informative. The first version leaves the choice of question to the subschema. The second additionally asks for analytical shapes the model rarely writes unprompted, such as a per-group measure feeding the final answer, a ranking within each group, or a cohort defined by one action and measured by another.

Role

```
You are an expert at writing diverse, realistic NL questions for SQL
schemas.  Answer ONLY with a JSON code block.  No explanations.
```

User

```
Generate {N} questions for the following schema:
```
[CREATE TABLE ... – sample rows; repeated per table]

Questions must require ALL tables. Use verbose, descriptive phrasing. Vary task types. Reference sample values where natural. Do not mention the schema. Do not repeat questions. IMPORTANT: questions must not admit an empty SQL result.

Difficulty: ''simple'' (SELECT/WHERE/aggregation), ''moderate'' (JOINs, GROUP BY, subqueries), ''hard'' (multiple JOINs, window functions, CTEs).

Provide exactly $\{n_s\}$ *simple,* $\{n_m\}$ *moderate,* $\{n_h\}$ *hard.*

[JSON code block: keys "simple", "moderate", "hard", each a list of strings]

Assistant

[JSON with "simple", "moderate", "hard" arrays]

Retry suffix (on parse failure)

```
# FORMAT REMINDER (retry {n}):  return ONLY ```json ... ```with keys
''simple'', ''moderate'', ''hard''.  No other text.
```

Figure 8: Question generation prompt. Italic grey text: runtime placeholders. n_s, n_m, n_h: per-tier counts derived from total N. The retry suffix is appended on parse failure (up to 3 rounds).

B Question-Answering Prompt Template

Figure 9 shows the multi-turn prompt structure used by the exploratory answering pipeline. The model receives a system prompt and an initial user message, then enters a tool-calling loop: it may call `list_tables`, `describe_table`, `sample_rows`, `distinct_values`, and `run_query` before finalising its answer with `submit_sql`. All tool calls are dispatched synchronously against an in-memory SQLite instance populated with the actual table rows; results are appended to the conversation and a new LLM round is issued. Extended thinking is disabled during this phase to allow reliable structured tool-call emission. After submission the SQL is executed and passed to a separate judge call; on rejection the session retries with the critique or execution error appended.

System

```
You are an expert text-to-SQL agent with access to database
exploration tools.  Keep in mind that every question has a plausible
answer.  Follow this strict loop:  (1) reason about required
tables/joins/filters; (2) explore with list_tables, describe_table,
sample_rows, distinct_values, run_query; (3) build and test the
query; (4) call submit_sql() with the final answer.  Rules:  MUST
call submit_sql(); valid SQLite SELECT or CTE only; verify schema
details with tools before submitting.
```

User

```
Use the database exploration tools to answer the following question,
then call submit_sql() with your final SQL query.
```

[with –include_ddl: full SCHEMA block inserted here]

`QUESTION:` *[natural language question]*

Tool loop (repeated)

[Assistant: tool_call(s) or plain text]

`Tool result:` *[tool output]*

[Budget nudge injected when ≤5 calls remain:
You have {k} call(s) remaining. Call submit_sql() immediately.]

Judge (separate call, post-submission)

System: `Check empty result → duplicates → content correctness. Output ACCEPT or REJECT: <reason>.`

User: `SCHEMA:` *[DDL]* `QUESTION:` *[question]*
`SQL QUERY:` *[submitted SQL]*
`RESULT SAMPLE:` *[first N of total rows]*

Retry user message (on rejection or error)

`SCHEMA:` *[DDL]* `QUESTION:` *[question]*
`PREVIOUS QUERY:` *[SQL]*
[EXECUTION ERROR: {error} ***or*** *JUDGE CRITIQUE: {critique}]*
`EXPLORATION FINDINGS:` *[recent tool outputs]*

Assistant

[SQL query: plain text, no markdown]

Figure 9: Exploratory answering prompt structure. Italic grey text: runtime placeholders. The tool loop runs up to `max_tool_calls` (default 15) rounds; a budget nudge is injected when few calls remain. The judge and retry turns form a separate fixed-context call (thinking re-enabled); retries append exploration findings to preserve verified schema details.

C JUDGE PROMPT TEMPLATE

Figure 10 shows the judge prompt used to evaluate whether an executed SQL result is semantically aligned with the original natural language question. The prompt was written by the authors.

System

```
You are a very critical impartial SQL correctness judge, that checks
whether the predicted SQL query produces a result that correctly
answers the question.  Follow this three-step procedure:

Phase 1 — Empty check:
   If the result sample is empty or shows zero rows, output
   immediately:
      REJECT: query returns empty result

Phase 2 - Duplicate check:
   If the result sample contains duplicate rows (exactly the same
   row multiple times) or has an excessive number of rows, output
   immediately:
      REJECT: <explanation of the issue>

Phase 3 — Content check (only if result is non-empty):
   Evaluate whether the returned data answers the question:
   - Whether the returned values make sense with regard to the
     question?
   - Correct values / aggregations?
   - Right level of detail?
   - No obviously missing rows or columns?
   - Does answer the question without including irrelevant
     information?

Output exactly one of:
   ACCEPT
   REJECT: <reason>
```

User

```
SCHEMA:
[DDL of the database]

QUESTION:
[natural language question]

SQL QUERY:
[submitted SQL]

RESULT SAMPLE (first [n] of [total] rows):
[one row per line, or "(empty result set)"]
```

Figure 10: Judge prompt. The judge receives the database schema, the original natural language question, the submitted SQL query and a sample of at most five rows of its result, together with the total row count. It answers `ACCEPT` or `REJECT` with a reason; on rejection the reason is passed back to the answering model as the critique for the rewrite (Figure 9). A response counts as an acceptance when its first or last line begins with `ACCEPT`, and as a rejection otherwise. Italic grey text: runtime placeholders.

D SCHEMA-WITHHELD ENVIRONMENT

Training and evaluation use the same environment, and the database schema is never part of the prompt. Figure 11 shows the prompt an episode starts from: a system turn with the task, the tool list and the execution-plan scaffold, and a user turn that states that the schema is not shown and gives the question. Everything the model learns about the database it has to obtain through the tools of Table 9.

Tool database. Behind the tools, every episode gets its own in-memory SQLite database. It holds a subset of the schema's tables, sampled by the procedure of Appendix F, together with their rows. The gold tables are always part of that subset, so the question stays answerable, and the table order is shuffled, so `list_tables` does not reveal which tables the question needs.

Turns. Each model turn is one tool call in JSON form, and the tool result is appended to the conversation before the next turn. Every tool result also states how many tool turns remain. An episode has at most six tool turns. After the sixth, or after four `run_query` errors in a row, the model gets a final turn in which it is asked to submit. If the answer submitted in that final turn is empty or does not parse, it is replaced by the last query that executed without error in the episode. An episode ends when `submit_sql` is called.

Execution-plan scaffold. The system prompt asks the model to reason through every query it writes in the order the database executes it: `FROM`/`JOIN` with the join keys checked through `foreign_keys` or `join_path`; `WHERE` with literals verified in the data; `GROUP BY`/`HAVING`; `SELECT` with exactly the requested columns; and `ORDER BY`/`LIMIT`. It then writes the query, runs it, and checks the result before submitting. The scaffold is part of the prompt, so it is present in training and in evaluation alike.

Submit gate. Early in training, the environment refuses a `submit_sql` call unless the same query has already been run through `run_query` in an earlier turn, so the model submits only after seeing the result of its query. The gate is a rule of the environment, not a reward term. It is active in each episode with a probability that decays linearly from 1.0 to 0.0 over the first 80 training steps, after which the model has to keep verifying its queries on the strength of the reward alone. The forced final turn bypasses the gate so that every episode terminates, and the gate is never active at evaluation.

System

```
You are an expert SQL assistant. Given a SQLite database schema, generate a single
valid SQL query that answers the user's question. Think carefully, then return only
the SQL query with no additional explanation.

You have access to the following database exploration tools. Use them to understand
the data before writing the final SQL.
Available tools:
list_tables, describe_table(table_name), foreign_keys(table_name),
join_path(table_a, table_b), sample_rows(table_name, limit),
run_query(sql), distinct_values(table_name, column_name, limit),
submit_sql(sql)
For every response output ONLY a single JSON object:
{"tool": "<name>", "args": {"<param>": <value>, ...}}
[one example call per tool]

Start with list_tables and explore as needed. When a query joins tables, get
the join keys from foreign_keys or join_path instead of guessing column names.
Before submitting, verify any literal you filter or join on with distinct_values or
sample_rows, and test your final query with run_query. Only call submit_sql after
seeing a plausible result.

Before you write ANY SQL, reason through it in your thinking in the order the database
executes it:
1. FROM/JOIN: which tables, joined on which keys.
2. WHERE: which rows survive each filter, using verified literals.
3. GROUP BY / HAVING: what is aggregated, over which groups.
4. SELECT: exactly the columns the question asks for.
5. ORDER BY / LIMIT: what "most", "highest", "top N" means.
Then write the SQL that implements that plan, run it, and check the result against step
4 before submitting.
```

User

```
The database schema is not shown. Use the exploration tools to discover the relevant
tables, columns and values.

Question: [natural language question]

Explore the database with the available tools, then call submit_sql with your final
answer.
```

Figure 11: Starting prompt of a schema-withheld episode, abridged. The system turn combines the task, the tool list and the execution-plan scaffold; the user turn gives the question without any schema. Italic grey text marks placeholders and abridged parts.

E TABLE VALUE SYNTHESIS

Generation order. Each schema is parsed into a directed foreign-key graph and decomposed into strongly connected components with Tarjan's algorithm (Tarjan, 1972). The components are populated in topological order, so every table is generated after the tables it references. Inside a cyclic component, references to tables of the same component are left out while its tables are generated, and a repair pass afterwards replaces every intra-cycle foreign-key value that does not resolve with a valid key of its parent.

Prompt. For each table the model receives the table's DDL with a column summary (type, primary key, nullability, foreign-key target), the names of up to five neighbouring tables found by a breadth-first search over the foreign-key graph in both directions, up to five of the table's original SchemaPile rows as a style reference where available, and the schema's domain description, which it is told to treat as a hard constraint on the vocabulary of free-text and categorical values. The allowed values of every foreign key are not listed as text but injected into the execution environment as a Python list; the prompt names the variable and previews its first five values. The model returns a Python function that builds the table as a `pandas` DataFrame, and a name-generation helper is available for columns holding personal names.

Row budgets and foreign-key skew. A table holding foreign keys into two or more tables is classified as a fact or junction table, any other table as a lookup or dimension table. Lookup tables receive a row budget drawn from 20–100 and fact tables one drawn from 200–400, seeded per table so that re-runs reproduce. The upper bound of the fact range is set by generation reliability: in a calibration run the share of tables whose generated function failed stayed at or below 3.1% up to 400 requested rows, but rose to 20.0% at 400–800 and 56.9% at 800–1400. Fact tables are instructed to reuse a small subset of parent keys across many rows instead of spreading the pool evenly, and lookup tables to spread their foreign keys evenly. Measured over all 9,259 schemas for 929,083 foreign-key columns in total, the most-used fifth of parent keys holds a median 66.9% of a fact table's rows against 34.2% under uniform sampling of the same size, and 82.4% of fact-table foreign-key columns exceed the uniform share by at least ten points, while lookup-table foreign keys come out flatter than uniform (44.4% against 52.2%). Non-empty lookup tables hold a median of 59 rows and fact tables 297, with 99.9% and 99.1% of them inside their ranges.

Validation and repair. The function runs in an isolated subprocess under a time limit that scales with the requested row count. Duplicate primary keys are removed, keeping the first occurrence, and foreign-key values that do not resolve are replaced by valid parent keys **(C4)**. If execution fails or violations persist after repair, the error is returned to the model, which regenerates the function, up to three attempts per table. Tables that exhaust their attempts stay empty. Together with 2,885 tables in 123 schemas that the filler's DDL parser does not recover and therefore never populates, this leaves 4.8% of the 1,103,669 tables empty.

F GRAPH-AWARE SCHEMA CONTEXT ALGORITHM

The schema is withheld from the prompt (Appendix D), but the tools still operate on a concrete database, and its size sets how hard the search for the relevant tables is. SQaLe schemas hold a median of 113 tables, and training only on full schemas would never show the model a small database, so each episode is given a database whose number of tables is drawn at random. The tables are chosen so that the question stays answerable and the additional tables are plausible neighbours of the gold tables rather than an unrelated scatter.

Graph construction. Given the schema's DDL, we extract every `CREATE TABLE` statement, yielding a set of tables $\mathcal{T}$ and their DDL snippets $\{d_t\}_{t\in\mathcal{T}}$. Foreign-key relations are parsed from both explicit `FOREIGN KEY ...` `REFERENCES` constraints and inline `REFERENCES` clauses and symmetrised into an undirected graph $G = (\mathcal{T}, E_u)$, so that a foreign key from t to t' makes the two tables neighbours.

Gold tables. The anchor set $R \subseteq \mathcal{T}$ contains every table the gold query references, together with the tables annotated as relevant for the question. All anchors are included. If none of them can be resolved against $\mathcal{T}$, a random table is used as the seed instead.

Propagation along foreign keys. The target number of tables n is drawn uniformly from $\{4, \dots, n_{\max}\}$ with $n_{\max} = 64$. Starting from the anchors, the subset grows along the foreign-key graph: a shuffled frontier of neighbours is expanded one table at a time, and the neighbours of each added table join the frontier, until n tables are included or the frontier is exhausted. If the connected neighbourhood is too small, the remaining slots are filled with random tables. Algorithm 1 states the procedure. The episode's database then holds the DDL and rows of the selected tables in shuffled order.

Algorithm 1 Graph-aware table sampling for one episode

Require: tables $\mathcal{T}$, DDL map $\{d_t\}$, undirected graph G, anchors R, maximum size $n_{\max}$
Ensure: DDL of the episode's database

$n \sim \mathrm{Uniform}\big(\{4, \dots, n_{\max}\}\big)$
$S \leftarrow R$ **if** $R \neq \emptyset$ **else** {a random $t \in \mathcal{T}$}
$F \leftarrow \textsc{Shuffle}\big(\bigcup_{t\in S} N_G(t)\big)$ ▷ frontier of foreign-key neighbours
while $F \neq \emptyset$ **and** $|S| < n$ **do**
  $c \leftarrow \textsc{Pop}(F)$
  if $c \notin S$ **then**
    $S \leftarrow S \cup \{c\}$
    $F \leftarrow F \,\|\, \textsc{Shuffle}\big(N_G(c) \setminus S\big)$
  end if
end while
if $|S| < n$ **then**
  $S \leftarrow S \cup \textsc{Sample}\big(\mathcal{T} \setminus S,\ n - |S|\big)$ ▷ disconnected or small neighbourhood
end if
return $\bigoplus_{t\in\textsc{Shuffle}(S)} d_t$

Here $N_G(t)$ is the set of neighbours of t in G, $\|$ appends to the frontier and $\oplus$ concatenates DDL statements. For a schema with at most n tables the procedure returns the full schema.

G REWARD FUNCTION

During training, the reward function has four tiers, checked in the following order. An *equivalent* query, whose result set matches the gold one, is rewarded 3.0. An *executable* query, which runs but does not match the gold result, is rewarded $1.0 + 0.9\,F_1 + b$. Here $F_1 \in [0, 1]$ is the multiset F_1 between the candidate's result set and the gold one. A candidate that returns the gold rows with extra columns is compared on the gold columns only, so a query recovering most of the gold rows outranks one recovering none. The bonus $b = 0.1\,\rho + 0.1\,\mathbf{1}[\text{PLURALITY}]$ adds two preferences, a *join-path* term and a *group-consensus* term: ρ is the fraction of the query's joins that follow a declared foreign key, and, inspired by the self-consistency that leaderboard systems use (Pourreza et al., 2025; Sheng and Shuai, 2025a), the indicator rewards candidates sharing the most common result set in their group. Because GRPO normalises within a group, the second term activates only when candidates disagree. The *executable* reward tops out at $1.0 + 0.9 + 0.2 = 2.1$, short of the 3.0 for an *equivalent* query, so no amount of shaping can rank a wrong query above a right one. A *parsable* query, which does not execute, is rewarded 0.5. Anything else is *invalid* and gets 0.

H SQL-LENGTH CURRICULUM

A uniformly sampled GRPO batch reproduces the length distribution of the training corpus, and in SQaLe half of all gold queries are shorter than 150 characters. The curriculum changes which questions are drawn at each step. The reward, the objective and the environment stay the same.

Complexity score and buckets. Each training question is scored by the character length of its gold SQL after comments are stripped, string literals are blanked and the inside of quoted identifiers is replaced. The masking keeps long column names and free-text literals from inflating the score. The corpus is split into four buckets at the 50th, 80th and 95th percentiles of this score. The percentiles are computed over the whole filtered training corpus rather than over the sampled pool, so the edges do not depend on the pool size. Table 6 shows the resulting buckets for SQaLe. On this corpus, length is a proxy for query constructs: common table expressions and window functions are almost absent from the two short buckets and concentrated in the two long ones. The long buckets are also mostly labelled *hard*, so hard questions are included in training.

Table 6: Length buckets of the SQaLe training corpus. CTE and Window fn. give the share of gold queries in the bucket that use a common table expression or a window function; Hard is the share labelled with the hard difficulty.

Bucket	Masked chars	Corpus (%)	CTE	Window fn.	Hard (%)
Short	0–149	49.8	0.00	0.00	3
Medium	149–278	30.2	0.01	0.01	23
Long	278–575	15.0	0.38	0.08	59
Very long	575+	5.0	0.86	0.19	76

Training pool. The training pool is filled by bucket with shares of 0.20, 0.25, 0.25 and 0.30 from short to very long. A uniform pool of that size would hold only about 200 very-long questions, which a tail-weighted mixture would repeat every few steps. The stratified pool holds about 1,230.

Sampling schedule. At each step the $B = 18$ questions of a batch are drawn from a mixture over the four buckets. For the first 400 steps, the batches come from the short bucket only. The mixture then starts at $(0.40, 0.30, 0.20, 0.10)$, which is close to the natural mixture of the corpus, and moves linearly to $(0.15, 0.20, 0.25, 0.40)$ over the next 1,000 steps, after which it is held for the remaining 400 of the 1,800 steps. Starting close to the natural mixture matters for GRPO: a policy that is suddenly given mostly questions it cannot solve produces groups in which all $G = 8$ candidates fail, and such groups carry no advantage signal. The batch slots are assigned to buckets deterministically by largest remainder, so every data-parallel rank draws the same batch.

Floor against forgetting. After the warm-up, no bucket's weight may fall below 0.15. If the schedule would push a bucket under this floor, the difference is taken from the buckets above it, in proportion to how far above the floor each one is. The end mixture above already satisfies the floor, so it does not bind in these runs. It guarantees that short queries keep contributing gradient for the whole run, so the policy is still trained on them when the mixture reaches the tail. With a floor of 0.15, roughly 22 of the 144 episodes in every step come from the short bucket.

Per-corpus settings. SQaLe uses the settings above. SynSQL's gold queries span a much narrower length range, so it receives a milder version of the curriculum with three buckets instead of four. The edges at 149 and 278 characters are kept, but there is no very-long bucket: the long bucket covers every query from 278 characters up and takes over the very-long bucket's shares. This gives pool shares of 0.20, 0.25 and 0.55, and a mixture that moves from $(0.40, 0.30, 0.30)$ to $(0.15, 0.20, 0.65)$ on the same schedule, a 400-step warm-up on the short bucket followed by a 1,000-step anneal. BIRD's range is narrower still, and it is trained without a curriculum, i.e. with uniform sampling.

I GRPO TRAINING HYPERPARAMETERS

The GRPO update. At each step $B = 18$ questions are drawn and the policy rolls out $G = 8$ candidate episodes per question, giving 144 episodes per update, with advantages computed within a question's group. Groups whose episodes all receive the same reward carry no gradient and are resampled, following DAPO (Yu et al., 2026), and the KL penalty is adapted to hold the divergence from the reference policy at 0.005. Each run is 1800 roll-out steps with a single inner epoch, i.e., one optimiser update per step, at a learning rate of 1×10^{-5} on a cosine schedule with 5% warm-up.

Table 7 lists the settings shared by M_{SQaLe}, M_{BIRD} and M_{SynSQL}. Apart from the training corpus, the only setting that differs between them is the strength of the length curriculum (Appendix H).

Table 7: GRPO hyperparameters. Identical across M_{SQaLe}, M_{SynSQL} and M_{BIRD} except for the strength of the length curriculum (Appendix H). Episodes per update is $B \times G = 18 \times 8 = 144$.

Hyperparameter	**Value**
Model & data	
Base model / initial policy	Qwen3.5-2B (base checkpoint)
Reference policy	base model, re-anchored every 111 updates
Training corpus	SQaLe / SynSQL-2.5M / BIRD train
SQaLe difficulty bands	simple, moderate, hard
Environment	
Schema in prompt	no (withheld; tools only)
Protocol	execution-plan scaffold (App. D), 7 tools + `submit_sql`
Tool turns / final turn	6 / 1
Submit gate probability (App. D)	$1.0 \rightarrow 0.0$ over the first 80 steps
Consecutive query-error cap	4
Tokens per tool turn	2048
Tokens for the final turn	3072
Episode token budget	12288
Rollout	
Questions per step B	18
Candidates per question G	8
Episodes per update	144
Rollout temperature	1.0 (no top-p / top-k truncation)
Rollout backend	asynchronous in-process vLLM
Optimiser	
Optimiser	AdamW (8-bit where the engine shares a card)
Learning rate	1×10^{-5}
LR schedule	cosine, 5% linear warm-up
Rollout steps	1800
Inner epochs per rollout	1
Gradient accumulation	1
Optimiser updates	one per roll-out step
GRPO objective	
Surrogate	token-level ratio, PPO clip
Clip ε (low / high)	0.2 / 0.28 (DAPO clip-higher)
Importance-ratio cap	2.0
Loss aggregation	token sum / 1024
Advantage normalisation	none; clipped to ± 2.0
Negative-advantage weight	0.7
KL estimator	k_3
KL target / β init / $\beta_{\max}$	0.005 / 0.02 / 1.0
Dynamic-sampling rounds	2
Reward	
Semantic equivalence	3.0
Executable (wrong result)	$1.0 + 0.9\, F_1$
Parsable (non-executable)	0.5
Invalid SQL	0.0
Column-subset credit	enabled, discount 0.9
Join-path bonus	$0.1\times$ FK-supported join share
Group-consensus bonus	0.1
Equivalence decision	deterministic result-set comparison (no LLM judge)
Evaluation & checkpointing	
In-training monitoring	BIRD dev (monitoring only)
Monitoring samples	300
Evaluation mode	schema withheld, as in training
Evaluation / save interval	every 50 steps
Checkpoint selection	last checkpoint (step 1800)
Selection on BIRD dev	none

J Compute–Accuracy Evaluation Protocol

All models in the compute–accuracy comparison of Section 4.2, baselines and M_{SQaLe} alike, are served through vLLM, so that token accounting is identical across variants and no part of the comparison depends on a different inference stack. Input and output token counts are taken from the per-request usage that the server reports, summed over every turn a strategy takes.

From those counts we compute an analytical estimate of the compute each model–strategy pair spends on a question,

$$\text{FLOPs} \approx 2\,N\,T + 4\,L\,d\,S^2,$$

where N is the model parameter count, T the total number of processed tokens summed across all turns of a strategy, and the second term captures the attention cost with L layers, hidden size d, and prompt length S. Because T is summed across turns, the multi-turn tool-use pipeline accumulates FLOPs over every exploratory call rather than over the final generation alone, which is what makes the single-shot and agentic strategies comparable on one axis.

Figure 12 repeats Figure 6 with every model labelled.

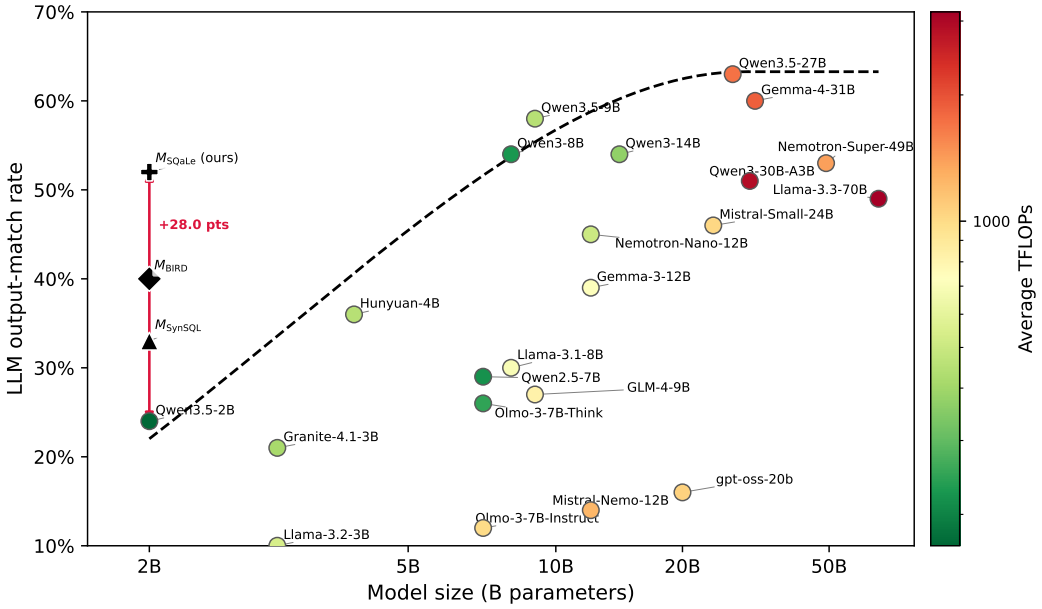


Figure 12: Labelled version of Figure 6: accuracy on 100 moderate questions from the SQaLe test set against parameter count, with the average TFLOPs each model spends per question as colour. M_{SQaLe}, M_{BIRD} and M_{SynSQL} are Qwen3.5-2B trained with GRPO on SQaLe, BIRD train and SynSQL-2.5M respectively; the red arrow is the gain of M_{SQaLe} over the untrained Qwen3.5-2B. Points at identical coordinates share a label. The x-axis is compressed so that the range below 10B is wider than on a log scale, and the dashed guide has its vertex fixed at 29B, with its height fitted to the frontier models (63%).

K VALUE GROUNDING

This appendix details the value analysis of Section 4.3. It uses the evaluation run behind Tables 4 and 5a (300 questions from SQaLe test, schema withheld) and pools its three schema-size levels. 162 of the 300 questions have gold SQL with at least one string literal, which gives 627 (episode, literal) pairs per model.

Categories. Every single-quoted literal in the gold SQL is compared case-insensitively, with `LIKE` wildcards removed, against the episode, and falls into exactly one of three categories, checked in this order. A literal is *discovered* if it occurs in a tool result the agent received before submitting, i.e. the agent read the value from the data. It is *inferred* if it occurs in no tool result but the agent's final SQL contains it, i.e. the agent took it from the question or guessed it without looking it up. Otherwise it is *never seen*. The order matters: a literal that the question states and that the agent also looks up counts as discovered, so the categories measure whether the agent checks a value against the data, not whether the value was available to it.

Why the two routes are not interchangeable. A wrong inferred literal does not fail loudly. `WHERE status = 'Complete'` against a column holding `'CMPLT'` parses, executes and returns an empty result, which is indistinguishable from a question that genuinely has no matches. An agent that queries for the value gets one of two outcomes, the value or its absence, and the second is recoverable inside the episode. An agent that guesses cannot tell the two apart, because it never asked. Equal shares of literals *seen* are therefore not equal reliability.

An inferred literal also comes from a closed world: the wording of the question, or string distributions the model learned during training. Both are fixed once the model is deployed. A discovered literal comes from the database in front of it. Real schemas carry value vocabularies – status enums, product codes, internal category names, abbreviations – that no training corpus contains and no user phrases correctly, which is the condition criteria C9 and C10 of Section 2.1 describe. Reading the data is the only mechanism that resolves them.

Literals stated in the question. Most literals are stated in the question (528 of the 627 pairs), and the benefit of looking them up is small: all three trained models carry them into their SQL equally often (Table 8). The difference between the models lies in the 99 pairs whose literal the question leaves out. Here M_{SQaLe} searches the data about twice as often and uses the correct value in 55% of cases, against 41% for M_{BIRD} and 39% for M_{SynSQL}.

Table 8: String literals of the gold SQL, split by whether the question states them. *Looked up*: the literal occurs in a tool result before submission. *In final SQL*: the agent's submitted query contains it. Shares in %.

	Stated in the question (n=528)		**Not stated** (n=99)	
Model	Looked up	In final SQL	Looked up	In final SQL
M_{SQaLe}	**92.8**	91.7	**60.6**	**54.5**
M_{BIRD}	42.2	91.5	27.3	41.4
M_{SynSQL}	58.5	91.7	37.4	39.4
Qwen3.5-2B	37.7	71.2	22.2	27.3

The same pattern holds when the questions are asked in all eight phrasings of the corpus instead of the verbose original: there, M_{SQaLe} looks up 79% of the literals, against 41% for M_{BIRD}, 53% for M_{SynSQL} and 33% for the base model.

L TOOL SET

The exploration interface exposes six callable tools implemented by a single wrapper class. The tool schemas are provided in OpenAI function-calling format during vLLM evaluation, and the same methods can be passed as bound callables to Ollama, which derives JSON schemas from Google-style docstrings. The question-answering loop of Section 2.2 uses exactly these six. The training environment of Section 4.1 exposes the same six plus the two foreign-key tools listed at the foot of Table 9, which matter there because the schema is withheld and join keys cannot be read off a DDL in the prompt. In the training environment, an episode ends when the model calls `submit_sql`. Each episode is limited to 7 turns, and the model is reminded to submit after the sixth turn.

Tool	Signature	Purpose and constraints
`list_tables`	`() → str`	Lists all table names in the database (newline-separated). Intended as the first call to establish the search space.
`describe_table`	`(table_name: str) → str`	Returns the `CREATE TABLE` DDL for a given table via `sqlite_master`. Used to recover exact column names, types, and key constraints.
`sample_rows`	`(table_name: str, limit: int=2) → str`	Returns a small sample of rows (JSON array of objects) for value-level inspection. The effective limit is capped by a runtime row budget.
`distinct_values`	`(table_name: str, column_name: str, limit: int=10) → str`	Returns distinct values of a column (newline-separated) to discover enums/categories and canonical string forms. The limit is capped.
`run_query`	`(sql: str) → str`	Executes a *read-only* query for exploration/validation. Only `SELECT` (or `WITH ... SELECT`) is permitted. Returns a header with the number of rows shown and (when available) the total row count, followed by a JSON-formatted sample.
`submit_sql`	`(sql: str) → str`	Records the final SQL answer and terminates the exploration loop. Should not be used for exploration.
Additionally available in the training environment of Section 4.1		
`foreign_keys`	`(table_name: str = None) → str`	Returns declared foreign-key relations as `child.col → parent.col`, for one table or for the whole database.
`join_path`	`(table_a: str, table_b: str) → str`	Returns the shortest chain of foreign keys connecting two tables, so a join can be grounded in declared keys rather than guessed from column names.

Table 9: Database exploration tools available to the LLM for NL→SQL.

M Models Used in the Generation Pipeline

Table 10 lists the language models behind the pipeline in Section 2.2, from the repository annotation to the question style variation. The table value synthesis and every stage after it use one model, Qwen3.6-35B-A3B in its FP8 release, served with vLLM. The judge is a separate call to the same model, with the prompt of Appendix C. Subgraph sampling, the integrity checks and the conversion to the released format involve no model.

Table 10: Language models used in the SQaLe generation pipeline.

Stage	**Model**	**Output**
Repository annotation	`Qwen/Qwen3.5-9B`	Domain description per repository
Table value synthesis	`Qwen/Qwen3.6-35B-A3B-FP8`	Python function per table
Question generation	`Qwen/Qwen3.6-35B-A3B-FP8`	Natural language questions
SQL generation	`Qwen/Qwen3.6-35B-A3B-FP8`	Gold SQL via tool use
Judge	`Qwen/Qwen3.6-35B-A3B-FP8`	Accept or reject with critique
Question style variation	`Qwen/Qwen3.6-35B-A3B-FP8`	Seven style variants