High-Quality Complex Text-to-SQL Data Generation through Chain-of-Verification

Zhang Yuchen[1,2], Gao Yuze[1,2], Chen Bin[1], Li Wenfeng[1], Sun Shuo[1], Su Jian[1]
[1]Institute for Infocomm Research @ A*STAR, Singapore
[2] CNRS@CREATE LTD, Singapore
{zhangyuc,gaoy1,bchen,liwf,suns1,su_jian}@a-star.edu.sg

Abstract

Can today's Text-to-SQL (T2S) benchmarks still stretch modern LLMs? We argue no. Spider1.0 and BIRD, painstakingly hand-built, remain small, costly, and skewed toward middle complex SQL. Meanwhile, LLM-generated corpora are inexpensive but often superficial and fragile suffering from shallow nesting, semantic drift, template fatigue, and insufficient quality check. We address this gap with a Chain-of-Verifications (CoVe) framework that turns a handful of expert-labelled seeds into a large, reliably checked dataset at a fraction of the usual cost. The resulting corpus, AIGT2S, delivers: **(1) 18k** Question–SQL pairs across 113 databases, **41–77%** larger than current English sets; **(2) 55%** queries in the Ultra band of our four-level difficulty taxonomy; **(3) 87.5%** inter-annotator agreement; **(4)** $\geq$**80%** labour and $\geq$**98%** monetary savings versus earlier efforts. Baselines including GPT-4o, Llama3, RESD-SQL, and MAC-SQL, achieve at most **56%** execution accuracy, indicating substantial room for improvement.

1 Introduction

Can a T2S benchmark be both large and accurate without consuming thousands of annotator-hours? Unfortunately, we didn't observe that in our survey.

Manual benchmarks: expensive yet limited in terms of complexity.

LLMs have transformed data generation, yet T2S evaluation still leans on Spider1.0 (Yu et al., 2019) and BIRD (Li et al., 2023b). Earlier benchmarks, from single-table WikiSQL (Zhong et al., 2017) to multi-table Spider1.0, DuSQL (Wang et al., 2020), and BIRD-SQL, and domain-specific sets Squall (Zhao et al., 2022), KaggleDBQA (Lee et al., 2021), Yelp & IMDB (Yaghmazadeh et al., 2017) share the same limitations: heavy annotation cost, modest scale, and constrained structural diversity across schemas.

LLM-generated data: cost-effective but superficial and fragile. LLMs streamline synthesis, but current attempts remain at surface-level. Borisov et al. (2023) show data diversity hinges on prompt design, while Gretel-SQL (Meyer et al., 2024) achieves volume but forgoes quality checking. Cross-table joins and realistic schema interplay are largely missing.

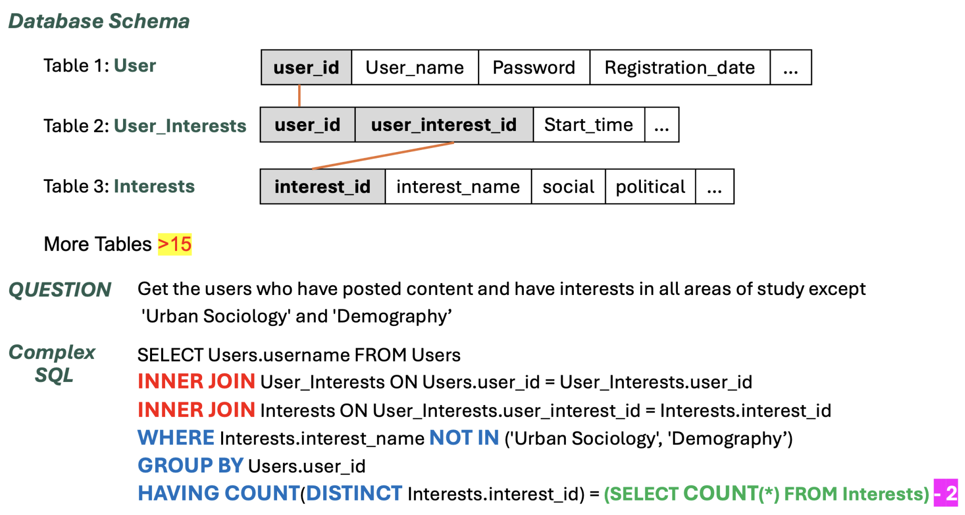
In practice, they show limited structural variety, Question–SQL misalignment, repetitive template-based patterns, poor schema utilisation, and minimal human validation. These shortcomings obscure an accurate assessment of model performance and increase data leakage risk, ultimately undermining the benchmark's credibility.

Three barriers to better benchmarks: Creating a robust, complex T2S set is hard because **1)** real-world schemas are often private, **2)** annotation requires expertise across natural language, SQL, and database semantics, which is costly to source, and **3)** crowdsourcing does not scale well as complexity rises.

Reasoning with self-validation: promising yet under-used. Chain-of-Thought(COT) prompting (Wei et al., 2022) and logic-aware variants (Zhao et al., 2024; Dhuliawala et al., 2024) reduce hallucination via step-wise self-explanation, but are rarely integrated into dataset construction. Most T2S synthesis still trades scale for rigor (or vice versa) and lacks pipelines that pair multi-schema SQL generation with systematic validation.

Our Proposed Solution 'CoT + CoVe': We present a multi-turn prompting pipeline that couples CoT generation with a CoVe filter. A manual annotated "gold" seed set familiarizes the LLM with high-quality examples, after which auto critics assess validity of structure, semantics, and schema. These procedures reduce annotation hours by more than **80%**.

Introducing Our AIGT2S: Our pipeline yields 18k instances spanning 113 synthetic, schema-



QUESTION: Get the users who have posted content and have interests in all areas of study except 'Urban Sociology' and 'Demography'

```
SELECT Users.username FROM Users
INNER JOIN User_Interests ON Users.user_id = User_Interests.user_id
INNER JOIN Interests ON User_Interests.user_interest_id = Interests.interest_id
WHERE Interests.interest_name NOT IN ('Urban Sociology', 'Demography')
GROUP BY Users.user_id
HAVING COUNT(DISTINCT Interests.interest_id) = (SELECT COUNT(*) FROM Interests) - 2
```

Figure 1: An example of text-to-SQL data instance from AIGT2S

rich databases (over 2.3k tables and 16k columns). Comparing to Spider1.0 and BIRD, AIGT2S is 41–77% larger, contains more than twice the percentage of Ultra-complex queries. Yet it was created for less than US$1.5k and 60 person-hours, only a few percent of traditional curation costs. It records an inter-annotator agreement of **87.5%** between AI and human expert, the highest reported for synthetic T2S benchmark to date. Figure 1 shows an AIGT2S example, additional examples can be found in the Appendix A.

Baselines: We benchmark 5 models selected to span the main T2S paradigms: **1)** GPT-4o (OpenAI, 2024): a frontier general-purpose LLM, **2)** Llama-3-8B (AI@Meta, 2024): open LLMs with light supervision, fine-tuned in zero/one-shot regimes, **3)** RESDSQL (Li et al., 2023a): a schema-/syntax-aware decoder; **4)** MAC-SQL (Wang et al., 2025): a multi-agent planner–refiner framework. In contrast to their strong Spider results, no model exceeds 56% execution accuracy on AIGT2S, highlighting performance remains far from saturation.

Summary of Contributions: 1) Scalable Data Synthesis Framework: we designed a CoT + CoVe framework that reduces annotation cost and timeby over 80% while preserving high annotation quality at 87.5% inter-annotator agreement. **2) AIGT2S Benchmark:** We release a substantial Question–SQL corpus, it doubles the proportion of Ultra-complex queries relative to Spider1.0 and BIRD.

3)Open-source Full-suite Toolkit: All prompts, verification scripts, complexity-analysis, and evaluation toolkit will be released upon acceptance.

Collectively, these contributions provide a more challenging and transparent testbed for advancing compositional generalization and robustness in T2S modeling.

2 Data Generation and Chain of Verification (CoVe) Pipeline

In this section, we present our 3-stage framework to generate and validate high-complexity T2S data.

Stage I: Large-Scale Synthesis with GPT-3.5
Figure 2 illustrates the workflow and detailed

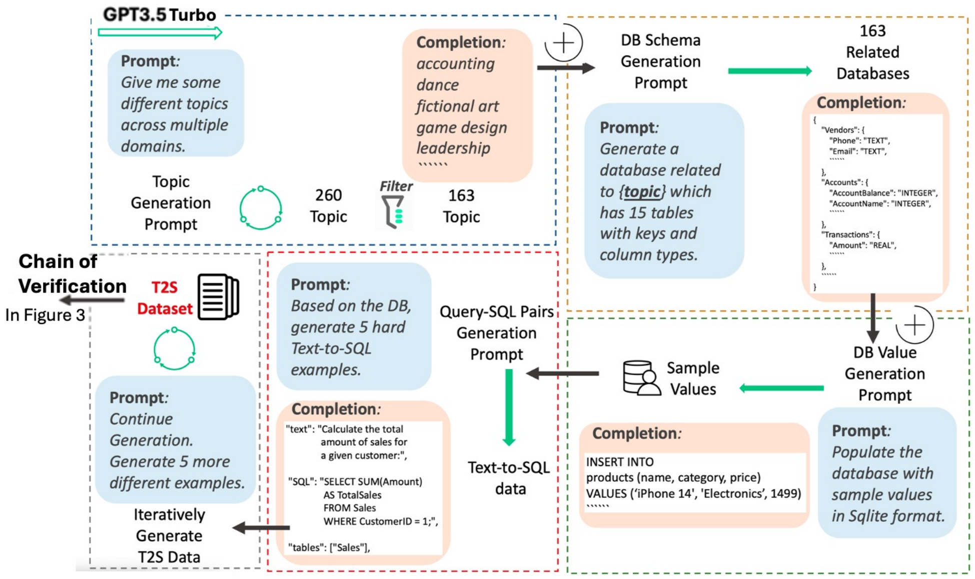
prompt of our large-scale data creation process using GPT-3.5 Turbo, which balance of cost-effectiveness and the ability to meet the high standard generation requirement for T2S query and database schema/content.

Topic Generation: We prompt GPT-3.5 iteratively "List unrelated real-world domains ..." until 163 distinct topics remain after de-duplication and synonym removal. Additional example topics are provided in the Appendix B

Schema & sample data: For each topic the model is asked to design a >=15 tables relational schema, including keys, types, and realistic column names, then populate it with SQLite formatted rows. This produces privacy-safe yet structurally rich databases.

Drafting Question–SQL pairs: Given the schema context, GPT-3.5 emits batches of 5 “hard” instances, and repeat 200 times per database to maximise linguistic and structural diversity (nested queries, multi-way joins, set operations).

Comprehensive post-processing and lightweight sanitisation: A regex-centric


GPT3.5 Turbo
Prompt: Give me some different topics across multiple domains.
Completion: accounting dance fictional art game design leadership ``````
Topic Generation Prompt
260 Topic
Filter
163 Topic
DB Schema Generation Prompt
163 Related Databases
Prompt: Generate a database related to {topic} which has 15 tables with keys and column types.
Completion:
DB Value Generation Prompt
Sample Values
Prompt: Populate the database with sample values in Sqlite format.
Completion:
Query-SQL Pairs Generation Prompt
Text-to-SQL data
Prompt: Based on the DB, generate 5 hard Text-to-SQL examples.
Completion:
Chain of Verification
In Figure 3
T2S Dataset
Prompt: Continue Generation. Generate 5 more different examples.
Iteratively Generate T2S Data

```
{
  "Vendors": {
    "Phone": "TEXT",
    "Email": "TEXT",
    ``````
  },
  "Accounts": {
    "AccountBalance": "INTEGER",
    "AccountName": "INTEGER",
    ``````
  },
  "Transactions": {
    "Amount": "REAL",
    ``````
  },
  ~~~
}
```

```
INSERT INTO
products (name, category, price)
VALUES ('iPhone 14', 'Electronics', 1499)
~~~
```

```
"text": "Calculate the total
         amount of sales for
         a given customer:",

"SQL": "SELECT SUM(Amount)
        AS TotalSales
        FROM Sales
        WHERE CustomerID = 1;",

"tables": ["Sales"],
```

Figure 2: GPT-3.5 Turbo driven Data Generation Workflow

pipeline is executed as the final step to eliminate residual inconsistencies or hallucinations. It **1)** unifies case or symbol variant identifiers, **2)** normalises primitive types and value formats across the five canonical SQL value types, **3)** strips some DML statements (UPDATE/DELETE/INSERT) to keep the corpus purely query-oriented, **4)** detects and repairs alias mis-bindings and orphaned columns via the SQL_metadata(Brencz, 2019) parser, and **5)** drops any pair whose SQL references schema elements absent from the accompanying database. All rules and illustrative corrections are documented in Appendices C & D.

Stage II: Expert Analysis for CoVe Design and Annotation Quality

Two senior annotators independently label 2k randomly sampled pairs (1k dev / 1k test). After 4 calibration rounds their inter-annotator agreement reaches 93%. These gold labels serve two roles: **1)** a quality yard-stick and **2)** seed material for designing automated checks in Stage III. Total human time: 60 hours.

Stage III: CoVe: turning "good" into "gold"

After bulk synthesis we enlist a stronger model, GPT-4-32K to "interrogate" every candidate pair. As shown in Figure 3, the model reasons step-by-step, acts as an automated reviewer, and is itself calibrated on a 2k example mini-gold set. Quality is measured by agreement score: the fraction of cases where model and humans concur on the SQL's correctness.

Baseline: Before CoVe checking(**Stage I**), the 35k raw pairs sit at 62.1% agreement, fall below benchmark-ready.

Round 1 Semantic fidelity check: The initial verification round, we posed the prompt: 'Based on the Database Schema content {#Schema} and User Question {#Question}, does my SQL match the user query?".

Pairs flagged "no" are purged. Agreement on the dev split leaps to **84.8%**, a **+22.7 pt** improvement.

Round 2 Structural integrity check: Based on the challenges identified during the annotation phase (**Stage II**), we designed the prompt of second verification round with: "Is there any redundancy or unnecessary complexity(see Appendix E)? Does the query correctly align with the provided schema? Are the condition functions used correctly? Please re-evaluate the SQL query, is it correct or not? "

Only queries receiving an unequivocal "yes" survive. Agreement rises further to **87.5%**, **25.4%** agreement improvement over stage I result. This agreement is **2.2%** higher DuSQL's, which involve lots of manual validation.

Final dataset curation: The 2-stage filtering process distilled the corpus to 18,081 high-quality Question–SQL pairs covering 113 databases, generated for under USD 1.5k in API fees and approximately 60 hours of expert annotation. Empirical evidence from CoVe indicates that inserting a lightweight, LLM-mediated critique step bridges nearly the entire quality gap to full human review, yet still scales effortlessly to tens of thousands of high-complexity T2S instances.
```

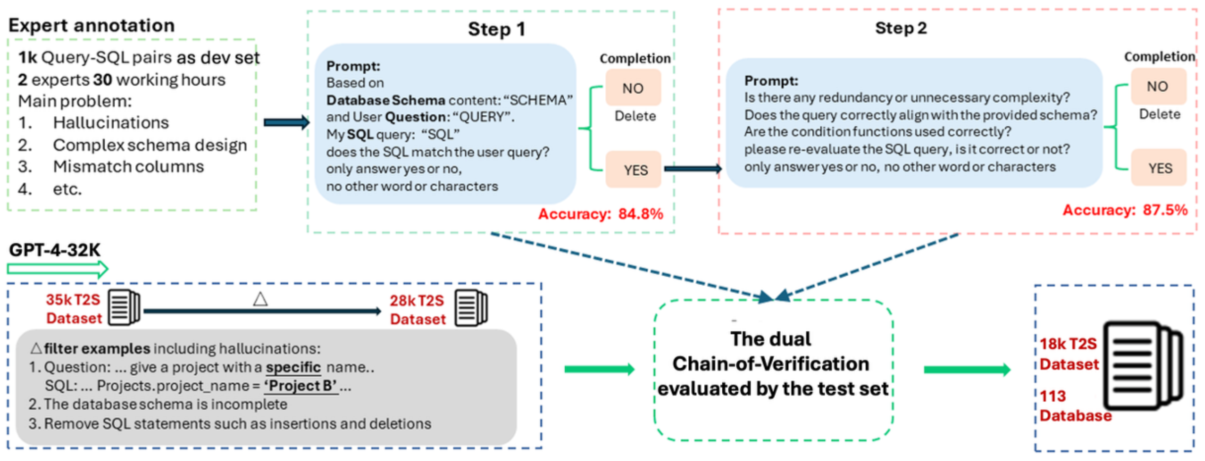

Expert annotation
1k Query-SQL pairs as dev set
2 experts 30 working hours
Main problem:
1. Hallucinations
2. Complex schema design
3. Mismatch columns
4. etc.
Step 1
Prompt:
Based on
Database Schema content: "SCHEMA"
and User Question: "QUERY".
My SQL query: "SQL"
does the SQL match the user query?
only answer yes or no,
no other word or characters
Completion
NO
Delete
YES
Accuracy: 84.8%
Step 2
Prompt:
Is there any redundancy or unnecessary complexity?
Does the query correctly align with the provided schema?
Are the condition functions used correctly?
please re-evaluate the SQL query, is it correct or not?
only answer yes or no, no other word or characters
Completion
NO
Delete
YES
Accuracy: 87.5%
GPT-4-32K
35k T2S Dataset
28k T2S Dataset
△ filter examples including hallucinations:
1. Question: ... give a project with a specific name..
SQL: ... Projects.project_name = 'Project B' ...
2. The database schema is incomplete
3. Remove SQL statements such as insertions and deletions
The dual Chain-of-Verification evaluated by the test set
18k T2S Dataset
113 Database

Figure 3: Chain of Verification Workflow

3 Complexity Analysis

Does AIGT2S merely add volume, or does it stretch models in genuinely new ways?

We measure hardness across three dimensions: surface length, structural richness, and schema breadth, and benchmark all figures against Spider and BIRD, two de-facto "hard" English T2S datasets.

Surface Length: Average query length rises from 35 (Spider) and 41.6 (BIRD) to 61 tokens in AIGT2S, nearly doubling lexical load.

Structural richness: Four metrics capture internal query complexity: **Joins:** 83% of AIGT2S queries contain multi-way joins (Spider 60%, BIRD 76.5%). **Group and Having:** 43% contain a GROUP BY and HAVING block (Spider 8%, BIRD 1.5%). **Nested Subqueries:** 16% embed at least one subquery (Spider 13%, BIRD 9%). **SQL Component Count:** rises from 5.0 (Spider) / 5.39 (BIRD) to 10.2, reflecting heavier use of set operations, arithmetic expressions, and window-style aggregates(full catalogue in Appendix F).

Schema breadth: Questions in AIGT2S reference an average of 3.3 tables (Spider 1.6, BIRD 2.0), drawn from databases averaging 20 tables / 139 columns (Spider 5 / 27, BIRD 7.5 / 51). This expansion forces models to perform cross-entity reasoning over substantially larger schemas, a critical bottleneck for execution-accurate decoding.

Difficulty grades: We propose a new 4-level classification for SQL query difficulty based on SQL structural complexity and compositional criteria, as in Figure 4: Basic, Advanced, Expert, and Ultra. This scheme more clearly reflects the structural and logical complexity of queries, including highly intricate ones involving nested structures, set operations, and conditional logic. Applying this refined hardness taxonomy yields: **Ultra:** 55%, **Expert:** 24%, **Advanced:** 21%, **Basic:** 0%, versus BIRD (25 / 23 / 52 / 0) and Spider (20 / 21 / 37.5 / 21.5).

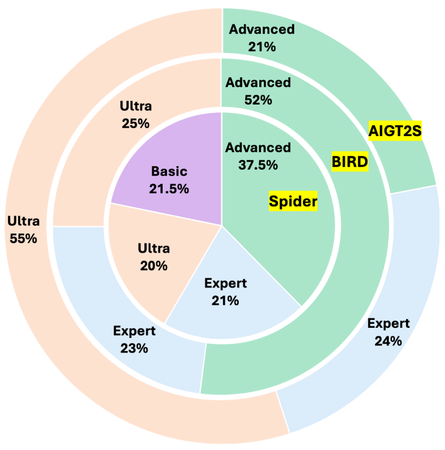


Figure 4: Comparison of difficulty levels between AIGT2S, Spider and BIRD.

Figure 4 shows how AIGT2S rebalances toward the upper tail, with over half of its queries falling into Ultra, doubling Spider's and BIRD's share. Roughly two-thirds of AIGT2S queries employ constructs unseen by prior difficulty toolkits, e.g., percentile aggregates or multi-level alias chaining. Over 5,000 cases store sub-query results as aliases subsequently reused across JOIN / HAVING / WHERE, introducing long-range dependency tracking. For more detailed criteria please refer to Appendix G.

AIGT2S is not only larger but decisively deeper: it couples longer queries with denser logic and broader schemas, offering a benchmark that better mirrors production-grade SQL workloads.

4 Experimental Protocol & Key Findings

Data Split: The corpus is partitioned into 16k CoVe-verified training pairs and two 1k expert sets for validation and test. The dev split guides hyper-parameter search and drives GPT-4 verification; the test split remains unseen until final scoring.

Evaluation Metric: Exact-match scoring collapses under heavy nesting and alias re-ordering. We therefore use Execution Accuracy (EA): a prediction is correct only if both gold and candidate SQL return the identical, non-null result on a populated database. See Appendices H and I for the database synthesis procedure that guarantees non-null, de-duplicated answers.

Benchmarked System: We apply two light supervision regimes, zero-shot and one-shot fine-tuning, to the LLM baselines: **GPT-4o** and **Llama-3-8B** (including its fine-tuned variant); while the structure-aware decoder **RESDSQL** and the multi-agent planner–refiner framework **MAC-SQL** are evaluated under their canonical training/inference setting. Prompts and hyperparameters appear in Appendix J.

Model	Exec. Acc.
Finetuned Llama3 $8B_{\textit{Zero-Shot}}$	53.8%
Finetuned Llama3 $8B_{\textit{One-Shot}}$	56.0%
Vanilla Llama3 $8B_{\textit{Zero-Shot}}$	27.7%
Vanilla Llama3 $8B_{\textit{One-Shot}}$	27.0%
GPT-$4o_{\textit{Zero-Shot}}$	47.1%
GPT-$4o_{\textit{One-Shot}}$	53.2%
RESDSQL-$base_{\textit{Spider Checkpoint}}$	3.9%
RESDSQL-$base_{\textit{Trained on AIGT2S}}$	11.2%
RESDSQL-$large_{\textit{Trained on AIGT2S}}$	12.8%
MAC-SQL + GPT $4o_{\textit{Zero-Shot}}$	54.7%

Table 1: Execution accuracy of various SOTA models on the AIGT2S benchmark

Table 1 lists the performance of each model,**1)** A compact 8B model(Finetuned Llama), when lightly fine-tuned on AIGT2S, outperforms GPT-4o; **2)** RESDSQL's syntax-first decoding struggles on multi-table, alias-heavy queries; and **3)** MAC-SQL's planner–refiner strategy (selector $\rightarrow$ decomposer $\rightarrow$ refiner) remains useful for large-schema, multi-step questions, as reported in prior work; this motivates a next step: coupling a multi-agent planner/refiner with our fine-tuned Llama-3-8B on AIGT2S. **4)** Overall, performance remains far from saturation, leaving ample scope for improvement.

5 Conclusion

We present a scalable CoT + CoVe pipeline that transforms GPT-3.5 generations into high-quality, deeply verified T2S examples. The resulting AIGT2S dataset is significantly larger, more complex, and more rigorously validated than previous English benchmarks. Our experiments show that even strong LLMs like GPT-4o and fine-tuned Llama3 struggle, which highlights that challenging data is more important than model size when true SQL reasoning is required.

Limitation

Despite the gains delivered by the CoT + CoVe pipeline, three limitations remain:

Incomplete domain coverage. While AIGT2S spans deep joins and advanced analytics, it still under-represents domain-specific operators and business conventions that appear in production SQL. Extending the dataset to multi-turn, dialogue-style interactions should help surface these long-tail patterns.

Dependence on proprietary LLMs. The verification stage currently relies on GPT-4, constraining transparency and portability. Ongoing work explores retrieval-augmented curricula and curriculum-based fine-tuning for open-source models so that comparable quality can be achieved without closed tools.

Residual human arbitration. Edge cases in CoVe still require expert review, introducing cost and potential bias. Future research will integrate self-consistency critics and static program analyses to push this last mile of validation toward full automation.

Addressing these issues will further increase the robustness, generalisability, and reproducibility of large-scale, high-complexity T2S benchmarks.

Acknowledgments

This research is partially supported by the programme DesCartes funded by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

References

AI@Meta. 2024. Llama 3 model card. Accessed: 2024-09-16.

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

Maciej Brencz. 2019. sql-metadata. https://github.com/macbre/sql-metadata. Version 2.12.0, accessed 2025-07-29.

Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. 2024. Chain-of-verification reduces hallucination in large language models. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 3563–3578, Bangkok, Thailand and virtual meeting. Association for Computational Linguistics.

Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. 2021. KaggleDBQA: Realistic evaluation of text-to-SQL parsers. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2261–2273, Online. Association for Computational Linguistics.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023a. Decoupling the skeleton parsing and schema linking for text-to-sql.

Jinyang Li, Binyuan Hui, GE QU, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023b. Can LLM already serve as a database interface? a BIg bench for large-scale database grounded text-to-SQLs. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Yev Meyer, Marjan Emadi, Dhruv Nathawani, Lipika Ramaswamy, Kendrick Boyd, Maarten Van Segbroeck, Matthew Grossman, Piotr Mlocek, and Drew Newberry. 2024. Synthetic-Text-To-SQL: A synthetic dataset for training language models to generate sql queries from natural language prompts.

OpenAI. 2024. Gpt-4o system card. Accessed: 2024-09-16.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025. Mac-sql: A multi-agent collaborative framework for text-to-sql. *Preprint*, arXiv:2312.11242.

Lijie Wang, Ao Zhang, Kun Wu, Ke Sun, Zhenghua Li, Hua Wu, Min Zhang, and Haifeng Wang. 2020. DuSQL: A large-scale and pragmatic Chinese text-to-SQL dataset. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6923–6935, Online. Association for Computational Linguistics.

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

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Sqlizer: query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA).

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. 2019. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *Preprint*, arXiv:1809.08887.

Chen Zhao, Yu Su, Adam Pauls, and Emmanouil Antonios Platanios. 2022. Bridging the generalization gap in text-to-SQL parsing with schema expansion. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5568–5578, Dublin, Ireland. Association for Computational Linguistics.

Xufeng Zhao, Mengdi Li, Wenhao Lu, Cornelius Weber, Jae Hee Lee, Kun Chu, and Stefan Wermter. 2024. Enhancing zero-shot chain-of-thought reasoning in large language models through logic. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 6144–6166, Torino, Italia. ELRA and ICCL.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

A Example Appendix

We randomly enumerate 3 samples from different hardness and unparsable situation.

Advanced

"Advanced" queries either satisfy no more than two rules involving multiple aggregations, selections, or conditions, using no more than one keyword from [WHERE, GROUP BY, ORDER BY, LIMIT, JOIN, OR, LIKE, HAVING], and no keywords from [EXCEPT, UNION, INTERSECT, NESTED]; or they contain exactly two keywords from [WHERE, GROUP BY, ORDER BY, LIMIT, JOIN, OR, LIKE, HAVING] with fewer than two instances of multiple aggregations, selections, or conditions, and no keywords from [EXCEPT, UNION, INTERSECT, NESTED].

1. **Database:** `surgery.json`
   Text: Find all the patients who have undergone a surgery with a surgery cost greater than $10,000 and display their first and last name along with the surgery type and surgery cost.
   Involved Tables: ['Patient', 'Surgery']

```
SELECT p.‘Patient First Name‘, p.‘
    Patient Last Name‘, s.‘Surgery
    Type‘, s.‘Surgery Cost‘
FROM Patient p
INNER JOIN Surgery s ON p.‘Patient
    ID‘ = s.‘Patient ID‘
WHERE s.‘Surgery Cost‘ > 10000;
```

2. **Database:** `electronic_music.json`
   Text: Find the tracks that have been favorited by more than three users.
   Involved Tables: ['Tracks','UserFavorites']

```
SELECT Tracks.track_title
FROM Tracks JOIN UserFavorites ON
    Tracks.track_id = UserFavorites.
    track_id
GROUP BY Tracks.track_id
HAVING COUNT(DISTINCT UserFavorites.
    user_id) > 3;
```

3. **Database:** `big_data.json`
   Text: Find the number of sales made by each salesperson in the month of June.
   Involved Tables: ['sales']

```
SELECT salesperson_id, COUNT(*) AS
    number_of_sales
FROM sales
WHERE date >= '20230601' AND date <
    '20230701'
GROUP BY salesperson_id;
```

Expert

Queries are classified as "Expert" if they satisfy more than two rules involving multiple aggregations, selections, or conditions with no more than two keywords in [WHERE, GROUP BY, ORDER BY, LIMIT, JOIN, OR, LIKE, HAVING] and no keywords in [EXCEPT, UNION, INTERSECT, NESTED]. Alternatively, "Expert" queries can contain more than two but less than or equal to three keywords from [WHERE, GROUP BY, ORDER BY, LIMIT, JOIN, OR, LIKE, HAVING] while satisfying fewer than two rules involving multiple aggregations, selections, or conditions. Queries that only contain one keyword from [WHERE, GROUP BY, ORDER BY, LIMIT, JOIN, OR, LIKE, HAVING], no rules from [multiple aggregations, selections, conditions] but exactly one keyword from [EXCEPT, UNION, INTERSECT, NESTED] also fall into the "Expert" category.

1. **Database:** `leadership.json`
   Text: Find all users who have a note with the word "campaign" in the title or text.
   Involved Tables: ['Users', 'Notes']

```
SELECT DISTINCT Users.username
FROM Users JOIN Notes ON Users.
    user_id = Notes.user_id
WHERE Notes.note_title LIKE '%
    campaign%' OR Notes.note_text
    LIKE '%campaign%';
```

2. **Database:** `cooking.json`
   Text: Retrieve the recipes that have a cooking time greater than the average cooking time of all recipes.
   Involved Tables: ['Recipes']

```
SELECT r.recipe_id, r.title
FROM Recipes r
WHERE r.cooking_time >
    (SELECT AVG(cooking_time)
    FROM Recipes);
```

3. **Database:** `sports_psychology.json`
   Text: Get a list of all athletes who have a goal related to "Endurance" or "Strength".
   Involved Tables: ['athletes', 'goals', 'athlete_goals']

```
SELECT athletes.name, goals.
    goal_name
FROM athletes INNER JOIN
    athlete_goals ON athletes.
    athlete_id = athlete_goals.
    athlete_id
INNER JOIN goals ON athlete_goals.
    goal_id = goals.goal_id
WHERE goals.goal_name = 'Endurance'
    OR goals.goal_name = 'Strength';
```

Ultra

These queries involve more than three keywords from [WHERE, GROUP BY, ORDER BY, LIMIT, JOIN, OR, LIKE, HAVING], contain more than one keyword from [EXCEPT, UNION, INTERSECT, NESTED], or satisfy at least three conditions involving multiple aggregations, selections, or conditions. The use of aliases or straightforward mathematical operations doesn't contribute to complexity determination.

1. **Database:** `food_and_drink.json`
   Text: Retrieve the name and total revenue generated by each customer who has placed at least one order in the past month, sorted in descending order by total revenue.
   Involved Tables: ['Customers', 'Order_Items', 'Menu_Items', 'Orders']

```
SELECT Customers.Customer_name, SUM(
    Order_Items.Quantity *
    Menu_Items.Price) AS
    Total_revenue
FROM Customers JOIN Orders ON
    Customers.Customer_ID = Orders.
    Customer_ID
JOIN Order_Items ON Orders.Order_ID
    = Order_Items.Order_ID
JOIN Menu_Items ON Order_Items.
    Menu_Item_ID = Menu_Items.
    Menu_Item_ID
WHERE Orders.Order_placed >= DATE('
    now', '-1 month')
GROUP BY Customers.Customer_ID
ORDER BY Total_revenue DESC;",
```

2. **Database:** `movies.json`
   Text: Retrieve the names of reviewers who have given the highest rating to all movies they reviewed.
   Involved Tables: ['Reviewer', 'Review']

```
SELECT rev.name
FROM Reviewer rev
WHERE NOT EXISTS
    (SELECT r.rating
    FROM Review r
    WHERE r.reviewer_id = rev.
        reviewer_id
    AND r.rating <
        (SELECT MAX(rating)
        FROM Review
        WHERE reviewer_id = rev.
            reviewer_id));",
```

3. **Database:** `western_films.json`
   Text: Show the titles and release dates of all Western movies that were released in the 1960s and have an average rating of at least 8.0, sorted by release date in ascending order.
   Involved Tables: ['Movie', 'Review', 'MovieGenre', 'Genre']

```
SELECT Movie.title, Movie.
    release_date
FROM Movie JOIN Review ON Movie.
    movie_id = Review.movie_id
JOIN MovieGenre ON Movie.movie_id =
    MovieGenre.movie_id
JOIN Genre ON MovieGenre.genre_id =
    Genre.genre_id
WHERE Genre.name = 'Western' AND
    strftime('%Y', Movie.
    release_date) BETWEEN '1960' AND
     '1969'
GROUP BY Movie.movie_id
HAVING AVG(Review.rating) >= 8.0
ORDER BY Movie.release_date ASC;",
```

Manipulative SQL Queries

These SQL queries primarily interact directly with the data in a table, performing actions such as removing records (DELETE), adding new records (INSERT), or altering existing records (UPDATE). AIGT2S contains a great number of these queries while they can not be parsed on Spider's script.

1. **Database:** `food_and_drink.json`

   Text: Delete all orders that were placed before January 1st, 2023.

   Involved Tables: ['Orders']

```
DELETE FROM Orders
WHERE Date_ordered < '2023-01-01';
```

2. **Database:** `accounting.json`

   Text: Add a new product to the Products table with a specific name, price, and supplier.

   Involved Tables: ['Products']

```
INSERT INTO Products (ProductName,
    Price, SupplierID)
VALUES ('New Product', 25.99, 4);
```

3. **Database:** `gardening.json`

   Text: Update the title of page with ID 1 to Äbout Our Company.̈

   Involved Tables: ['Pages']

```
UPDATE Pages SET title = 'About Our
    Company'
WHERE page_id = 1;
```

B Example of Topics Appendix

Table 2 refer to a sample of 45 diverse topics used in our dataset. The full dataset covers a wide range of domains, including science, arts, medicine, technology, everyday life and etc.

Accounting	Cooking
Action films	Cooking shows
Acupuncture	Copywriting
Advertising	Corporate social responsibility
Agriculture	Cosmetics making
Anthropology	Counseling
Archaeology	Country music
Astronomy	Craft beer brewing
Athletic training	Cybersecurity
Food and drink	Food and drink books
Forensic psychology	Forensic science
Game design	Game shows
Gardening	Genetics
Geology	Makeup artistry
Marine biology	Marketing books
Martial arts	Medicine
Meditation	Mobile app design
Mobile app development	Movies
Sculpting	Search Engine Optimization
Shipping and logistics	Smart lighting
Smart transportation	Snowboarding
Soapstone carving	Sociology
Sports medicine	

Table 2: Sample topics covered in the dataset.

C Post-Processing Appendix

A comprehensive post-process pipeline is employed as the last step to eliminate the inconsistencies or hallucinations, we used the following steps to address certain issues in the generated pairs:

Schema Error Correction:We developed sophisticated regular expressions to extract the required content related to the schema in the SQL queries. Based on the extracted contents, we addressed 2 schema errors.

First, duplicated tables and columns are merged. Such duplications resulted from inconsistencies in capitalization or the use of special characters like '_' and '/' during generation (such as: "order_id" vs "Order_Id").

Second, we applied keyword-based verification to correct the columns with incorrect data types and standardized the data format across the dataset. The verified and standardized data types include text, float, int, datetime, and Boolean(such as: all Boolean values are standardized to the correct format "True" and "False" instead of 1 and 0).

Schema Adjustment: As we prompt GPT3.5 to generate T2S instances over 200 times per database, GPT3.5 often loses track of the context and hallucinates, deviating from the database schema. To address the inconsistencies between the SQL queries and database schema:

First, columns that appeared fewer than 5 times across the generated T2S pairs were identified as rarely used and distantly related to the database schema. These columns, along with their associated data pairs, were removed to improve coherence(such as: "Security_Clearance_Level" column in "Employee" table).

Second, to address columns "hallucinated" from extensive iterations, we identified frequently occurring columns that were not originally part of the schema. These columns were verified for relevance, and if deemed relevant, they were incorporated into the corresponding database schema(such as: "Discount_Code" column in "CustomerOrders" table).

SQL Reserved Word Validation: The refinement step addresses 2 types of issues to improve accuracy further and refocus our data generation scope.

First, we removed all instances involving UPDATE, DELETE, or INSERT operations. As the generation focuses on database queries, database entry creation, deletion, and update are irrelevant.

Second, we correct the column aliasing errors occurring during the generation. Column aliases are commonly used in SQL to simplify column names for readability. However, in the data generated, we encountered issues like column name confusion or incorrect use of aliases, particularly in nested queries. These errors (e.g. Appendix D) could lead to incorrect SQL parsing. We corrected these problematic instances using the 'SQL_metadata' [1] library to ensure syntax correctness.

After these extensive post-processing steps, a total of 34,792 instances with higher quality were retained, representing a diverse and challenging set of T2S pairs ready for further analysis and annotation.

D Column Aliasing Errors Appendix

For example, in nested queries, the outer query references a column name that does not exist in the inner query or incorrectly references an alias.

Error: The outer query attempts to reference a non-existent column "b.Age".

```
SELECT a.Name, b.Department
FROM (
```

[1]https://github.com/macbre/sql-metadata

```
    SELECT Name, Age FROM Users
) AS a,
(
    SELECT Name, Department FROM
        Employees
) AS b
WHERE a.Age > b.Age;
```

Correct: Ensure that the outer query references the correct column name.

```
SELECT a.Name, b.Department
FROM (
    SELECT Name, Age FROM Users
) AS a,
(
    SELECT Name, Department, Age FROM
        Employees
) AS b
WHERE a.Age > b.Age;
```

E Redundancy and Unnecessary Complexity Criteria Appendix

When generating SQL queries, checking for redundancy or unnecessary complexity ensures that the generated queries are concise and efficient. Redundancy and complexity can manifest in the following ways:

1. Redundant conditions or operations

For example, repeated conditions or multiple operations on the same column, which lead to unnecessary duplication in the query.

2. Unnecessary subqueries

In some cases, subqueries can be simplified or replaced with regular queries, reducing nesting complexity.

3. Unnecessary sorting operations

If the result set does not require sorting, using ORDER BY should be avoided to reduce computational overhead.

4. Unnecessary joins

If some table joins are not required, removing them can reduce the complexity of the query.

F SQL Component Catalogue Appendix

Standard clauses: SELECT, WHERE, GROUP BY, ORDER BY, HAVING, LIMIT, JOIN, INTERSECT, EXCEPT, UNION, NOT IN, OR, AND, EXISTS, LIKE, nested queries.

Sorting: ASC, DESC

Join Variants: INNER, LEFT, RIGHT

Feature	Spider	AIGT2S
Order By	29.1%	22.5%↓
Group By	31.8%	42.5%↑
Having	7.6%	15.4%↑
Join	60.3%	83.5%↑
Nested	13.2%	15.7%↑
Involved tables	1.55	3.34↑
Average Tokens	35.4	61.3↑
Average Components	5.0	10.2↑

Table 3: Comparison of SQL feature distributions in Spider and AIGT2S datasets.

Arithmetic & Basic Aggregates: $+$, $-$, $\times$, $\div$, COUNT, SUM, AVG, MAX, MIN, MONTH, ROUND

Date/Time: DATE, YEAR, TIME, DATE_SUB, CURDATE, DATEADD, GETDATE, DATETIME, DATEDIFF, STRFTIME, TIMESTAMPDIFF, TIME_FORMAT, JULIANDAY

String: LENGTH, REPLACE, CONCAT, COALESCE, SUBSTR, INSTR, CHAR_LENGTH, TRIM

Advanced Aggregates: GROUP_CONCAT, PERCENTILE_CONT, STDEV

Conditional: CASE … WHEN … THEN … ELSE … END

Miscellaneous: CAST, WITHIN GROUP, CORR

G Hardness Criteria Appendix

We categorize the difficulty of SQL queries based on the presence and complexity of specific SQL components.

1. Definition of SQL Components

• SQL Components 1 (Basic Clauses)

WHERE, GROUP BY, ORDER BY, LIMIT, JOIN, OR, AND, LIKE, HAVING, BETWEEN, ASC, DESC

• SQL Components 2 (Aggregation and Functions)

DATE, COUNT, AVG, SUM, MIN, DISTINCT, STRFTIME, DATETIME, SUBSTR, ABS, FLOAT, YEAR, CAST, ROUND, JULIANDAY, TIME, MONTH, DATEDIFF, TIMESTAMPDIFF, GETDATE, DATEADD, CONCAT, COALESCE, INTEGER, INT, LENGTH

• SQL Components 3 (Nested Queries and Set Operations)

EXCEPT, UNION, INTERSECT, NESTED, SELECT(multi)

• SQL Components 4 (Conditional Expressions)

CASE, WHEN, THEN, ELSE

2. Hardness Classification

• Basic

The query contains only keywords from SQL Components 1, and each keyword appears at most once.

• Advanced

The query satisfies one of the following conditions:

① Contains up to three keywords from SQL Components 3, does not include any keywords from SQL Components 4, and includes fewer than two keywords from SQL Components 2.

② Contains exactly one keyword from SQL Components 3, does not include any keywords from SQL Components 4, includes fewer than three keywords from SQL Components 2, and each keyword from SQL Components 1 appears at most once.

• Ultra

The query satisfies one of the following conditions:

① Contains at least one keyword from SQL Components 4.

② Does not contain any keywords from SQL Components 4, but contains more than seven keywords from SQL Components 1, 2, or 3.

• Expert

Any query that does not fall into the Easy, Medium, or Ultar Hard categories.

H Synthetic Database Appendix

We have developed a more robust evaluation process with sophisticated database content generation capability to address these issues. This database generation process begins by parsing both the development and test datasets to extract the involved values in the dev and test cases. These values are then mapped to the appropriate positions in the database using column aliases. Afterward, the DB synthesis creates 1,000 rows of new data by inserting unique, valid values guided by the extracted values and schema types, filling any missing data to ensure that every test query yields a valid, non-null result. Furthermore, we provide each query's result is unique, avoiding inflating final performance.

We only consider a test-case correct if both the gold-standard SQL and the predicted SQL return the identical non-null value. Partially matched returned lists are counted as wrong cases.

I Impact of Missing Database Entries and Null Values on Performance Evaluation Appendix

Datasets such as Spider and DuSQL, which lack extensive database entries, often return null values due to missing or incomplete data. These null values can inflate performance estimates in the following ways:

1. Incomplete Query Execution

When the required data for a query is missing, the database may return null values or no results. This means that although the model generates an SQL query, it may not retrieve valid data. During evaluation, these queries might be considered "correct" simply because they did not result in errors, even though no meaningful data was retrieved. As a result, queries returning null values are erroneously counted as successful, inflating performance.

2. Tolerance for Null Handling

Some SQL engines or models may automatically skip over errors or incomplete data when null values are encountered. In such cases, the model might treat a null value as a valid output. This could lead to incorrect assumptions that the model has successfully processed the query, when in fact the result is simply a null value that doesn't contribute useful information.

3. Evaluation Metric Bias

When null values are frequent, evaluation metrics (e.g., accuracy, execution success rate) may be overestimated. Even if the generated SQL did not retrieve meaningful results, as long as it didn't produce errors, it might be considered a correct query. This can lead to misleading performance estimates.

J LLM Fine-tuning Appendix

We fine-tune Meta-Llama-3.1-8B-Instruct as our base model without using any (Q)LoRA adapters. The fine-tuning process is carried out with the torchtune toolkit, utilizing Fully Sharded Data Parallel

(FSDP) training. Modifications to certain components of the toolkit were made to suit our specific needs, and these adjustments will be released alongside our code.

The fine-tuning takes place on a GPU server equipped with 8 NVIDIA A6000 GPUs (48 GB each). Training is conducted over six epochs with a batch size of 3. We use the AdamW optimizer with a learning rate of 2e-5 and compute the loss using CrossEntropyLoss. To optimize performance, we employ BF16 precision for mixed-precision training.