

StreamSQL-Repair-Bench: Does Execution Feedback Let Agents Fix Event-Time Streaming SQL?

Zhuoxi Wang

Northeastern University Boston, USA

zhuoxi2000@gmail.com

Jingwen Hou

Northeastern University Boston, USA

hjingwen1229@gmail.com

Abstract.

Writing a streaming query that runs is easy; writing one that is correct under event-time semantics is not. Watermarks and late-data policies are subtle, and the kind of temporal reasoning they demand is something large language models (LLMs) handle poorly. The dominant fix in agentic coding is execution feedback: let the model run its code, see what comes out, and revise. We ask whether that loop actually rescues streaming-SQL errors. StreamSQL-Repair-Bench hands a model a streaming-SQL query with a single planted bug, an intent description, and a few input/output tests, and asks it to repair the query; ground truth comes from a small reference engine implementing Dataflow-model event-time semantics, so we grade by execution accuracy without running a cluster. We separate bugs into shallow (readable straight off the I/O) and deep (require simulating event time), and compare one-shot repair, blind retrying, and repair with execution feedback. Across five LLMs, execution feedback barely moves shallow bugs (approx.0.9 either way) but roughly doubles deep-bug accuracy (e.g. Claude Haiku 0.14->0.53, Gemini Flash 0.28->0.56; aggregate 0.24->0.49). A blind-retry control stays at or below one-shot, so the gain comes from the executed result, not from extra attempts. Yet feedback is not a cure: it repairs structural mistakes (wrong window type or size) but leaves watermark and late-data bugs unfixed, the same semantics that resist one-shot reasoning. The benchmark, reference engine, and harness need no streaming infrastructure.

Keywords:

large language models, agents, execution feedback, self-repair, streaming SQL, event time, benchmark

1. INTRODUCTION

Stream processing now routinely involves an LLM somewhere in the loop - drafting a pipeline from a description, fixing a query a user pasted, or adjusting a windowed aggregation, much as LLMs already draft batch SQL [1], [2]. The hard part of these queries is not syntax but *event-time semantics*: the result depends on the window type (tumbling, hopping, session), on the watermark that decides when a window may fire, and on the late-data policy that decides which out-of-order events still count [3], [4]. These are exactly the rules that make streaming SQL harder than its batch counterpart, and they are not visible from a query that merely parses and runs. That difficulty meets a known weakness. LLMs are poor at temporal reasoning even in natural language [5], and our own one-shot results make the point concretely for streaming SQL: on the event-time bugs in our benchmark, four of five models repair fewer than 30% unaided (Section 5). But one-shot

reasoning is not how agents are deployed. The standard pattern in agentic coding is *execution feedback*: the model writes code, an environment runs it, and the model revises based on what came out [6], [7], [8], [9], [10]. The premise is that running the code substitutes for reasoning the model cannot do unaided. So the natural question is: *when a streaming query is wrong for an event-time reason, does letting the agent run it and see the result let it fix the bug, or does the model fail to use the feedback because it cannot interpret event-time output either?*

We answer this with StreamSQL-Repair-Bench. Each item is a streaming-SQL query in a small documented dialect with exactly one planted bug, paired with a short statement of intent and a handful of input/output tests. The agent must return a corrected query; we score by *execution accuracy* (does the repaired query reproduce the reference output on a held-out replay?), the text-to-SQL standard [11], [12] lifted to streaming. Ground truth, and the agent's "run it" tool, are one and the same small Python engine we implement following the Dataflow model [3]; nothing requires Flink [13], Kafka, or a container. The design turns on two distinctions. First, we split planted bugs into *shallow* ones, visible directly in the input/output examples (e.g. a wrong aggregate), and *deep* ones that require mentally executing event time to localize (wrong window size or type, a mis-set watermark, a dropped late-data clause). Execution feedback should matter most for deep bugs. Second, we separate *execution feedback* from *mere retrying* with a blind-retry control that iterates the same number of turns but is told only "incorrect, revise," with no executed result. The difference isolates whether running the query, not just trying again, is the active ingredient. We report three findings. (1) Execution feedback gives almost nothing on shallow bugs (already near-ceiling) but roughly doubles deep-bug repair accuracy across models. (2) Blind retrying stays at or below one-shot, so the executed result, not the extra attempt, drives the gain. (3) Feedback is selective: it fixes structural bugs but not watermark and late-data bugs, the same event-time core that resists one-shot reasoning. We release the benchmark, the self-tested reference engine, and the agent harness.

2.Related Work

Text-to-SQL. Spider and BIRD [11], [12] and the enterprise-scale Spider 2.0 [14] grade generated SQL by execution accuracy, and LLM-based methods add task decomposition and self-correction on top [1], [2]. All of this is batch relational SQL; none touches streaming or event time. We borrow the execution-accuracy metric and reframe the task as *repair* under streaming semantics.

Self-repair and execution feedback. Teaching models to self-debug from executed output [6], verbal self-reflection [7], [9], tool-interactive critiquing [10], and reason-act loops [8] are the backbone of agentic coding; reinforcement from executed tests [15], tool use [16], and real-issue resolution benchmarks [17] push the same idea further. Tellingly, even on batch code self-repair is not a panacea: gains shrink once the cost of feedback is counted, and they hinge on the model's ability to diagnose its own output [18]. This literature is overwhelmingly batch code and SQL. We ask where the loop succeeds and fails on a semantics the underlying model finds hard: event time.

LLMs and stream processing. AutoStreamPipe [19] *generates* streaming pipelines (Flink/Storm/Spark code) with a multi-agent framework and ships a small generation benchmark, but scores by an Error-Free Score (counts of syntax, logic, and runtime-crash errors), not output correctness against a semantics-faithful oracle; it tests no event-time semantics (watermarks, late data, window types) and has no execution-feedback repair loop (its "resilience" is API retry). Our three load-bearing pieces are exactly its three gaps: execution-accuracy grading against a Dataflow-model oracle, an event-time bug taxonomy, and an execution-feedback repair loop. The reference semantics follow the Dataflow model [3] as realized in engines such as Flink [13] and Spark Structured Streaming [20], and the query surface follows continuous-query languages [4].

Temporal reasoning. LLMs handle temporal reasoning poorly even in natural language [5]. We localize an analogous weakness in a precise, executable setting, event-time query semantics, and ask whether execution feedback overcomes it.

3. Benchmark Design

Dialect and engine. We define a compact streaming-SQL subset over one append-only stream $s(\text{key}, \text{val}, \text{ts})$ with ts the event time. A query is an optional `CREATE TABLE` with a `WATERMARK FOR ts AS ts - INTERVAL 'D'` clause, an optional `SET allowed_lateness`, and a windowed `SELECT key, window_start, AGG(val)` over a windowing table function: `TUMBLE`, `HOP (slide, size)`, `SESSION (gap)`, or a processing-time `TUMBLE` over `DESCRIPTOR(proctime)`. The full grammar is given to the model. A query is parsed to a spec and executed by a small Python reference engine that implements watermark-driven firing and late-event dropping under Dataflow-model semantics [3] for all four window families and four aggregates (`sum`, `count`, `max`, `min`). The engine is both the ground-truth oracle and the agent's execution tool; it passes a hand-checked self-test, and we cross-validate its gold against Apache Beam (Section 5).

Repair task. A generator samples a correct (gold) spec and an out-of-order event stream, then applies exactly one *mutation* to produce a buggy query whose output actually differs from gold on the eval stream. The model receives: a one-line *intent* that deliberately withholds the numeric parameters (so it cannot transcribe a solution: a pilot found one-shot *generation* from a fully specified description saturates at 100% and carries no signal); the buggy query; and three input/output tests sampled from the same spec. It must return a corrected query. We grade by execution accuracy: the repaired query is run on the item's held-out replay and must reproduce the gold output exactly.

Bug taxonomy. We label each mutation *shallow* or *deep*. Shallow bugs are visible directly in the I/O examples: a wrong aggregate (`agg`) or a swapped `HOP` argument order detectable from window starts. Deep bugs require simulating event time to localize: wrong window size (`wsize`), wrong window type (`wtype`), and a mis-set or dropped watermark/late-data clause (`watermark`, `lateness`). The split lets us ask not just *whether* feedback helps but *on which bugs*.

Conditions. *One-shot*: repair once, no iteration. *Blind*: up to K turns, but each turn's feedback is only "incorrect, revise" plus any syntax error - no executed result. *Exec*: up to K turns with execution feedback, i.e. the failing input/output test diffs. Both iterative conditions see the identical starting prompt; they differ only in whether the feedback contains executed output. After each turn we record correctness on the hidden eval replay, so a single K -turn run yields the whole turn-budget curve.

4. Experimental Setup

We generate 180 items over 4 window families times 3 difficulty levels ($n=15$ per cell): 72 deep-bug and 108 shallow-bug items. We evaluate five LLMs (GPT-4o-mini, Gemini-2.5-Flash, Claude-Haiku-4.5, GPT-4o, and Claude-Sonnet-4.6) in the one-shot and exec conditions, with $K=5$ turns and temperature 0. The blind control is run on GPT-4o-mini and GPT-4o (the weak and strong ends). The total API cost is under USD 30.

5. Results

Table 1. Repair execution accuracy, one-shot $\rightarrow$ exec feedback, per model. Deep bugs ($n=72/\text{model}$) require event-time simulation; shallow bugs ($n=108/\text{model}$) are readable from the I/O. *Delta_deep* is the exec-minus-one-shot gain on deep bugs. The aggregate row pools all five models (360 deep, 540 shallow items).

Model	all	deep	Delta_deep	shallow
GPT-4o-mini	0.56 -> 0.59	0.18 -> 0.25	+0.07	0.81 -> 0.81
Gemini-2.5-Flash	0.66 -> 0.80	0.28 -> 0.56	+0.28	0.92 -> 0.96
Claude-Haiku-4.5	0.62 -> 0.77	0.14 -> 0.53	+0.39	0.94 -> 0.94
GPT-4o	0.56 -> 0.64	0.12 -> 0.32	+0.20	0.84 -> 0.85
Claude-Sonnet-4.6	0.71 -> 0.88	0.50 -> 0.81	+0.31	0.84 -> 0.94
Aggregate	0.62 -> 0.74	0.24 -> 0.49	+0.25	0.87 -> 0.90

Feedback helps deep bugs, not shallow ones. Table 1 and Fig. 1 carry the main result. On shallow bugs every model is already near ceiling one-shot (0.81-0.94) and execution feedback adds essentially nothing, because the mistake is visible in the examples without running anything. On deep bugs one-shot accuracy is low (0.12-0.28 for four of five models) and execution feedback roughly doubles it: Claude Haiku 0.14->0.53, Gemini Flash 0.28->0.56, GPT-4o 0.12->0.32; the aggregate over all models goes 0.24->0.49. GPT-4o-mini is the exception (0.18->0.25): the weakest model cannot turn the feedback into a correct revision. Claude Sonnet is strongest at both ends: highest one-shot deep accuracy (0.50) and the largest absolute exec result (0.81). The strongest model both reasons about event time best unaided and uses execution feedback best.

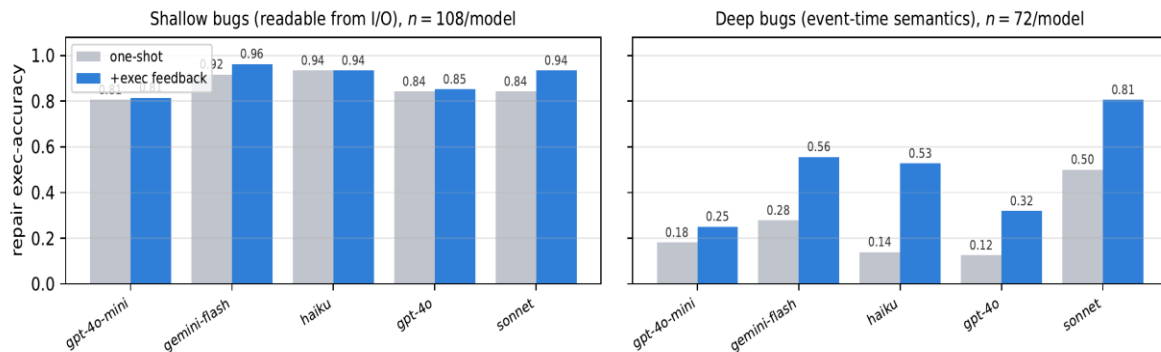


Figure 1. One-shot vs. execution feedback, per model. Feedback is flat on shallow bugs (left, $n=108/\text{model}$) and large on deep, event-time bugs (right, $n=72/\text{model}$). Bar labels are exec-accuracy values.

The executed result, not the extra attempt, is what helps. The blind-retry control (Fig. 2) separates "run it and see" from "try again." On deep bugs, blind retrying does not beat one-shot and on GPT-4o-mini slightly hurts it (one-shot 0.18, blind 0.14, exec 0.25; GPT-4o: 0.12, 0.15, 0.32). Iterating without executed output gives the model nothing to localize the bug with; the gain in the exec condition is specifically attributable to seeing the query's output.

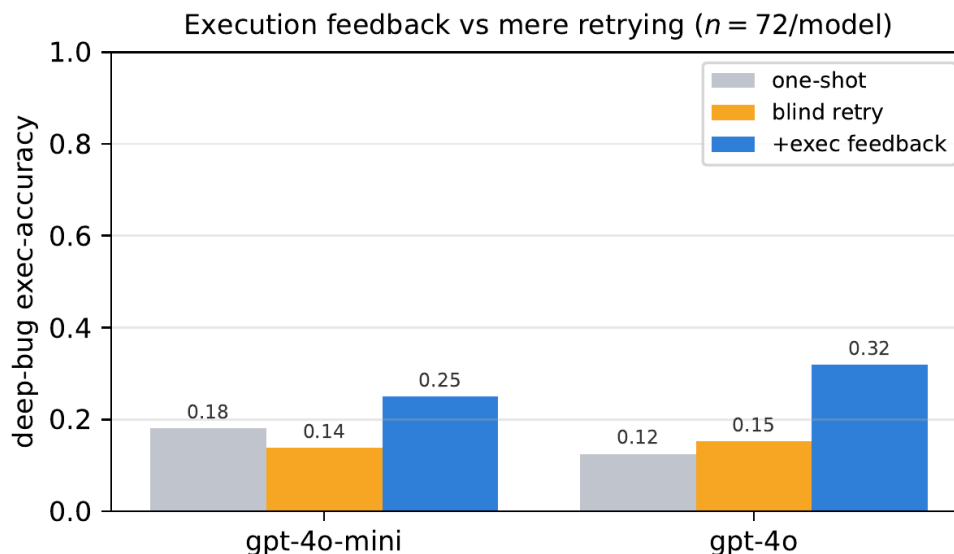


Figure 2. Deep-bug accuracy ($n=72/\text{model}$): one-shot vs. blind retry vs. execution feedback, on the weak and strong models. Blind retrying stays at or below one-shot; only execution feedback lifts accuracy.

Most of the gain is in the first few turns. Fig. 3 plots deep-bug accuracy against the turn budget. The capable models climb steeply over the first two to three rounds and saturate by about turn four (Gemini $0.28 \rightarrow 0.56$, Haiku $0.15 \rightarrow 0.53$); GPT-4o-mini's curve is nearly flat, again showing it cannot exploit additional feedback. A budget of three to four execution rounds captures essentially all the benefit.

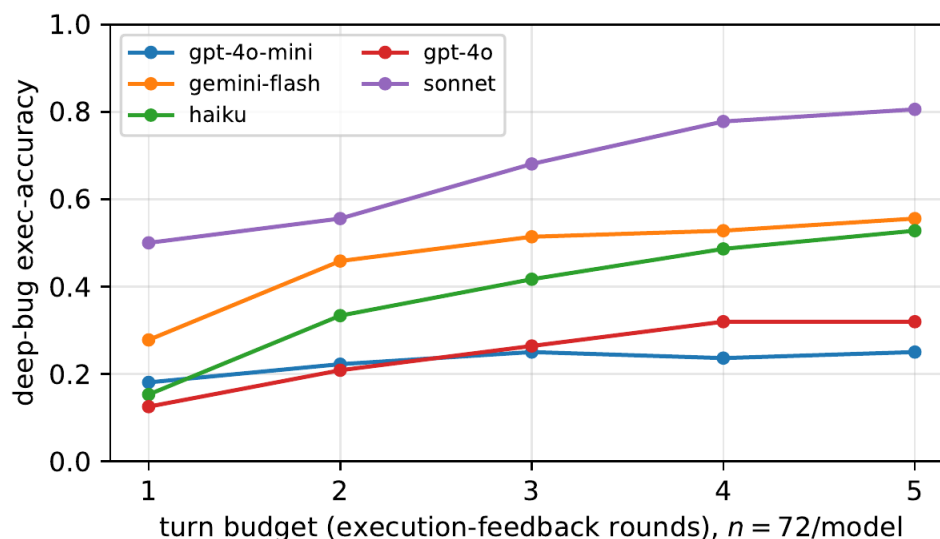


Figure 3. Deep-bug accuracy vs. execution-feedback turn budget ($n=72/\text{model}$). Gains concentrate in the first 2-3 rounds and saturate by approximately turn 4.

Feedback fixes structure, not time. Breaking the lift down by bug kind (Fig. 4) shows where it stops. Execution feedback repairs *structural* bugs, where the executed output makes the mismatch evident: wrong window type to hopping ($0.23 \rightarrow 0.64$), a dropped late-data clause ($0.20 \rightarrow 0.80$), wrong window size ($0.24 \rightarrow 0.47$). It barely moves *watermark* bugs: a dropped watermark ($0.33 \rightarrow 0.33$), the window-type-to-tumbling case ($0.32 \rightarrow 0.32$), and an added watermark ($0.00 \rightarrow 0.20$).

These are the event-time mechanisms that resist one-shot reasoning: when a window fires and which late events still count. That fits the broader finding that LLMs handle temporal reasoning poorly [5]. Seeing the wrong output tells the model *that* the timing is off but not *how* to set the watermark, because diagnosing it requires the very event-time simulation the model cannot perform.

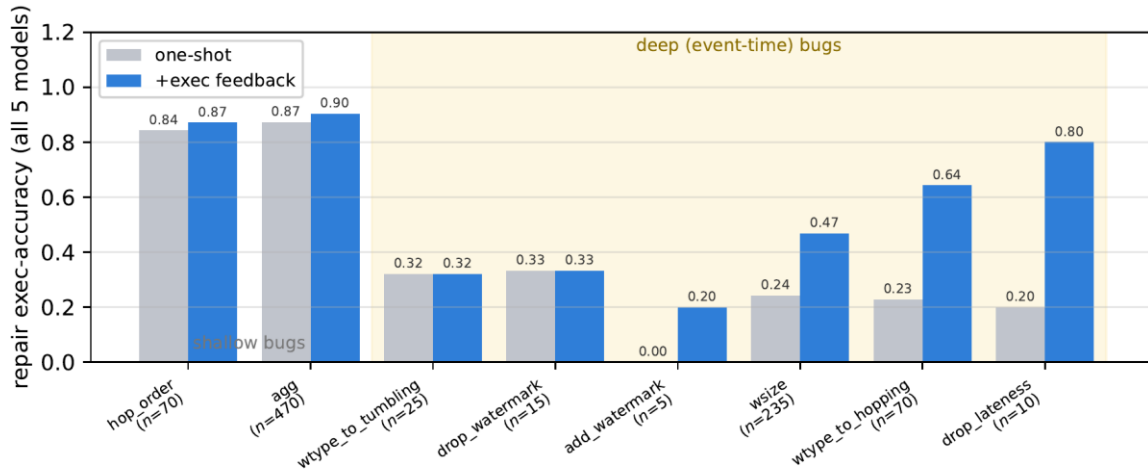


Figure 4. Per bug kind (all five models), one-shot vs. exec, with per-kind item counts. Feedback repairs structural bugs (window type/size, lateness) but not watermark bugs; the shaded block is the deep, event-time kinds.

Discussion and Limitations

Execution feedback is a real but *bounded* aid: it converts "my output is wrong" into a correct fix when the bug is structural and visible once run, but it cannot supply event-time understanding the model lacks. This refines a common assumption, that running the code substitutes for reasoning. It does, but only for the parts of the problem the model could verify by inspection, which echoes the finding that self-repair is no silver bullet even for batch code [18].

Gold validity. We cross-checked the reference engine against Apache Beam's DirectRunner, the canonical Dataflow-model implementation, on a 120-item tumbling+hopping sample: **113/120 (94)**. The residual seven are all out-of-order, event-time items differing only on the late-data boundary when the watermark exactly equals a window's end (our engine closes a window strictly, at $wm \geq end$); these are a documented convention, not aggregation or window-assignment errors, and grading is self-consistent regardless. Session and processing-time windows rest on the hand-checked self-test.

Other limitations. The mutation sampler leaves watermark-class bugs with small counts ($n=5-15$); we report them as directional and a full release will up-sample them for tighter intervals. The dialect is a single-stream subset (no joins or MATCH_RECOGNIZE), and items use synthetic streams; whether the same feedback dynamics hold on real workloads is open. The blind-retry control was run on two of the five models (GPT-4o-mini, GPT-4o), spanning the weak and strong ends.

6. Conclusion

StreamSQL-Repair-Bench tests whether execution feedback lets agents repair event-time streaming SQL. It does, but selectively: feedback roughly doubles deep-bug repair accuracy and beats blind retrying, yet leaves the watermark and late-data core unfixed, the same semantics that defeat one-shot reasoning. Running the query helps an agent correct what it can recognize, not

understand what it cannot. The benchmark and zero-infrastructure harness are released to support work on agents for streaming systems.

References

- M. Pourreza and D. Rafiei, "DIN-SQL: Decomposed in-context learning of text-to-sql with self-correction," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, and J. Zhou, "Text-to-sql empowered by large language models: A benchmark evaluation," *Proc. VLDB Endowment*, vol. 17, no. 5, pp. 1132-1145, 2024.
- T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernandez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1792-1803, 2015.
- A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121-142, 2006.
- Q. Tan, H. T. Ng, and L. Bing, "Towards benchmarking and improving the temporal reasoning capability of large language models," in *Proc. 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, 2023, pp. 14,820-14,835.
- X. Chen, M. Lin, N. Scharli, and D. Zhou, "Teaching large language models to self-debug," in *Int. Conf. on Learning Representations (ICLR)*, 2024.
- N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *Int. Conf. on Learning Representations (ICLR)*, 2023.
- A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhunoye, Y. Yang, S. Gupta, B. P. Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, and P. Clark, "Self-refine: Iterative refinement with self-feedback," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- Z. Gou, Z. Shao, Y. Gong, Y. Shen, Y. Yang, N. Duan, and W. Chen, "CRITIC: Large language models can self-correct with tool-interactive critiquing," in *Int. Conf. on Learning Representations (ICLR)*, 2024.
- T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task," in *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2018, pp. 3911-3921.
- J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo, X. Zhou, C. Ma, G. Li, K. C.-C. Chang, F. Huang, R. Cheng, and Y. Li, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," in *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023.
- P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bull. IEEE Computer Society Tech. Committee on Data Engineering*, vol. 38, no. 4, pp. 28-38, 2015.
- F. Lei, J. Chen, Y. Ye, R. Cao, D. Shin, H. Su, Z. Suo, H. Gao, W. Hu, P. Yin, V. Zhong, C. Xiong, R. Sun, Q. Liu, S. Wang, and T. Yu, "Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows," in *Int. Conf. on Learning Representations (ICLR)*, 2025.

- H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "CodeRL: Mastering code generation through pretrained models and deep reinforcement learning," in Advances in Neural Information Processing Systems (NeurIPS), 2022.
- T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," in Advances in Neural Information Processing Systems (NeurIPS), 2023.
- [17] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can language models resolve real-world github issues?" in Int. Conf. on Learning Representations (ICLR), 2024.
- [18] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama, "Is self-repair a silver bullet for code generation?" in Int. Conf. on Learning Representations (ICLR), 2024.
- A. Younesi, Z. Najafabadi Samani, and T. Fahringer, "AutoStreamPipe: LLM-assisted automatic generation of data stream processing pipelines," arXiv preprint arXiv:2510.23408, 2025.
- M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured streaming: A declarative API for real-time applications in apache spark," in Proc. ACM SIGMOD Int. Conf. on Management of Data, 2018, pp. 601-613.